# CSE 180 Introduction to Robotics

Searching for pairs of poles given a map

Carlos Diaz
Manuel Meraz
Jesus Sergio Gonzalez Castellon

## I. Pole Detection

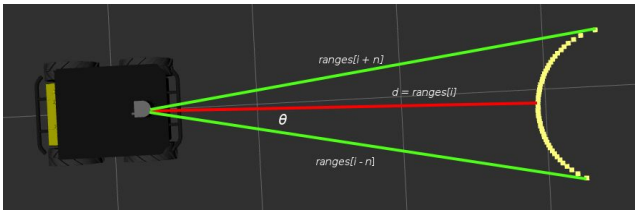
### *A. LiDAR Pole Detection*

Our approach to detecting pairs of poles using LiDAR data was to split our detection algorithm into three main tasks. First was parsing through the LiDAR data for any data satisfying our parameters telling us a pole is being detected. Second was transforming the location of the detected pole into coordinates based on the current global frame, or the map frame. Third was checking if any pair of poles met the criteria of being 1m (+-0.1m).

### *B. Single pole detection*

To detect a single pole, we assume every index of the LiDAR data is pointing to the center of a pole. We then, through numerous tests, try to prove the current sensor reading is not a pole. To do this, we use a 'for' loop that iterates over all ranges returned by the LiDAR.

```
for(int i = 1; i < laser_scan.ranges.size() - 1; i++)
     //Determine if ranges[i] hits the center
```




Assuming the laser_scan.ranges[i] is the center of the pole (line in red on above diagram), we calculate the theoretical distance and array index for the readings in laser_scan.ranges containing the edges of the pole.

Calculating theoretical laser_scan range readings based the following characteristics of a pole:

1. We can calculate what the distances are to the edge of the pole based on the suspected center distance to the pole, and the radius of the pole.

$\theta = atan2(\frac{r}{d+r})$

$\alpha = laser\_scan.angle\_increment$

$n = \lfloor \frac{\theta}{\alpha} \rfloor$

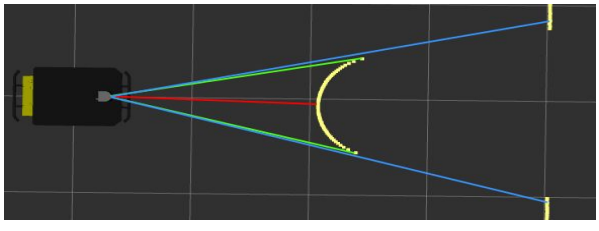
```
If (
  0.9 < pole_edge_distance_theo / ranges[i+n] <
  1.1 && 0.9 < pole_edge_distance_theo /
  ranges[i-n] < 1.1
)
     //Pole center detected at
     laser_scan.ranges[i];
```

2. All points to the right and left of a pole will have increasing values when read from one point to the next.

```
//Code for right of center of pole
 for(int j = 1; j<n; j++)
   if(ranges[i+j > ranges[i+j-1])
     //Pole not detected
```

3. Any readings beyond the pole (blue) should extend beyond the edge of the pole. In other words, the pole should not be larger than the provided diameter.

```
if(0.9<theoretical_hypotenuse/ranges[i+n]     <
1.1)
     //Confirm pole detected
else
     //pole not detected
```



### *C. Poses of Poles using a Point Cloud*

Once we were able to detect poles in the ranges provided by the laser_scan, we needed to determine the

position of the poles in relation to the map frame. To do this we found a tutorial that gave us the code to transform the laser scan ranges data to a point cloud [1]. Which we could use to get an (x,y) coordinate for every laser scan range.

D. Detecting Pairs of Poles

To detect pairs of poles, we used the (x,y) coordinates returned by the Point Cloud to match up the detected pole indexes to their (x,y) coordinate. We then found the straight line distance between the two points. If the distance between them was within a 20% threshold of 1m (0.8m to 1.2m), then we have a pair of poles. We also determined that there was potentially a pair of poles if the distances between two poles was within a 40% threshold of 1m (0.6m to 1.4m). This helped with long range detections, where the Point Cloud transformations were not too accurate.

*E. False Positives*

In our testing we occasionally received false positives for poles detected with cylinder shaped objects such as Fire Hydrants. Our detection of false positives was low enough to ignore since we determined the condition for pairs of poles would filter out any anomalies for single pole false positives. We made the assumption that the likelihood of Fire Hydrants placed at 1m apart was low, and that if there were Fire Hydrants placed at 1m apart, the probability that they both get detected as poles would be low enough to not be detected as a pair.

## II. Map Exploration

*A. Initial Approach*

The purpose of this project is to search a known space given an occupancy grid. After testing out the move_base commands allotted to us by the localization node, the robot would consistently fail to complete the predetermined path due to inadequate obstacle avoidance, proving itself to be unreliable. We also attempted to use a few packages that offered the *Frontier Explorer* (FE) algorithm, and none of them seemed to do exactly what we wanted [2,3]. We would receive premature notifications that the map was "fully explored". We decided to scrap the planners and explorers available to us and develop our own explorer that utilizes drive commands with twist messages.

Naively, we believed that it would be as simple as generating a single road map. Using the *Traveling Salesman Problem* (TSP) algorithm we could traverse the whole map in a loop while using *A** to generate the most efficient path from node to node in the TSP graph.

*B. Path Planning with A**

In the interest of generating path planning, we decided to go with AMCL due to being in map frame and saving us a transformation from odom to map using /odometry/filtered, and the global costmap for localization. The first step in this problem was to be able to map a connection between a pose in map frame to an index in the occupancy grid. We used a package called occupancy_grid_utils after discovering a github page for it [4]. After reading through the source code, we found that it contained all the tools we needed to start implementing *A** and map the pose to an index for a given map.

After integrating the package to our code and implementing code that updates the global costmap, we found that there were synchronization issues between what the rover's pose was and what the global costmap was telling us. At time $t_0$ of the robot spawning in the simulation, *A** path planning works marvelously and we can see the robot navigating around obstacles following a predetermined path with high precision. Given this, the *A** path planning we had only worked well for short distance planning to a goal close enough relative to the robot.

This meant that we could not, with high reliability, follow a road map with our current level of understanding. We would need to spend more time and be able to transform the road map as we traversed the map to account for drift. What this means is that if we passed in the point *(4, 4, 0)* in map frame to our path planner at some time $t_0$ and then later at some time $t_1$, we would receive different paths leading to different locations.

Instead we went with a random walk that utilized *A** path planning to a random point around the robot that satisfied a few conditions:

1. The random pose had to be *8.5m* (an arbitrary value) away.
2. Directly in the front half of the robot
3. In an obstacle free space within the arena.

Far enough that it would not stay in the same area for too long and avoid staying on the edges of the map. If a path is failed to be generated, we rely on a simple way to traverse:

1. Move forward, unless there's an obstacle
2. If there's an obstacle, avoid it by turning in the direction with the shortest angular distance.

Which would put the robot in a different location that would allow the path planner to generate a valid path to be used.

## III. References


1. "Introduction to Working With Laser Scanner Data." *Ros.org*. N.p., n.d. Web. 05 May 2017. <http://wiki.ros.org/laser_pipeline/Tutorials/IntroductionToWorkingWithLaserScannerData>.
2. Neuhold, Daniel. "Explorer." *Ros.org*. N.p., n.d. Web. 05 May 2017. <http://wiki.ros.org/explorer>.
3. Bovbel, Paul. "Frontier_exploration." *Ros.org*. N.p., n.d. Web. 05 May 2017. <http://wiki.ros.org/frontier_exploration>.
4. Marthi, Bhaskara. "Occupancy_grid_utils." *Ros.org*. N.p., n.d. Web. 05 May 2017. <http://wiki.ros.org/occupancy_grid_utils>.