Questions

Who did what in your group?

Toby and Suraj worked together on the maze.py file to implement Dijkstra's algorithm to find a solution to the maze. Lonnie also helped debug this code. Suraj and Lonnie worked in the solution.py file to make the robot execute a list of actions from the path of the solved maze, making tweaks to both the code and the physical maze in the lab to make the robot operate smoothly. Suraj and Lonnie also implemented the extra "going further" functionality which solves the maze using a different technique that we came up with on our own.

What happens when the robot receives a laser scan message? Does the robot take any immediate action?

scan_callback() is called every time the robot receives a laser scan message from the subscriber. scan_callback() uses the transform from the camera depth to the base frame to perform a point conversion into the correct reference frame. The array of points corresponds to the array of angles.

At each control cycle during the straighten behavior, the controller examines the 2D locations of the endpoints of the "laser beams" 5◦ off center. How is the displacement vector between these two points (expressed in the robot's own body frame) used to estimate an angular error?

Delta is the displacement vector between the two endpoints of the laser. We find the angular error by measuring the angle subtending this vector.

Why does the lookup_angles method need to check for NaN values? How does it try to compensate if the desired angle doesn't have a valid range associated with it?

The lookup_angles() method needs to check for NaN values in case there are values where the robot does not see an object. If there is no valid range within the cutoff angular distance, the lookup_angles() function returns None.

What happens if you try running the straightening behavior when the robot is facing directly into a corner? What happens if you try running it when the robot is too close (just a few inches or so) to the wall? Why does the robot do what it does in these cases?

When the robot is facing directly into a corner, the program prints "points was None in straighten state!" When the robot is too close to the wall, the program prints the same message. This means the program did not find points at plus and minus 5 degrees.

How did you know what code to add to the maze.py module to test your Dijkstra's algorithm? Did writing the tests reveal anything about how your code should work?

Using the test mazes we could print solutions firstly as coordinate-locations of cells within the maze. We had to add the pdb python debugger in order to follow the data and realized that for Dijkstra's algorithm we needed to implement a priority queue to figure out which of the points in the queue had the minimum cost, setting the cost as the priority so that the lower ones would get pushed up first.

At this point in the semester, we are using a large number of software tools, including git, ssh, ROS command line tools, ROS GUI tools (possibly, for visualizing things), text editors, and others I'm sure I've left out. Give an example of a time when a tool wasn't working the 4 way you wanted it to – how did you resolve the problem? What resources did you consult? Have you learned anything in general this semester about getting the hang of new tools?

One method that I had not used before this class was writing code in vim, so that I could write code for the lab laptop while working on my personal macbook. While working on project 3, the robot slalom course, we had several member functions that simply were not getting compiled (they didn't exist) and we simply could not figure out they weren't being recognized on the program runs. By consulting with Professor Phillips, we eventually figured out that the code wasn't compiling correctly because of issues using spaces versus tabs in our code because the program had multiple authors with different coding habits. Professor Phillips helped us use tools like expand tabs to retab our indentations to set counts of spaces. Coding in vim was a pain at first but the best way to get the hang of a new tool is by using it.

Provide a short (2-3 paragraph) description of the extra functionality you added.

We decided to solve the maze our own way by coming up with our own algorithm instead of Dijkstra's. Our method could be described as "picking the closest neighbor in mostly the right direction." It is similar to the A* algorithm, but we simply used the euclidean distance from the neighbors of the current location to the end location, and chose the neighbor that has the minimum distance to be in our path.

There are cases where the algorithm would simply fail, such as points in the maze where picking the cell with the smallest Euclidean distance leads to a dead end. But for special cases of maze setups where it does work, this is a lot less computationally expensive than a complete dijkstra's algorithm. This is because we never explore neighbors that are further away from the end point than the closest neighbor to the endpoint.

Include a link to a video or videos on Google drive shared with me which demonstrates successful maze traversal on a series of start/goal pairs. The pairs should provide an interesting challenge to the robot.

https://docs.google.com/document/d/1KlKcLJ2ta0cYLhZ0yrAsf5Ou-FXFIr8hQ2f_KihayMw/edit?usp=sharing

dijkstras1.MOV shows the maze solved using Dijkstra's algorithm. dijkstras2.MOV shows a different set of starting and ending points. extra.MOV shows a run where we used the same set of points as in dijkstras2.MOV, but our technique managed to find the same solution in a simpler way.