Evan John Gonnerman
Connor L Yates
Jiongcheng Luo
Yeang Kong Zheng

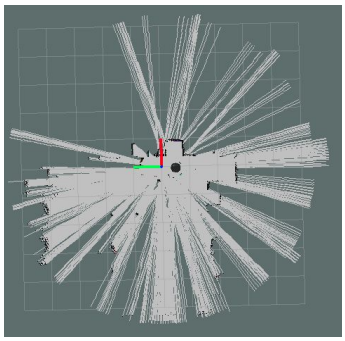
Final Report

Exploration Problem:

The situation to be solved, for this project, is to map an enclosed area with multiple obstacles scattered about. A robot is dropped in a random location in this map equipped with a computer running ROS, a 360^{o} laser scanner, and odometry encoded wheels. The mission of this robot is to explore the enclosed area until all landmarks and unknown locations have been marked. The robot will take laser scans continuously as it moves throughout the environment returning a map as occupancy data. This map is represented as an occupancy grid with values of -1 and 0-100. A value of -1 represents an unexplored area, 0 represents a verified open area and 100 represents a verified obstacle. Any value between 0 and 100 will be a confidence (probability) that the cell is occupied. For example a value of 0.5 means there is equal certainty that the cell is occupied as it is empty. As the robot traverses the map it will verify cells as occupied/empty until all cells have been seen. The situation will be complete when there are no remaining unknown locations adjacent to open cells.

Gmapping and nav_bundle Packages:

Figure 1



The robot will rely on two methods of traversing through the unknown world. By running the gmapping package the robot will utilize its onboard laser sensor to map its surroundings. This package will convert the reflected laser data of the sensor into the occupancy grid discussed in the exploration problem. Shown in Figure 1 is an example output of what the occupancy grid will look like with grey cells representing open space and black cells representing occupied space. In order to navigate through the world the robot runs the nav_bundle package. This allows the robot to subscribe to control velocity commands and odometry data from encoded wheels. By simultaneously running both packages the robot will simultaneously move about in the world and build a map around itself that physically represents landmarks seen.

Waypoint Setting Algorithm:

There are three steps necessary to set a waypoint and navigate towards it. The first step involves using the occupancy grid output from gmapping to select a viable waypoint to go to (an unknown cell adjacent to an unoccupied cell). The second step calls upon a path planning algorithm (A* for this project) to provide a series of vertices that will result in the robot traveling from its current position to the waypoint's x-y location. The third step is the navigation stage that sees the robot physically following the vertices provided by the path planning algorithm and eventually arriving at the waypoint. This process is then repeated until an end condition has been met. The details of each step are as follows:

Evan John Gonnerman
Connor L Yates
Jiongcheng Luo
Yeang Kong Zheng

Waypoint Selection:

The waypoint selection algorithm is a frontier based search. On a high level, the occupancy grid is scanned in a linear fashion. When a cell is an seen to be “unoccupied” (with probability < 10), it checks the immediate neighbors to see if they are unobserved. If they are, then point is known to be on the frontier, and it will send this point to the navigation code.
This was originally implemented with two nested for loops, and while it would find a point, it took quite a long time (over 20 seconds), and this caused massive delays and backups in the entire system.
To resolve this, I turned to numpy to find some black magic. I knew that I wanted to select all of the unoccupied cells, then scan these cells for neighboring -1 values. To select all these unoccupied cells and get their indices, the following pure black magic was used:

```
indicies = np.transpose(np.where((map2d.data > -1) & (map2d.data < 10)))
```

How this works:

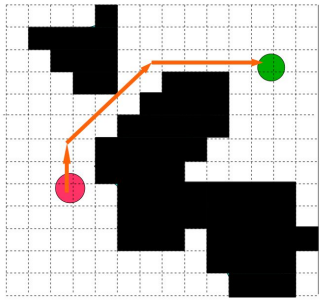
1. map2d.data stores the occupancy grid data, in a 4000x4000 numpy array. When a numpy array is compared against a single value (map2d.data > -1), the result is a n x m dimensional array of booleans representing the result of the element at index (n,m) in the boolean expression.
2. To get unoccupied yet seen indices, we select all values seen (>-1) and unoccupied (<10). The arrays are combined with AND to calculate the final boolean array which maps out all of the indices that meet our conditions.
3. To extract the indices, np.where() takes in the boolean array and returns the indices in a 2 x n array. It splits the indices into 1 array of column values, and 1 array of row values. This is the prefered format for some other operations, but not for what we want, so we return the transpose of this array to get 1 x n array of the indices.

With this list of indices on the frontier, we scan each one to see if it borders a unseen value. If this is the case, this index becomes the next waypoint to move toward.
With the numpy optimizations, the runtime went from around 20 seconds, to around 0.2 seconds.

Path Planning:

The path planning algorithm runs A* to generate a set of vertices for the robot to follow. It subscribes to the odom topic in order to receive odometry data on its x and y position. It also receives an x and y waypoint from the waypoint selection algorithm. Using a heuristic search algorithm, A* will calculate the optimal path to take in order to arrive at the waypoint while avoiding obstacles/walls. This is completed by subscribing to the map topic that holds an occupancy grid made by the laser scanner, this occupancy grid is an 4000 by 4000 cell grid, and each cell contains a value (-1 or 0 or 100) that represent if the cell is unseen (-1), empty (0) or occupied

Figure 2



(100). So that the A* graph search algorithm can generate a safe and shortest path for the robot movement.

Once a set of vertices have been created the script will publish vertices one at a time to the navigation script. Each time a vertex is met, the path planning algorithm will output a new vertex for the robot to navigate to (next position of the path). Finally, once the robot arrives at the desired waypoint, a new waypoint will be asked for from the waypoint selection script and the process will be repeated until an end condition has been met.

Navigation:

The navigation algorithm subscribes to the odometry data produced by encoded wheels on the robot, a waypoint topic that receives vertices from the path planning algorithm, and the command velocity of the robot. Once the path to a waypoint is set and a path is planned the robot will incrementally ask for a single vertex from the path plan. For each vertex, the robot will update its currently held position of itself with respect to its known world map. It will then check to see that it has stopped and set a waypoint to the next vertex. This waypoint has an x-offset, y-offset, and angular offset. The robot will send an x-goal, y-goal, and angular goal to the base_link_goal topic. This will lead the robot to the given location and stop. After the goal has been reached, the robot will ask for a new goal from the path planning algorithm. This process is repeated until the path planning algorithm has exhausted all vertices and the robot has reached the desired waypoint. At this point, a new waypoint will be selected and a new path will be generated for the robot.

Discussion on Performance:

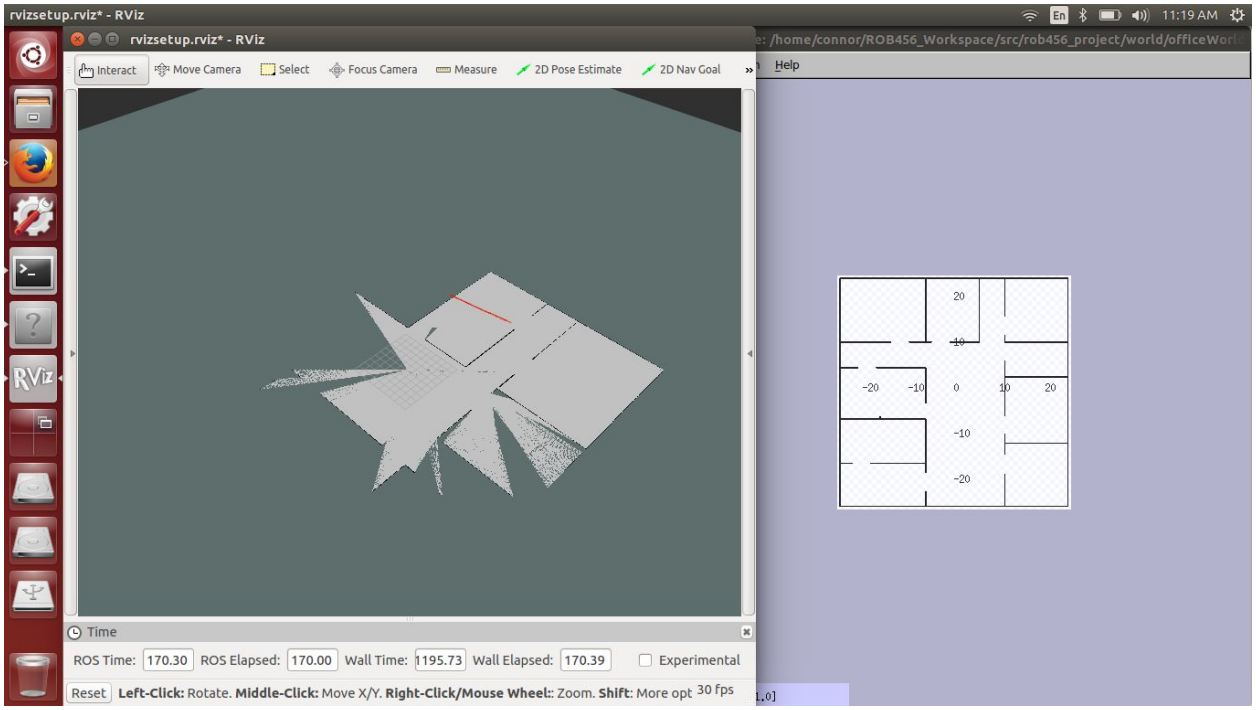
When each section (Frontier navigation, A*, and waypoint movement) were tested individually, they performed as expected. The frontier code would loop through the occupancy grid array, and pick a goal point to navigate to. A*, when provided a goal, would create a path toward the goal point, and the waypoint navigation would move the agent toward a given point, as best as it could.
The program broke down during the integration of these three parts. Communicating messages between each piece of code required substantial rewrites of the waypoint movement and A* programs, so that their interfaces would robustly work in a concurrent manner.
A partially-working version was produced that bypassed A* entirely, and simply passed goals to the waypoint movement algorithm. In this version, the world would be partially mapped but the robot would eventually reach a blocking state, and refuse to move further. This blocking state appeared to be “parallel parking” against a wall, which would prevent rotation of the robot, preventing further motion. Screenshots of the mapping under this control policy are provided on the next page.
To improve the performance, we believe fully integrating the A* algorithm into the policy would help to build a more efficient path that is able to avoid obstacles without winding around parallel with the walls/obstacles. In addition, due to the expansion of the explored map, the current A*

algorithm takes growing time to scan possible paths along with the robot moving, to improve the scanning/exploring method of the A* algorithm is to find out a proper scan resolution of the scan range, for example, to scan 10 cells rather than 1 cell at a time.

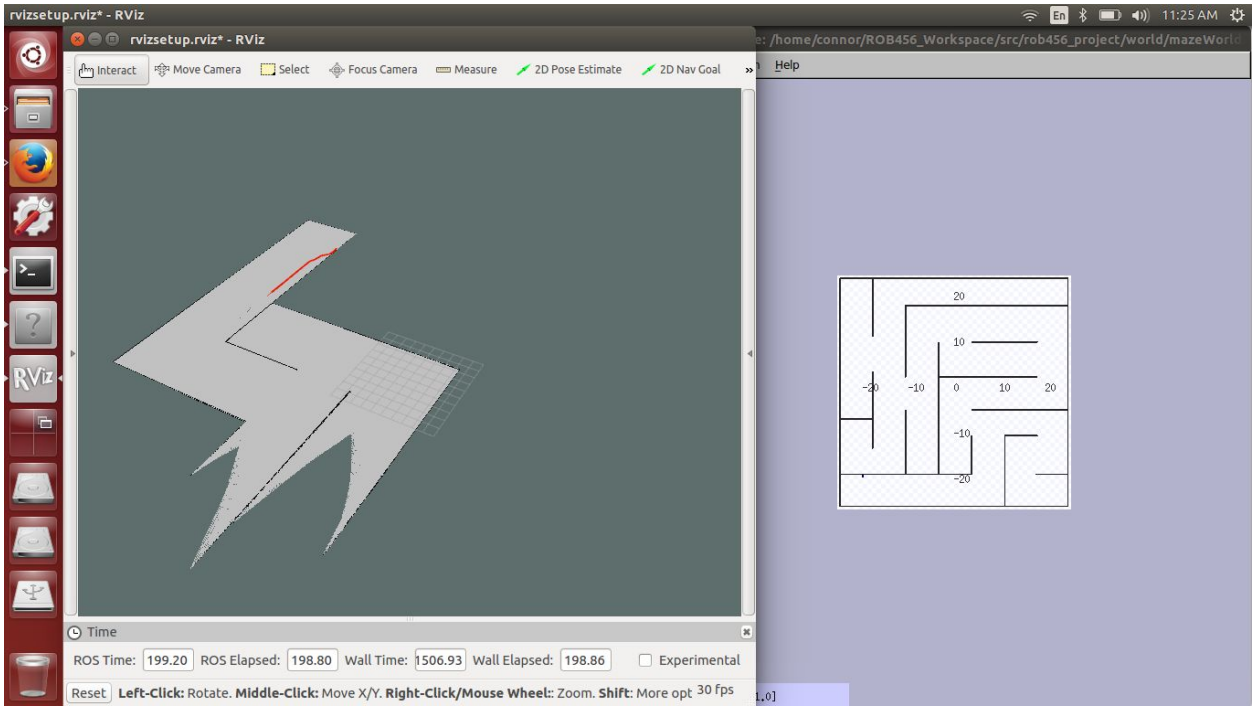
One more method to prevent the robot from getting stuck would be to pick a more intelligent frontier point. Currently, it provides the first point *directly on the frontier*. This seemed to direct the robot toward points next to objects, where it would get stuck. I would rather have the frontier point be a safe spot in the observation radius of a large section of the frontier. This would move the robot to a spot away from walls by moving the robot to a place where it would look to *observe* to expand the frontier, but not move directly on top of the frontier.

Evan John Gonnerman
Connor L Yates
Jiongcheng Luo
Yeang Kong Zheng

Map of the office world:



Map of the maze world:



Evan John Gonnerman
Connor L Yates
Jiongcheng Luo
Yeang Kong Zheng

Discussion of Extra Credit Solution:

Extra Credit 1 (Code for A* and BFS provided)

Three random coordinates was tested on A* and Breadth First Scan (BFS). Based on the run times, A* required the shortest time to complete despite the higher number of nodes expanded. This is due to the fact that BFS needs to scan every node in the open list until the goal node is reached whereas A* selects the node with the smallest cost value.

When travelling through an area with only open cells, A* produces a staircase pattern while BFS stays close to walls and has minimal turns. In real life application, BFS might have a better quality of performance because it has less stops and turns. This would reduce drift, slipping and sensor noise despite its higher run time and high memory usage.

		Run Time	Nodes Expanded
Test 1	A*	0.0081 s	347
	BFS	0.0097 s	258
Test 2	A*	0.0015 s	54
	BFS	0.0030 s	57
Test 3	A*	0.0101 s	423
	BFS	0.0104 s	286

Extra Credit 2 (since we had a 4 person group):

See the frontier waypoint selection section.