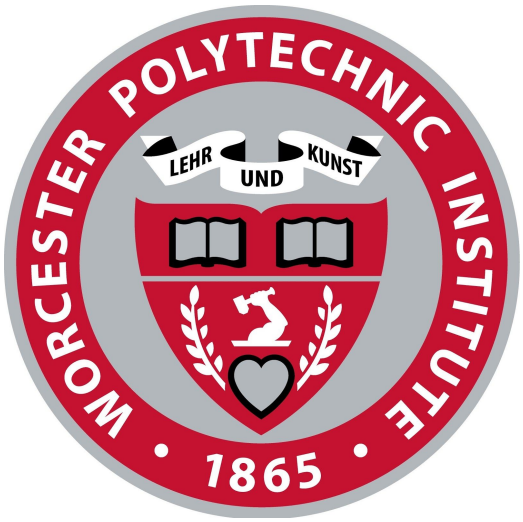

Worcester Polytechnic Institute

RBE3002 Team 3:

Final Project Report

Dembski, Clayton John
van Rossum, Floris
Xie, Harry

supervised by:

Professor: Carlo Pinciroli

Abstract

Using Python in conjunction with the node system and mapping applications provided through ROS, the goal was to connect to, and work with the Turtlebot 3 in order to create an occupancy map of an unknown location, then pathfind back to the known home. Map creation was accomplished by creating a grid system of occupancy data, identifying and connecting valid movement positions, and navigating to sets of frontiers with the Navigation Stack. The homing of the robot was formed via A* methodologies with a series of modified heuristics to further avoid obstacles.

For those who have access, all of the code created for this project can be found here: https://github.com/RBE300X-Lab/RBE3002_D18_Team_03

Table of Contents

Introduction

This lab had a little lamb

Project Goals

Control and program a Turtlebot3 so that it can explore and navigate a maze, find the lowest cost path from the start to the goal, generate waypoints and then travel from the start to the goal autonomously.

Methodology

The following section covers the major steps that were followed in order to complete the assignment.

1. A significant amount of configuration and software is required in order to allow for the execution of the project. By following the tutorials listed in References [1] and [2], an adequate knowledge can be obtained while achieving the proper configuration on your personal computer. A linux distribution is required in order to correctly run Robot Operating System. For this project ROS Kinetic was used. This methodology section assumes all previous lab assignments have been completed correctly.

Connecting to the Turtlebot3

2. The guide posted on canvas was followed. References: [3]
3. A quick summary of the guide. Set the ROS_MASTER_URI variable to the computers IP address along with 11131 as the port address. In order to find the computer IP address run the command “ifconfig”. This will print out all the needed network information. The ROS_HOSTNAME, should be set to the IP address of the computer. TURTLEBOT3_MODEL should be set to “burger”.
4. Now you can connect to the turtlebot3 with the command “ssh burger@turtle-NUMBER.dyn.wpi.edu” where NUMBER corresponds to the number on the turtlebot.
5. Once connected to the turtlebot, ROS_MASTER_URI should be set to the same as the variable above.
6. Now RVIZ can be launched on your computer and after, the ros bringup command can be run on the turtlebot.

Frontiers

7. In order to travel to unexplored regions of the map, frontiers must be found and navigated to. This is a lengthy process with multiple different steps.
8. With the map generated by the previous code, frontier cells can be identified. A frontier cell can be identified as a cell that is unoccupied, explored and adjacent to an unexplored cell. The entire current map has to be looped through in order to create an array of frontier cells that meet this criteria.
9. Once the frontier cells are obtained and in one array, they must be grouped together in clusters based on their distance apart. A distance of less than 1 or 2 will do in this case. That means if two frontier cells have a euclidean distance of more than two it will cause them to be considered as two seperate frontier clusters. Once all the frontier clusters are identified they can be grouped into one single array. So, this operation should return an array of arrays of frontier cells.
10. Frontier clusters centroids have to be found next. In order to navigate to the frontiers properly, centroids of the frontier clusters have to be found. This can be achieved by adding up all the x and y locations of each of the individual frontier clusters and then dividing them by the total of the x and y. This will produce an average location of the x and y. Next, a waypoint has to be found that corresponds to this location or near to it, this waypoint should be unoccupied. All these centroids should be added to an array along with the length of their frontier.
11. These frontier centroids and lengths need to be prioritized next. This means that a longer, closer frontier centroid is more preferable than a farther, shorter frontier. This can be done by performing a linear combination of the distance to the centroid plus the length. Then two constants in front of these variables can tune it to your own application. This prioritization function should return one frontier waypoint which is the best to travel to next.

Nav Stack Driving

12. In order to reduce the chance of error and simplify driving, the ROS nav stack library can be used to drive from one place to another while exploring.
13. A nav stack driving method should be written or used which allows you to send a waypoint to the nav stack, which will then command the robot to go there.
14. This nav stack driving should be integrated with the frontier code. Once the ROS python node is started in an unknown environment it should keep running until the entire map is explored. This means a while loop should be written that keeps running until the map no longer updates, all frontier waypoints have been reached or are out of reach. This loop should continually call the frontier prioritization method in order to obtain the most recent frontier waypoint to travel to. As the robot travels to the next frontier centroid it will observe new terrain and create new frontier centroids until the whole map is explored.

A* Path Planning and Navigating

15. Once the entire map is explored, it must be saved to a .yaml file with the command:

```
rosrun map_server map_saver -f <your map name>
```

    This will save the map in the package folder. This map can then be loaded into rviz and the amcl localization can be started with the command:

```
roslaunch rbe3002_d2018_final_gazebo final_run.launch
    map_file:=absolute_path_to_map_yaml_file
```

    Assuming that the rbe3002_d2018_final_gazebo package has been downloaded from canvas. This is a localizing node that will localize the robot given a pose estimate.

16. Once RVIZ has been launched correctly, a pose estimate should be given with the tool in RVIZ, this pose estimate should reflect the actual location of the robot within the map.
17. The driving node should also be started now and waiting for a target Pose input by the user via RVIZ. This driving node, which uses A* and the driving code specified previously, should come from the previous labs.
18. A target pose can now be sent to RVIZ which will use the currently loaded map to send a waypoint to the A* node. This node will then return a path.
19. This path needs to be adjusted first before it can be sent to the driving code. In order to properly avoid walls and remove any unnecessary instructions the path must be modified. First, each path waypoint should be checked for adjacent occupied waypoints, if one is found the path waypoint should be adjusted in the opposite direction. This results in a “safer” path. An optional change that can be made is to adjust the heuristics of A* to avoid travelling next to walls. For the gCost calculation, add (4 - number of neighbors). This means that A* will prioritize the nodes that are not next to a wall, thus creating an even safer path.
20. Aside from making the path safer, the path should be checked for any unnecessary waypoints. For example, if the robot is already travelling in a certain direction and the next waypoint features the same orientation and is along the current direction, then that waypoint is not necessary and can be removed. This can be done by checking the orientation of each waypoint and removing it if it is the same and along the path of the current orientation.
21. Once the path modifications are completed the path can be sent to the driving code which should send the robot along it correctly.

Results

Project Goals

Overall, the above defined project goals were completed. The turtlebot3 successfully explored a previously unexplored map and then navigated around it using our own A* algorithm and driving code. There were numerous problems and challenges that occurred along the way toward the project completion that will be discussed and reported here.

Map Acquisition and Exploration

The 4 testing maps, along with the final map that was created for the robot to traverse, with the specifications of 40 cm minimum corridors, and walls taller than the robot, were eventually successfully mapped without the robot running into walls or corridors. The mapping was performed automatically via a Gmapping and Nav Stack launch file provided, however

identifying the positions to map to, to create the full map was left as a user task.
The driving problem was difficult to solve. Although it seemed like a simple challenge many problems were encountered. The driving code would be given a Pose as a target position and orientation and navigate to it. If the robot drifted too much, as described above, it would cause the robot to miss the threshold and sometimes keep driving past the target. Although this was fixed by driving to a goal in half distance increments, it was a problem. Transforms also resulted in significant problems. Whenever coordinate transforms had to be done from the world frame to the robot frame a few lines of could had to be run that were given by the lab instructors. Therefore, the knowledge on them was limited. Still, a significant number of errors were encountered when using transforms. The most significant one was an “Extrapolation Exception” caused by the tfLookup method. This unexplained error was very mysterious but was fixed by adding rospy.wait() function calls before or after the tfLookup() function. This problem was encountered numerous times, usually when connecting to the actual turtlebot. The robot itself moved through a list of poses specified by A*, approaching the same node Zeno’s Paradox style until within a specified range of the node.

Out of the trials, one in roughly 10 in small spaces would create occupied locations that would block the robot from continuing through the corridor. Using the heuristic to displace one gridspace from the wall, a distance threshold of .193 and a speed of .05, and assuming the robot had properly calibrated its initial position, the robot could successfully pathfound and navigated around wide open, 90 degree walls without failure. Smaller corridors, and acute walls inflected inwards towards the robot had a mixed success of ~50% due to drift in the robot and the robot meeting waypoint thresholds too early.

Discussion

Grids

To implement A*, a graph of connected nodes had to be implemented. Instead of interpreting the map data as a direct grid, we wanted to make the system more robust: Each waypoint contains a list of waypoints it is connected to. In this way, if we wish to reinterpret the path information later, and connect waypoints in a larger scale, we may. To create the graph, first we subscribed to the /map topic, which gave us a 1 dimensional array containing all the points in the grid. To find a particular row and column info in the grid, the once can use the index

```
self._currmap.info.width*r+c
```

And can scroll through the grid as if it were two dimensional. To find the x and y coordinates of each point in the grid, to deliver to the way point, two transformations must be employed. First, one must account for the fact that the occupancy grid has an offset from the world origin

```
Self._currmap.info.origin
```

Then, using the resolution of each gridspace, they must find the total distance to the current column for the x and row for the y

```
c*self._currmap.info.resolution
r*self._currmap.info.resolution
```

Once each of the waypoints were created, the next step was to connect them with one another. As the info delivered to us was a grid, if a waypoint was unoccupied, it was connected each unoccupied waypoint in the 4-connected system. As the array of waypoints was also a 1 dimensional grid, to find out if there is an item in the previous column in a 2d grid, one should use:

i % self._currmap.info.width -1

In the next column

```
i % self._currmap.info.width + 1
```

In the previous row

```
i - self._currmap.info.width >= 0
```

And in the next row

```
i + self._currmap.info.width <
self._currmap.info.height*self._currmap.info.width
```

For the simple_map graph, the result of the connected waypoints is as follows, where the symbol '_' is the robot position on the map and ' is the current goal. Each number indicates how many waypoints each node has

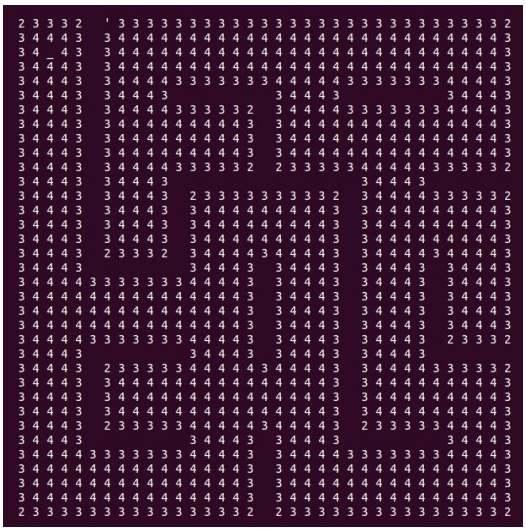

Figure 1: Waypoint connection visualization

The size that we chose for each item on the occupancy grid is slightly larger than the width of the greatest side of the turtlebot 3, at .2m. It is important to note, that, in Rviz, we are using the old turtlebot model, not the hamburger model, and therefore the robot looks larger than it performs to be. This occupancy grid size is set as a parameter when the object is initialized as the variable `_robotSize`. The occupied grid is calculated by first, finding the size of each grid the base occupancy map given when loading in the data. Next, it divides the robot size by the size of each grid to determine an integer value for how large each cell must be to meet the minimum grid cell size needed to be large enough for the robot. It then checks each occupancy cell of the base map in each space in the new c space grid, and if any are occupied, the particular cell is considered occupied. An example is as follows:
A robot has the size of 2 grid cells. The occupancy grid sent by /map is as follows

100	0	0	0
0	0	0	0
0	0	0	0
0	0	100	0

100	0
0	100

As the robot is 2 units long and wide, each c space grid is a 2X2 occupancy grid

1 2

3 4

As robot gridspace 1, and 4 have occupied cells, as seen by occ(0,0) and occ(2,3) respectively, 1 and 4 are occupied. The grid therefore becomes

OCC FREE

FREE OCC

An example of this expansion can be seen below

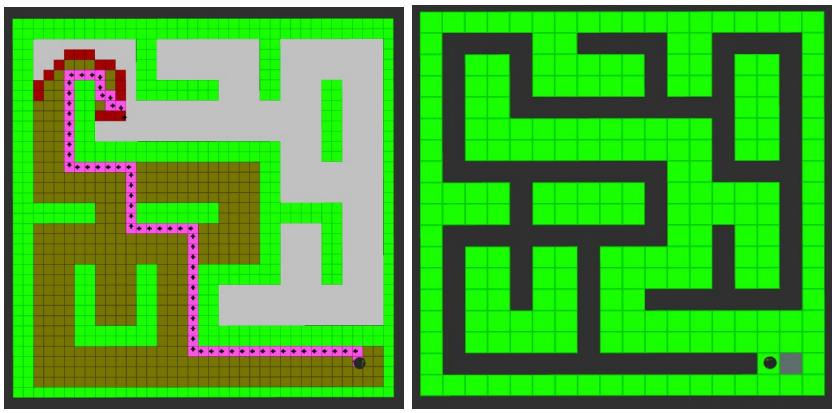

Figure 2: C Space Expansion Via SuperSampling

Each waypoint also contained the information _occ, a value from 0 to 100 or -1. This occupancy data was updated at every tick of the /map occupancy grid. If the incoming data has an occupancy of above 70, all neighbors of the waypoint are disconnected from the given waypoint, allowing the new barrier to be formed, and stopping the robot from pathing through the location.

Frontier

To navigate to each of the frontiers, a series of calculations were performed: Identifying the frontier areas, grouping the frontiers, finding the centroids of each group, and mapping to the

centroids position. To identify the frontiers, each unoccupied known waypoint was identified. If any were adjacent to an unknown value, the location would be identified as a frontier and would be added to a list of frontiers. Once every possible frontier location was identified, they were grouped by a threshold. If a frontier was within a measured range of another waypoint those two would be grouped together and the second waypoint would would check for any ungrouped waypoints in its range, and so on. The centroids of these frontier clusters were then found using the process described in the lecture. The best frontier was then found based on the frontier centroid distance from the robot and the length of the frontier cluster.

```
    distanceToFrontier = centerFrontierValue.calculateMDistance(self._robot)
                        lengthOfFrontier = len(frontier)
  Priority = (distanceTune*distanceToFrontier) + (lengthTune*lengthOfFrontier)
```

This position would then be snapped to the closest known, unoccupied neighbor to insure that the robot would not try to path to somewhere unreachable. The position would be sent to the nav stack so that the robot could travel to the location. The program repeatedly pulled the highest prioritized frontier centroid from the array until there were no more left, meaning the entire map was explored.

A* and Pathing

Implementing the A* algorithm was a lot of fun. Especially having RVIZ available to visualize the process was extremely helpful. In our case, we chose to draw the closed set and the frontier of A*. This provided great insight as to how A* works, and if it didn't work, why it didn't. Our A* algorithm acted as expected, a waypoint destination and start as well as the map was sent to the A* function and a path was returned. Because of the availability of the lecture notes and lab documents, writing A* was not significantly difficult. To better understand the functionality of A*, we were able to change the heuristics and cost algorithms in order to produce different results. For example, we added more preference to x or y in the heuristic in manhattan distance. This meant that the path hugged the wall on the x or y side, depending which axis was weighted. We also managed to create a greedy best first search be weighing the end waypoint euclidean distance heuristics over the cost of each waypoint. This resulted in pathfinding that was significantly faster than the A* counterpart, however, paths ended up with much greater twists and turns. The final heuristic for a was one that forced the robot to avoid walls whenever possible. We realized, when running amcl on the robot alone, our robot visualization would slowly diverge from the actual physical position of the robot. This would cause problems because, in most cases, the most efficient route would take the robot directly adjacent to the wall. The asyncrity of the robot would thereby cause it to run into said wall. To solve this, two pieces were implemented. First, the heuristic was modified to give a higher cost based on how many occupied waypoints were connected to an unoccupied space. In this way, waypoints adjacent to walls costed more, and were given a wide berth. Second, seen below, if the robot did have to path next to a wall, and the wall was identified in the 4 connected set, the point would be moved as far from the edge as possible. This meant that walls in the 8 connected set, however, would not be identified. Due to this, the path would display in a spline curve around the wall, smoothing movement.

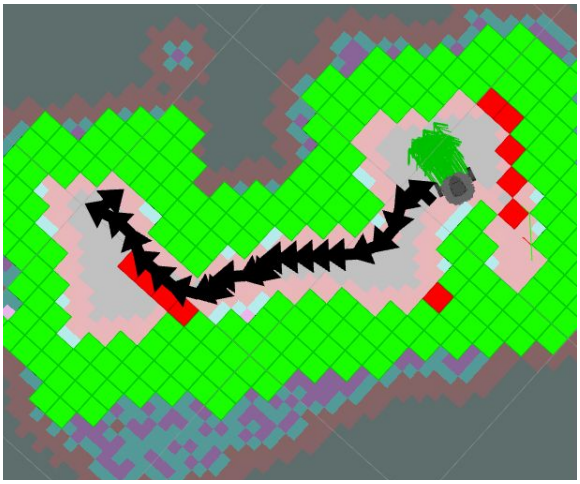

Figure 3: Waypoint Positions Placed on Edge Furthest from Wall

Conclusion

While the robot was able to successfully complete its initial task of mapping, the inconsistencies of localization and thresholds of driving the robot ultimately lead to a pathing system that was unreliable and error prone. Given more time and a deeper understanding of how to read and interpret the data returned by the Lidar, we would expect to create a more dynamic movement function. This function would react, not by recreating the waypoints that the robot would move to, but by pushing the robot back away from the wall based on how close it was. A pid system could be used to push the robot further from the wall and avoid collisions. While, overall, we are satisfied with how the robot could perform, we were ultimately disappointed with our final presentation of the robot, as an added accidental mark in the pathing code caused it to malperform. We did, however, feel we learned a great deal through these labs. Directly, no one in the team had prior experience with ros, and we felt that we have gained much confidence in the node topic system used by messages and services. Indirectly, we gained much experience in graph traversal and .xml, and in setting up larger file and application structures rather than just working with code structures. We found the project overall less stressful and more fun than any of the previous robotics courses

References

1. http://wiki.ros.org/ROS/Tutorials - ROS Tutorials
2. http://wiki.ros.org/turtlebot/Tutorials - ROS Turtlebot Tutorials
3. https://canvas.wpi.edu/courses/6998/files/folder/labs/lab2?preview=1159162 - Turtlebot Connection Guide, Canvas

4. https://answers.ros.org/question/188023/tf-lookup-would-require-extrapolation-into-the-past/ - TF Lookup Extrapolation Exception