RBE 3002: Autonomous Path Planning

1st Gandhi, Arjun
Worcester Polytechnic Institute
Robotics Engineering
Worcester, MA, USA
agandhi@wpi.edu

2nd Gere, Nathan
Worcester Polytechnic Institute
Robotics Engineering)
Worcester, MA, USA
njgere@wpi.edu

3rd Prem Sankar, Prathamesh
Worcester Polytechnic Institute
Robotics Engineering
Worcester, MA, USA
ppremsankar@wpi.edu

***Abstract*—This lab combines all of knowledge from previous RBE 3002 labs and add aspects of navigation and sensing in order to explore and map an unknown area to be able to navigate in an efficient manner from point to point.**

***Index Terms*—WPI RBE 3002, SLAM, TurtleBot**

I. Introduction

The objectives of this lab are numerous. The first is to localize the robot within the world. The robot must also drive around the map while avoiding obstacles. Additionally the robot must be capable of finding interesting areas (frontiers) to expand the map, and determine the optimal path to each point of interest. The robot must be able to identify when the map has been completely explored, and navigate back through the map to a given point.

The assessment to gauge the robot's fulfilment of these objectives will consist of 3 phases. In Phase 1 the robot will create a map of the environment using G-mapping. It will calculate the C-Space of the map, find a frontier to explore, plan the optimal path to said frontier, navigate to said frontier, and repeat this process until all frontiers have been exhausted. The map will then be saved on file. In Phase 2, the robot will use the map to navigate back from its current position to the starting position while avoiding obstacles. In Phase 3, the robot will navigate to a specific goal in the maze, which has been chosen by a malevolent professor.

II. Methodology

During Laboratory 2, the go_to() method was programmed by calculating the differences in positions between the current and target pose by using the math.hypot() function. Then the angle between both was calculated using math.atan2() function. Finally we used the result by plugging it into the drive() and rotate() functions after path planning and rotate() is called again after reaching the target to match the end pose direction.

During Laboratory 3, the A* algorithm was implemented for path planning in Rviz and creates a sequence of waypoints that allows the Turtlebot to get from its initial position to the target pose. This was also the first implementation of a ROS service to communicate with the tf node and path planner.

During Laboratory 4, we prepared for the PDR by drawing diagrams of our coding infrastructure to explain how we wanted to code this lab. We also explained our different nodes and what they are communicating to each other. From there although the plan was to space out writing the functions due to various sicknesses and inability to meet together we were not able to beginning programming until Wednesday afternoon. However, we were still able to complete all the objectives that were necessary for this lab.

III. Results

We were able to successfully complete all sections of the lab. To do so, we needed to write four nodes and three classes to aid us in the completion of the lab. The node layout of our code is as follows.

The first class we wrote is a Radian Class. The Radian class serves as a way to perform normalization on the angles that are used by the move.py node. The Radian class simply overwrites most of the basic python math functions and internally normalizes the angles between -pi and pi space.

The move node is responsible for moving the robot linearly to any point specified. The move node uses the tf node to know the position of the robot. It is subscribed to the /path_exec/go topic, which is a pose stamped message. The node calculates a direct drive path that rotates to the original heading and drives the appropriate distance to reach the target. The move node outputs the command velocities to the /cmd_vel topic in a Twist message. The node also publishes a Boolean message on the /path_exec/reached topic after it reaches its target point.

The next node needed was the navigation node. The nav node serves as a path manager node. The nav node receives a desired goal on the /move_base_simple/goal topic. Once received, the nav node requests the /plan_path service and uses the tf node to determine the current position of the robot and requests a path from the current orientation to the target goal. Once the path is received for each pose listed in the path, the node publishes a method on /path_exec/go and then waits for a message on /path_exec/reached before publishing the next message. Once the execution of its path is complete, it publishes a Boolean message on /explorer/reached.

After looking at our path_planning service in Lab 3 we determined it would be best to write a map class that would have the responsibility of containing generic transformations that could be applied to the map. We also realized that converting the given 1d occupancy grid into a 2d NumPy array would allow us to perform some of the more useful matrix operations to our map. So the Map class takes in an

Occupancy Grid as a message and returns a new Map object. A map object has several useful functions:

- A map object can be converted into a grid cells message. This is useful for outputting various things such as our cspace, as well as our explored nodes into rviz.
- A map object can calculate a cspace and return a new cspace map with a varying border specified by an input. There is also an additional parameter that allows the cspace function to treat unknown space as a wall.
- A map object can use its metadata to convert an (x, y) coordinate to world coordinates and vice-versa.
- A map object can return a list of neighbors around a given (x, y) point. Neighbors are defined as the (x, y) points in a square matrix with the center being the desired point and the input distance “d” being the radius of said matrix. The neighbors method also has 2 optional exclusion thresholds which are a min or max to ignore certain neighbors whose values do not fall between the 2 thresholds.
- A map object can display itself as an image.
- A map object can run erosion and dilation upon itself.
- A map object can isolate frontiers by finding the borders of 0 and -1 and setting that value to 100 while setting the rest of the points in the map to zero.
- The map can run Canny edge detection on its self with an option to blur the map using Gaussian blur beforehand.
- The map can return a list of edges found in the map. This runs contour detection on the image and then returns a list of Edge objects with centroids and size calculated.

The map class was used heavily in both the path_planning and explorer node.

The edge class is a small class written to make storing edges (frontiers) of a map easier and debugging. The edge class holds the centroid as an (x,y) tuple and the edge size as a float. It also contains a print method that allows it to be printed on the terminal for the debugging process.

The path planner node uses A* to find a viable path between the two specified poses. The path planner node relies heavily on the map class. It uses the neighbor function of the map class to perform a weighted breadth-first search from the start position to the end position. Before running A* and generating a path, it uses the cspace function to inflate the borders of the map, as well as uses the walkable neighbor function to find the closest point to the target and start that is accessible by the robot. A* has 3 weights that are added together:

- A euclidean distance from the start to the current node.
- A euclidean distance from the current node to the end.
- A euclidean distance from the current node to its previous node.

The explorer node is responsible for identifying and selecting which frontier the robot travels to. The node first inflates the map using cspace to prevent the identification of small frontiers, which are likely to be insignificant. The node then runs the isolate frontier function to show each identified edge. The node runs erosion and dilation to clean up any frontiers. The robot then checks to see if there are any identifiable frontiers in the map. If there are not, it exits the loop and finishes exploring. If there are, it turns the found frontiers into edge objects using the to_edge method of maps. It then sorts each frontier from biggest to smallest, publishes a message on the /move_base_simple topic, and waits for a message on /explorer/reached. After it finishes exploring, the robot publishes a Boolean to the /explorer/state topic.

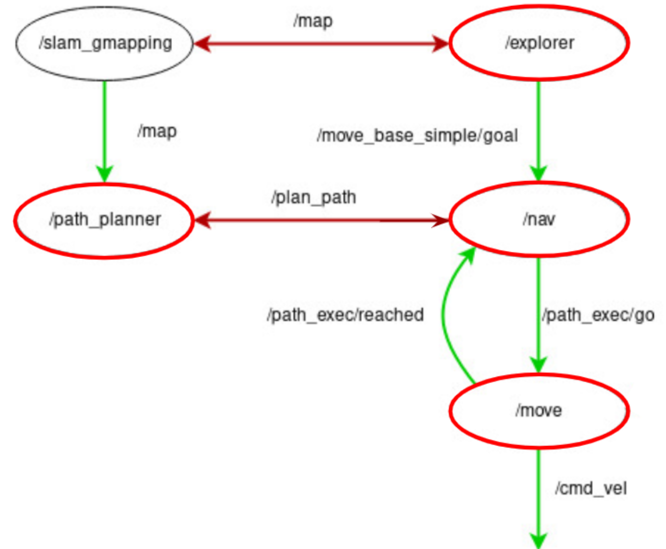

Figure 1: Map of ROS node communications

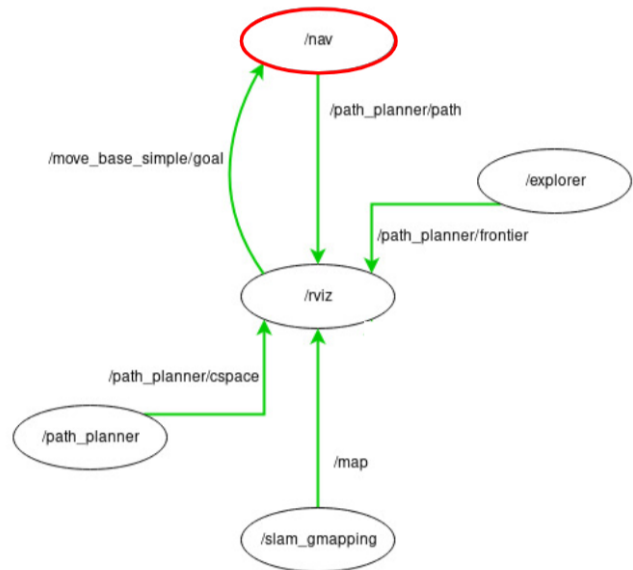

Figure 2: Map of ROS simulation node communications

IV. Discussion

Once again, the goals of the lab were many. The first was to localize the robot within the world. The robot must also drive around the map while avoiding obstacles, and must be capable of finding interesting areas (frontiers) to expand the map. It must be able to determine the optimal path to each

area of interest. The robot must be able to identify when the map has been completely explored, and navigate back through the map to a given point.

In general, these goals were all achieved by our team. The robot was able to localize effectively within a novel map. In operation, it rarely made contact with the walls of the environment. We have theorized that the cause of these occasional collisions is that the robot is programmed to drive the full path to a desired frontier, but during the execution of that path, it does not update the cspace edge buffers to reflect new measurements. This causes the robot to drive too closely to the previously hidden wall, potentially causing a collision. A possible remedy to this issue could be to have the robot update the cspace wall buffers more frequently to accurately reflect the information it receives during its drive to a location.

The robot was able to determine a relatively optimal path to each area of interest. However, a flaw in the programming of our A* algorithm may have caused some suboptimal pathing. Rather than incorporate both the euclidean and Manhattan distances into the heuristic [1], we opted to neglect incorporating the latter, which may have impacted the efficiency of our algorithm. However, this issue was minimized due to the fact that we filtered out untraversible paths later on in our path-planning code. In the end, our paths still reflected essentially the same optimality as with A*, but the implementation in the code could have been more elegant.

An additional boost in navigational accuracy and speed could have been obtained by incorporating acceleration curves into our driving and turning motions. We chose not to incorporate this feature for the sake of time, but it was clear from the real-world operation of the robot that our driving was imprecise and not as fast as it could have been. While our time to navigate through the map in Phase 3 was acceptable, at just over 1 minute, the fastest group was able to perform the same task in under 30 seconds. This means that there was ample room for improvement to our robot's driving performance.

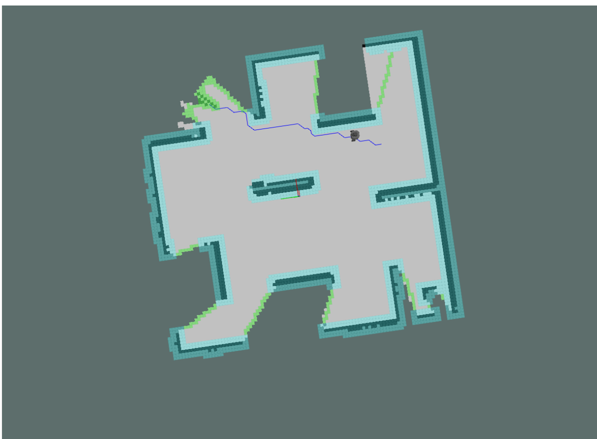

Figure 3: Navigating to a frontier centroid

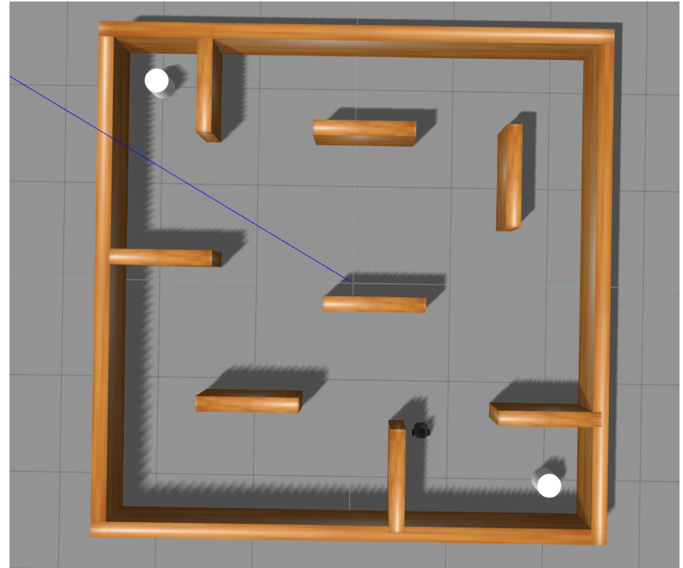

Figure 4: Robot's proximity to the walls

Conclusion

In conclusion, during Laboratory 4, we were able to successfully localize the robot using G-mapping and some AMCL, find frontiers using a size hungry heuristic, drive through the map using the A* algorithm in our path planner node, and navigate through the mapped area with an optimized path that skips waypoints in straight lines. We began by trying to create and save a 2D occupancy grid and was able to achieve on the 2nd try during our demo. Next we had to start exploring our frontiers and we were able to explore it completely on our 3rd try due issues with moving too close to C-space and possibly low Wifi-speed. Next the goal was to navigate back to the start position using this created map in rviz which worked during the demo on our 1st attempt. Lastly we had to 'teleport' the robot to a random position in the map and reach the specified position, which we were able to accomplish in 1 minute and 3 seconds.

Altogether through mapping and navigation Team 15 was able to use their skills and programs from previous labs to accomplish autonomous path planning. Team 15's completion of these exercises demonstrates their knowledge and capability in all laboratory 4 objectives. This has been a great learning experience and the team is looking forward to exploring new applications with this knowledge.

References

[1] A. Patel, *Introduction to the A* Algorithm*, May 26 2014. Accessed on Dec. 13, 2019. [Webpage]. Available: www.redblobgames.com/pathfinding/a-star/introduction.html

V. Authorship

Section	Author
Introduction	Nathan Gere
Methodology	Prat Prem Sankar
Results	Arjun Gandhi, Nathan Gere
Discussion	Nathan Gere
Conclusion	Prat Prem Sankar