WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Lab 4

Submitted By:
Hushmand Esmaeili
Joshua Fernandez
Erin Lee

Date Submitted: 12/16/2021
Course Instructor: Professor Carlo Pinciroli
Lab Section: RBE 3002 B'21

Introduction

In this RBE 3002 final laboratory, the entire curriculum was taught over the course of 7 weeks and the previous three labs were brought together for this final lab. The purpose of this final lab was to use ROS to control the Turtlebot3 in order to navigate and explore an unknown maze with the goal of mapping an unknown area. Packages created in previous labs were included with the addition of new ones to help achieve this goal. There were multiple objectives in this lab including:

- ➢ Localize the robot in the world
- ➢ Drive around the map while avoiding obstacles
- ➢ Find new areas to explore
- ➢ Find an optimal path to a point
- ➢ Identify when the map is complete
- ➢ Navigate back to a point on the map

This lab was broken down into three main sections: Phase 1, Phase 2, and Phase 3 where these objectives were implemented.

Methodology

This lab is broken into three phases: Phase 1 was navigating to frontiers of unknown environments until creating a full map filled with cspace while avoiding obstacles. Phase 2 was using the map from phase 1 and navigating back to the starting position while still avoiding obstacles. Phase 3 used the saved map from phase 1 to navigate to a specific goal using Rviz.

In order to perform this lab the architecture of the code was designed with the following logic and will be broken down for each phase:

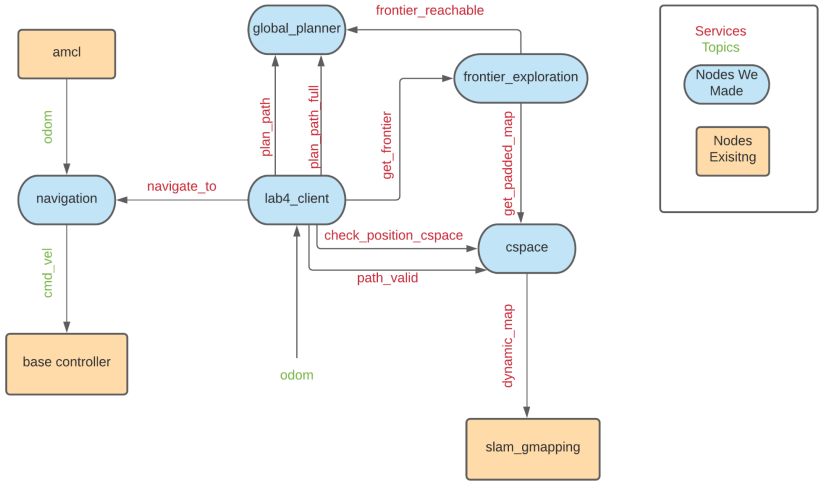

Figure 1: Node Structure of Final Project. Refer to key for more details.

The main node lab4_client controls the overall state machine that was used to move between states in order to complete each of the phases. There were three main states within the lab4_client, one for each state, and within phase states were sub states. In addition to the state machine which was used to complete this lab, lab4_client contained service requests offered by the other nodes in order to perform specific tasks.

Phase 1:

Within the phase 1 state, there were 5 sub states that the Turtlebot would switch between in order to complete phase one before moving onto phase 2. The first state STORE_START_LOCATION saved the initial starting x and y position of the turtlebot and saved it as a PoseStamped message. Then the state would switch to CHECK_POSITION which used the response from the service check_position_cspace and would receive a position and a boolean on whether the pose was valid. If it got the position then the Turtlebot would navigate to that position. If it did not get a valid position then the state would switch to GET_FRONTIER which would use the get_frontier service response to find a frontier to explore. In order for a position to be valid, it could not be in cspace or an obstacle.

Within the cspace node there is a function, calc_cspace, that was created in order to calculate the cspace of the map and make the obstacles bigger so that the Turtlebot did not crash into any in the real world. The function takes in mapdata and padding and shows on Rviz simulation. This function was created in the previous lab and implemented in the final. It uses the helper function is_cell_walkable which takes in x and y coordinates in the grid and determines whether the Turtlebot can move to that cell. If the cell is 0 then it is walkable. It also uses the function neigbors_of_8 which looks to see if all eight neighbours of the cell are walkable. If they were not walkable they were added to an array CSpace If a cell was an obstacle then it would check it's eight neighbors' x and y coordinates, and if those coordinates were not already in CSpace they were then added. The cells were then changed from being free to being an obstacle and a new map was created with the new larger obstacles.The final part was having an if statement which added to a new map for each padding amount. A function was created for this final lab where in the event that the Turtlebot got stuck in the cspace while navigating the course find_nearest_valid_pose() within the CSpace node. It uses a breadth first search using the neigbors_of_eight() created in a previous lab to search for a new and walkable cell for the Turtlebot to move to before recalculating a path.

Since the cspace was added, next was finding the new frontiers for the Turtlebot to explore. In the frontier_exploration node a function was created to get the frontier getFrontier() which created bins of frontier points after we segmented and used edge detection on the map. The median of the frontier bins were then calculated and used as the goal for the Turtlebot to navigate to. The distance to the frontier point was calculated using the euclidean distance between the frontier point and the Turtlebot's position. The length of the frontier bin was also calculated to create a priority over frontier distance to length. The shorter distance between

frontier had a higher priority than a frontier with a longer distance. The frontier bin with more frontier points had a higher priority than a frontier bin with fewer points (frontier more narrow). After testing the Turtlebot in the real world it was noticed that the Turtlebot would sense frontier points outside of the map and to fix this problem a new function isFrontierReachable() which uses the service frontier_reachable and it filters out the frontier points that are out of range.

Since the CSpace and the frontiers were calculated the Turtlebot needed to navigate towards the frontier points and explore the map. The navigation node stored the functions that were created in previous labs which made the Turtlebot move. The function go_to() calls functions made in previous labs such as send_speed(), drive(), rotate(), and computingAngleError(). The update_odometry() is also stored in the navigation node which updates the current pose of the Turtlebot.

The global planner node has the A* algorithm which is used to plan a path from the robot to the frontier point. The A* function takes in start and goal arguments of type PoseStamped. We decided to perform all A* calculations with x,y coordinates, therefore, we implemented a function PoseStamped_to_GridCoor that takes a map as an OccupancyGrid and a PoseStamped message and returns a tuple of the x,y coordinates. This was used in the beginning of A* to convert the PoseStamped start and goal to (x, y) start and goal. Similarly, we implemented a Grid_to_PoseStamped function, which converts an x-, y-coordinate to a PoseStamped message. This helper function is used to compute the cost and heuristic of each neighbor-of-eight of the current node, since the functions find_cost and find_heuristic take in PoseStamped messages as arguments.

Another two important helper functions that we implemented were publishFrontier and publishVisited. These functions were used within A* to publish the frontier and visited nodes (or grid cells). These functions are mostly similar, except that publishFrontier takes in a frontier priority queue and publishVisited takes in a visited list. Both functions construct GridCells() messages and set the cells component to the list of elements in frontier and visited. The GridCells message is then published using the corresponding publisher functions, namely frontier_pub and visited_pub.

The last helper function implemented for A* was the CameFrom_to_Path, which creates a list of the path from start to goal from the came_from dictionary. This function takes in the start and goal for the path as well as the came_from dictionary. Firstly, it appends the goal to an empty path list. Then, it traverses through the came_from dictionary until start is appended to the path list. The next step was to optimize the path to remove redundant waypoints; this meant, removing intermediate waypoints in straight motions, whether vertically, horizontally or diagonally. To achieve this, we implemented the function optimize_path, which takes as an argument the full path as a list of tuples (grid coordinates) and returns an optimized path as a list of tuples (grid coordinates). We will now explain how this is done.

Firstly, we created an empty path_optimized list. Then, we appended the first element of the full path to the new list. Then, we traversed through the input path list. If the current index

was the last index of the path, we appended the element, since it was the last element. Otherwise, we appended the element to the optimized_path list if the following conditions were true:

- The x and y values of the current node in the full path were not equal to the previous node, or
- The x and y values of the current node were not equal to the next node,
- and
- The absolute value of the difference of the x and y values between the current node and the previous node were not equal to 1 (diagonal), or
- The absolute value of the difference of the x and y values between the current node and the next node were not equal to 1 (diagonal)

Once the input path list was traversed, the new list, optimized_path, was returned. The Turtlebot used the optimized path to navigate the map and keep finding new frontiers. When there were no more frontiers the Turtlebot switched to the next Phase.

Phase 2:

The map of the maze the Turtlebot navigated through in the first phase was saved to be used later in Phase 3. The Turtlebot used the end goal point as the initial starting position that was saved at the beginning of the previous state. The A* algorithm was used to generate this path and the Turtlebot navigated back towards the start of the map. Once it arrived at the start the Turtlebot switched to Phase 3.

Phase 3:

Phase 3 of the lab involved implementing localization for the Turtlebot using the saved map generated using the gmapping node in Phases 1 and 2. The robot then had to move to a location on the map specified by Rviz after finding out where it was on the map through localization. Unfortunately, the team was unable to complete this phase. The AMCL node was up and working, but the pathing to a point set in Rviz did not work. The team ended up going for partial credit by getting the robot to move to a point using gmapping in simulation. However, since AMCL was running, the process of how that was set up will still be explained.

To set up localization, the ROS AMCL node first had to be configured to work for the Phase 3 guidelines. The required configuration was done in the team's personal copy of the AMCL launch file, where the range of generated particles was changed to 100 - 2000 particles and the minimum distance the robot had to move and turn before updating the particles was changed to 10 centimeters and 0.05 radians, respectively. This launch file was then placed into the team's directory used for Lab 3 in the launch file used for the real-world portion of that lab (the team made one for Lab 3 even though the simulation was all that was needed). The implementation of this launch file was all that was needed to get the AMCL node running on the Turtlebot.

The second issue was having the robot path to a goal specified through setting a 2D Nav Goal in Rviz and move to that path using the AMCL localization algorithms. As the team was

unable to do this, the process to move the robot to a point using gmapping was done instead. In order to achieve this, a new subscriber was set up in the "lab4_client' file that subscribed to the topic "/move_base_simple/goal" and ran a function called "update_goal" whenever it received a message (i.e. a goal was set using Rviz). This function took the PoseStamped message received from Rviz and set a global PoseStamped variable called "self.goal" equal to that message to be used for navigation.

To make the state machine work with this phase, several things were done. The code to execute Phase 3 was placed in a "while" loop in order for the phase to run infinitely. Within this loop, the value of "self.goal" was checked in every iteration. If the goal was equal to anything but a "Nonetype" object (meaning that a goal existed and was set in Rviz), the code would enter the check and execute code to start pathing. The code would run a function to create an optimized path from the current robot's position to the specified end goal, then the code looped through every pose in that optimized path and ran the navigation client on each pose to move the Turtlebot to each waypoint. Once the robot reached the goal waypoint, the code went back to the infinite "while" loop and waited for another goal to be set, repeating the process when there was a new goal.

Results

Throughout testing all three phases, the results were about what was expected. Firstly, Phase 3 was the most predictable one to test, as the pathing did not work in the code and therefore did not work in the real world. The robot would spin around in circles then drive to a seemingly random place either on or outside of the saved map. Along with this, the path that was supposed to show on the computer map did not show at all or did a weird path that did not make sense with the start and goal positions. However, the AMCL localization that was tested seemed to work only when the model of the robot in Rviz was within the bounds of the saved map, as the model would jump around and the scans corresponding to the real-world obstacles would gradually line up with the walls of the simulated map. The robot refused to work in any other circumstance, and it is believed that it was due to the pathing function that the team originally used only working in the bounds of the saved map, so if the model of the robot was outside of those bounds, the robot would not path and an error would show itself. However, the pathing when the model was in the map's bounds also did not work, probably due to an error in one of the pathing functions used.

For Phases 1 and 2, the first tests were wildly inconsistent. Numerous trials were required to determine the correct padding amount to use for calculating the C-Space as well as a good map resolution to procedurally generate the map with. When the resolution was increased, the padding generally had to be increased as well, and vice-versa. There were two main reasons why these values were time-consuming to figure out. The first was due to real-world noise not present in the simulations. In each run, the robot would generally slip a little or act unpredictably if the battery power was low. This led to the path of the robot veering off course sometimes, with the

current run normally ending in an object collision. At the end, for the final demonstration, the values chosen for the padding and the resolution that worked most consistently were 6 cells and 1.6 centimeters/cell, respectively.

The second reason why the padding and resolution had to be tweaked numerous amounts was due to another code-based issue that was not seen until the day before testing. On occasion, the robot would try to find a new path to a frontier, doing this if a new frontier was chosen or if a previous path was closed off by C-Space that generated in its way. Sometimes, the robot would return an error saying that a path was not able to be generated. This error would also occur if the robot found itself in C-Space and could not find a path to get out. The pathing error that occurred when the Turtlebot was outside of the C-Space was fixed by implementing a check into the code. If the path returned a "Nonetype" object (i.e. the path was not found), the state in Phase 1 was set back to where it re-checked the robot's position to see if it was valid. By doing this, the map was able to update without interfering with the robot's pathing as much, and the error stopped showing up for the remainder of testing. To fix the error when it was in C-Space, a function that was handling that code needed to be replaced with another that was supposed to be in its place.

Figure 2: Robot navigating unknown map, detecting goal frontiers and planning path to best frontier.

Figure 3: Robot successfully navigating through Phase 1 of final demo.

Once these issues were fixed and the padding and resolution values were tuned, the robot was able to complete Phase 1 and Phase 2 on the final demonstration day with only a single collision at the end of Phase 2 at the last obstacle. Figure 3 shows the robot successfully completing Phase 1 in the final demonstration. Weirdly, the robot was clipping the obstacle consistently even with the changes in padding and resolution, and the team still is not sure what could have been the cause of the issue. However, besides this and the lack of Phase 3 functionality, the final demonstration went smoothly and as expected. Figure 2 shows an example of the robot navigating through Phase 1 successfully, in simulation.

During the testing phase, one of the most critical functionalities was to be able to detect frontiers. This was done by creating functions to detect cells in frontiers, **edgeDetection()**, and to segment the list of cells in frontiers into groups (or bins) of frontiers, using **segmentFrontiers()**.

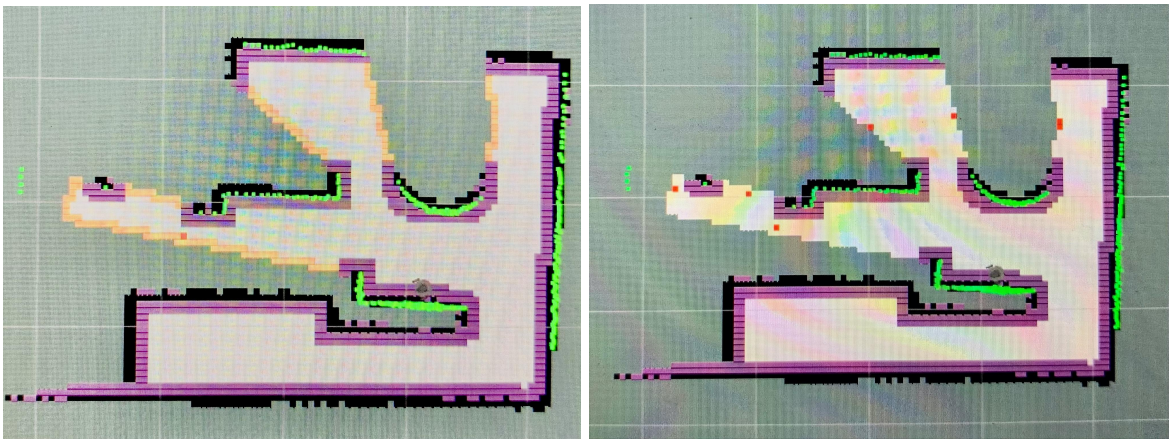

Figure 4: Frontier edge detection (left) and frontier segmentation to goal frontier (right).

To test our edge detection and frontier segmentation functions, we published as GridCells() the full list of frontier cells — cells part of the frontier — and the segmented, goal frontiers, which were initially the centroids of the bins (groups) of frontiers (Figure 4).

Discussion/Conclusion

Our initial design for the final project was in some ways similar to our final solution, but it was also very different. Initially, our design was basic — like a skeletal version of the final solution. We had our main nodes that we had in the final solution, with the flow of information structured similarly. However, the final solution accounted for many more cases that we had not accounted for initially. We will briefly discuss some of these scenarios and considerations we had not factored for. Firstly, the way we were initially calculating our goal frontier was by finding the centroid of the frontier group. Although this was not a bad solution, in some cases, depending on the geometry of the frontier group, the centroid would end up in unknown or occupied areas of

the map. Hence we decided to instead set the goal to the median of the list of frontiers in a frontier group. Also, related to the way we found our frontier groups, our segment frontiers worked mostly well, though sometimes having more than one group frontier for a frontier that could have been further merged. So we implemented a feature to ensure that neighboring frontier groups — frontier bins that had intersecting elements — were merged. This resulted in much less and goal frontiers to choose from and made computation for other sections of the code quicker.

For choosing the goal frontiers, we also had not accounted for the fact that some frontiers were unreachable. An example of this is when our robot is able to see through a narrow slit and identifies a frontier on the other side of the slit, however, due to CSpace padding, or because the point was outside of the map, the frontier was technically unreachable, since no path could be found for it. Therefore, we implemented a service to check if the frontier is reachable or not; if it was, then it was added to the priority queue of frontiers to choose from.

Another key feature we added after a few iterations of testing was to replan a path whenever a previously planned path became invalid. Before, once a robot identified a goal frontier, planned a path to it and began navigation to it, the robot was not able to see that an obstacle (or CSpace) was in the way, and only until it reached a new position, was it able to see the obstacle. However, sometimes the path was already planned and if it went through the invalid space, then the robot would collide. Hence, we had to add a functionality so that the robot checks each time that it reaches a new waypoint whether the full path that was planned was still valid or not. For this, we created a service offered by the CSpace node and was requested by the main client.

Another key functionality that we added was to solve the issue of the robot entering a CSpace and getting stuck in it because it could not plan a path from it or detect new frontiers from it. For this, we implemented a service offered by the CSpace node that checked if the current pose of the robot, sent through the service request, was in the CSpace or not; the service was requested by the lab4_client node. If it was, the service responded with a pose, which was the closest unoccupied and known position in the map. To find the nearest valid pose, we made use of a breadth-first search algorithm. Once the new valid pose was sent back, the robot would navigate to that valid position, and resume main tasks.

In conclusion, we were able to achieve most of the main goals of this lab. Through gmapping and amcl, we were able to localize the robot in the world, and we were able to drive around the map while avoiding obstacles using gmapping. Through our frontier exploration node, we were able to find new areas to explore. Once we found the goal area to explore, we were able to find an optimal path to that point through our global planner; this same node also allowed us to find any optimal path to any point. Finally, our robot was able to identify when the map was complete and navigate back to the starting position on the map. In this lab, we further learned key concepts of ROS, mainly the creation and implementation of custom Services, since our solution heavily relied on Services. We also learned different ways in which nodes can communicate with each other. Most importantly and generally, we were able to design and implement a full ROS system that can perform SLAM, resulting in a mostly successful project.

Appendix

Link to tag repo: https://github.com/RBE300X-Lab/RBE3002-07/releases/tag/Lab4.final