Jackie Scanlon
Algorithmic Robotics - Programming Assignment 4
4/25/19

Code guide at the end of the document.

Part 1: Localization and Mapping

Tasks

1. Dead reckoning localization.

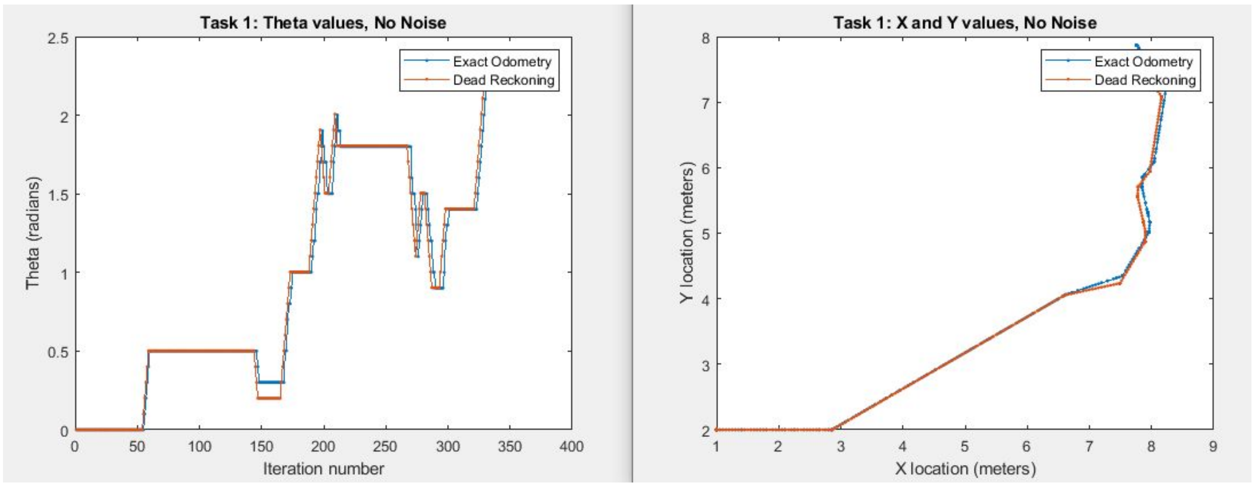


Figure 1.1: *The left plot shows theta values computed via the exact odometry and through my code using dead reckoning. The right plot shows the X and Y values for the same computations. In this plot, the controls have no noise added.*

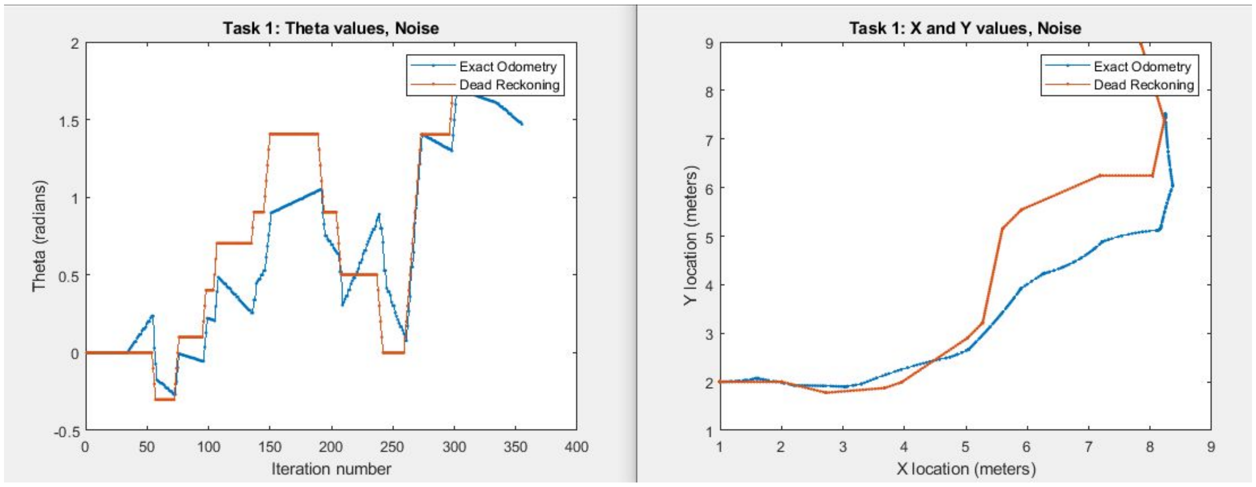


Figure 1.2: *The left plot shows theta values computed via the exact odometry and through my code using dead reckoning. The right plot shows the X and Y values for the same computations. In this plot, noisy controls are in included.*

2. Extended Kalman Filter localization.

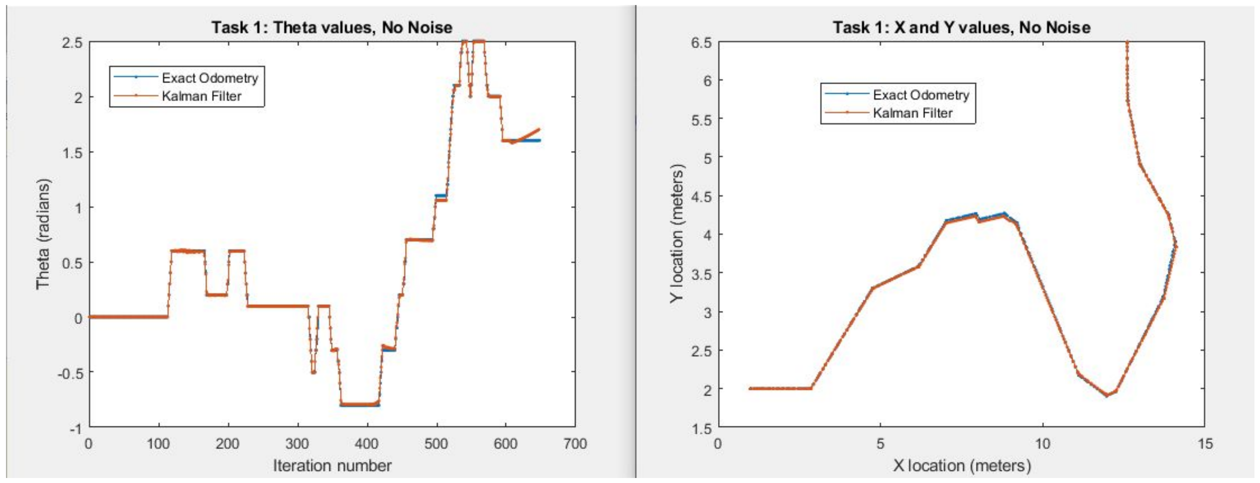


Figure 1.3: *The two lines indicate exact odometry and my EKF. The left plot shows theta values and the right shows X and Y values. In this plot, there is no noise in either the controls or the landmarks.*

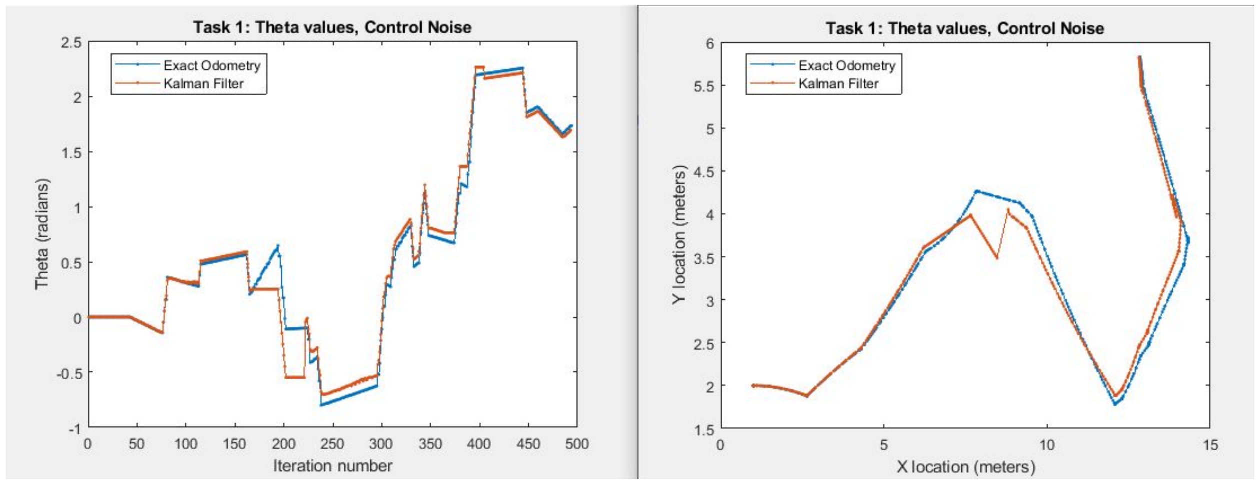


Figure 1.4: *The same as above, but in this plot, there is noise in the controls but not the landmarks.*

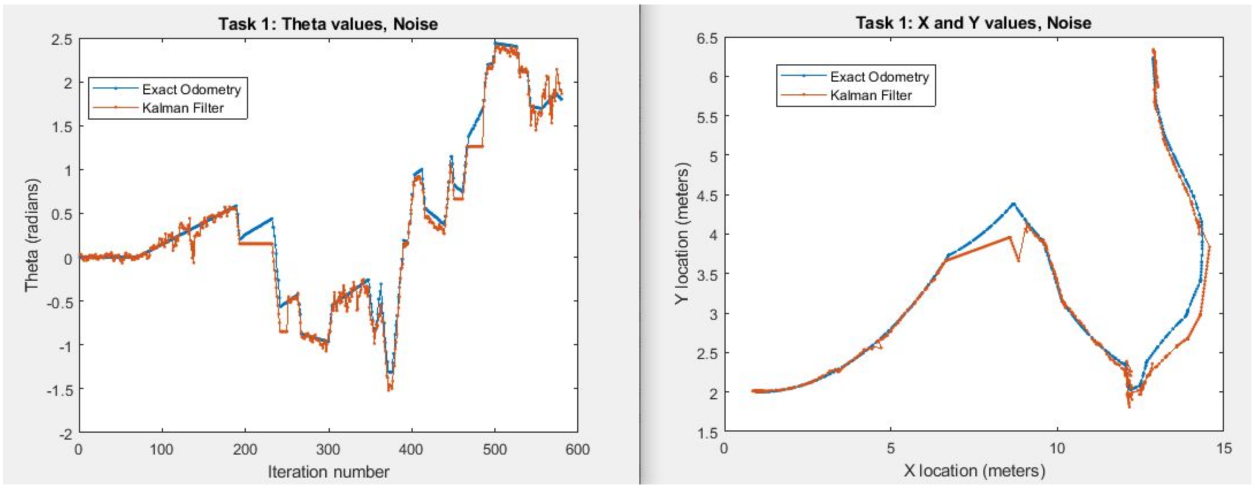


Figure 1.5: *The same as above, but with noise in both the landmarks and controls.*

3. Static Occupancy Grid.

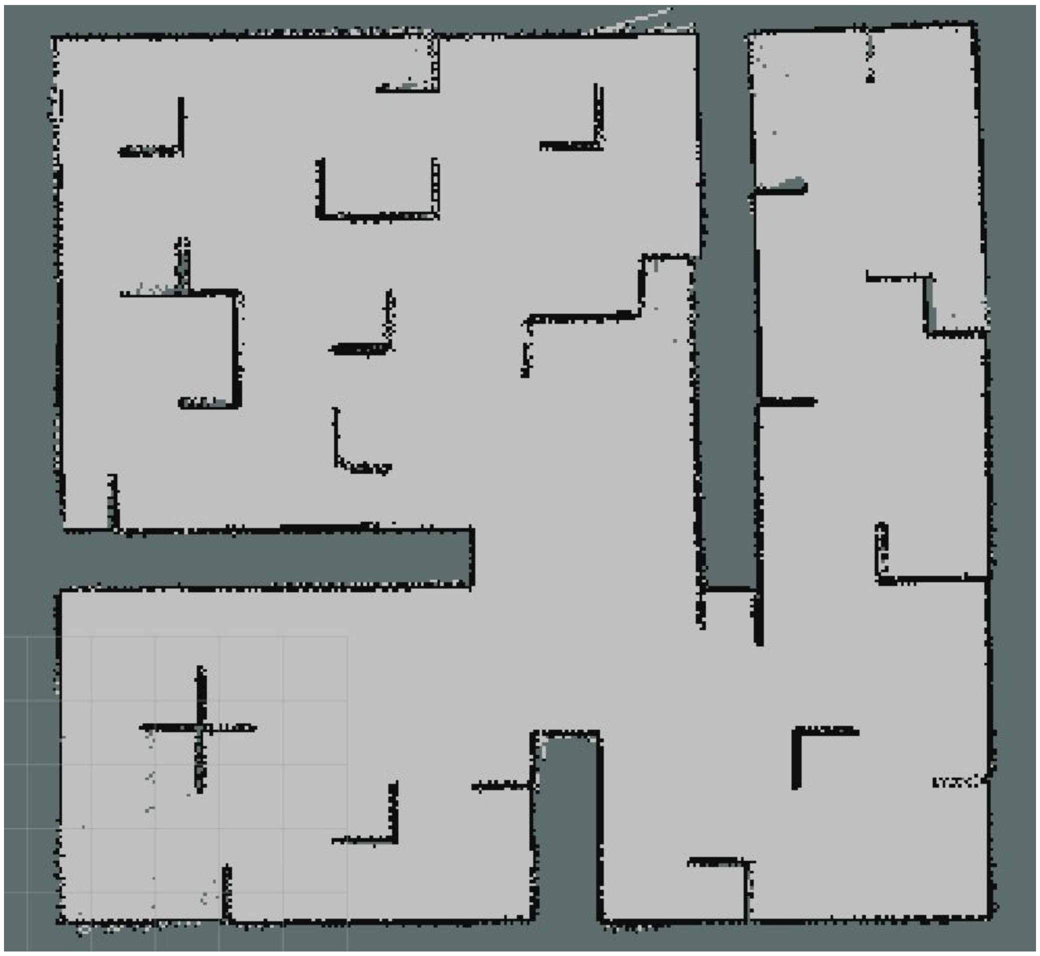


Figure 1.5: *The occupancy grid produced by the GMapping package. In the creation of this map, no noise was added.*

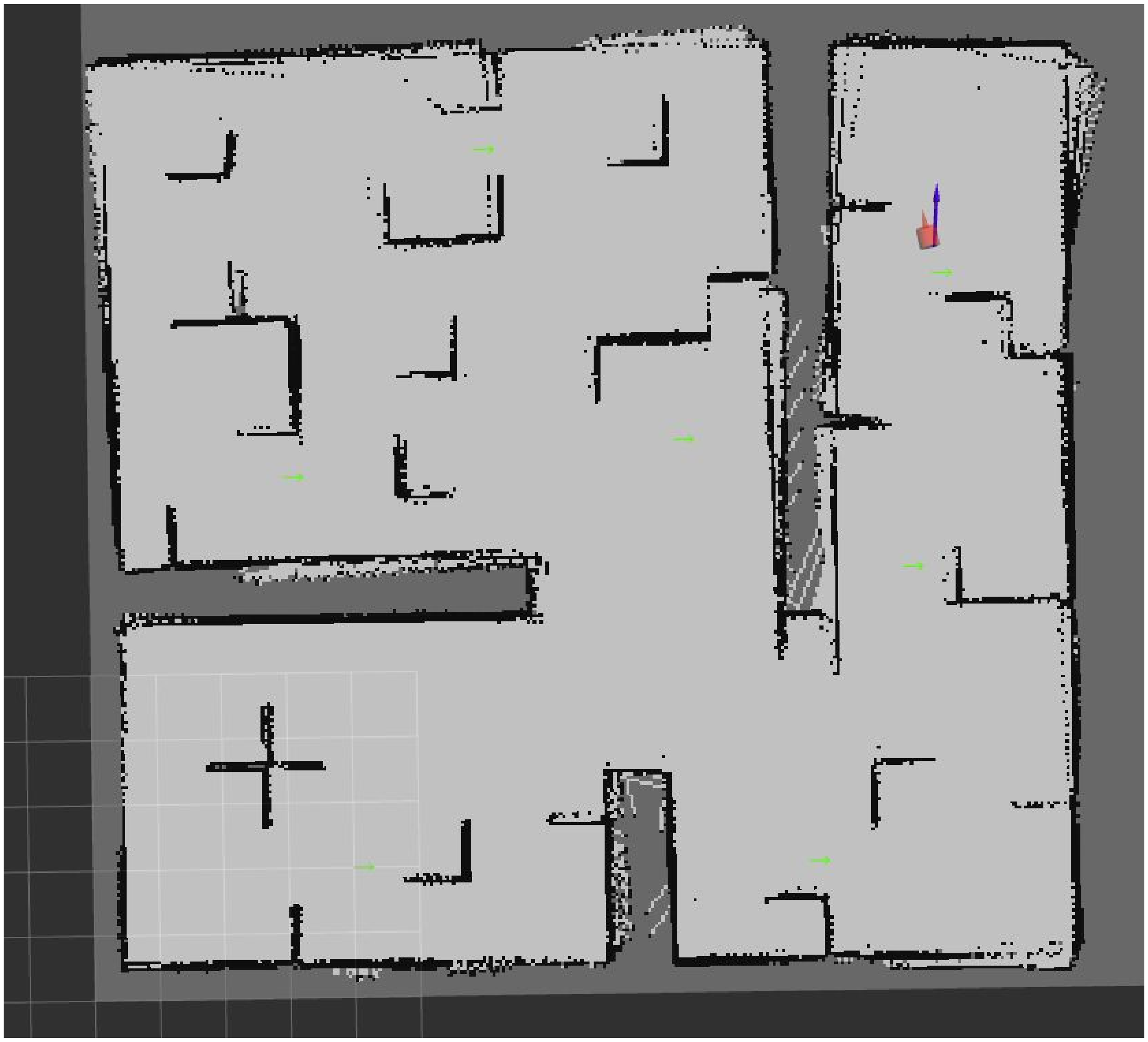


Figure 1.6: *The occupancy grid produced by my code, using my mapping.py occupancy grid creation and my kalman filter to provide the pose estimation. In the creation of this map, no noise was added.*

4. Dynamic occupancy grid.

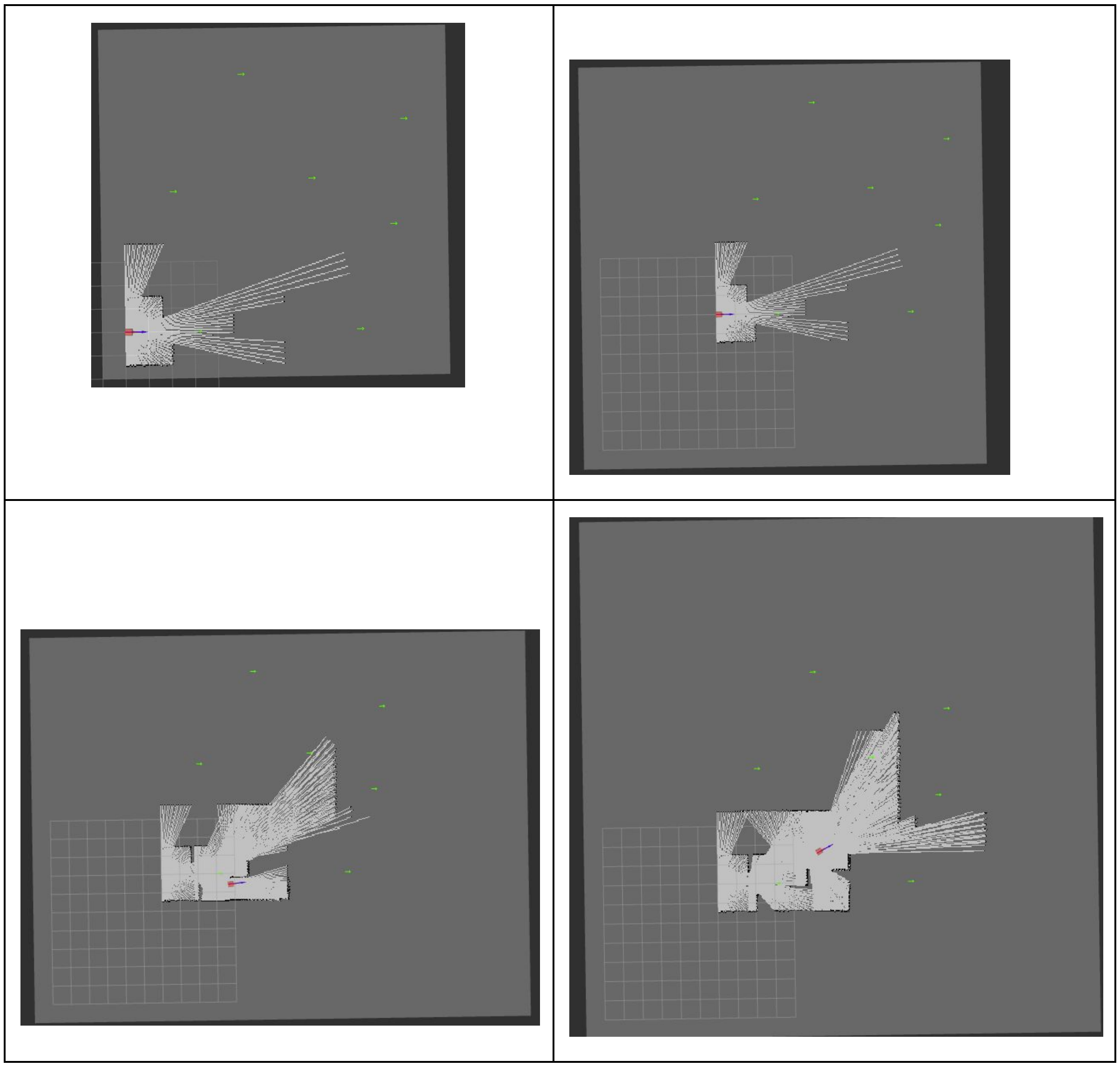


Figure 1.7: *The dynamically changing occupancy grid. The dark grey area encompasses the entire grid, with light gray being empty cells and black being occupied cells.*
Top left: Original map size (300x300). The robot senses that it is very close to the bottom left corner of the map, and thus expands both the bottom and left edges of the map, as can be seen in the top right image (420x420). Then, in the bottom left, the robot has moved closer to the right edge of the map and so expands the right edge of the map (420x540). Then the robot gets closer to the top of the map and so expands the top edge (bottom right image) (540x540).

Questions

1. Some of the advantages of dead reckoning are that it is extremely simple to setup and use. EKFs require lots of calculation, derivation and troubleshooting before they work correctly. Dead reckoning also doesn't require additional landmark information, which further simplifies the problem. No noise must be calculated or estimated to get dead reckoning to work as intended, nor must any tuning of parameters take place. However, when it comes to accuracy, dead reckoning falls far short of EKFs. I would not rely on dead reckoning to build maps or locate items in a world, given noisy controls.

2. The landmarks in EKF provide additional information that further narrows the probability distribution of the state of the robot - i.e., the landmarks not only use control information but also landmark information to make a better estimation of the robot's state.

3. If there was no noise in the controls, dead reckoning is near perfect. I would skip the complications of EKF and use dead reckoning alone, although I found that if the robot gets "stuck" on an object dead reckoning becomes completely useless, whereas EKF can recover from such issues. Thus, if there is a possibility of running into items, I would use EKF instead.

4. In a SLAM scenario, the robot has no idea how large the area it needs to map is. However, we don't want to allocate an enormously large area for it to map because that wastes precious memory. Instead, the dynamic map can account for this unknown large area while also keeping memory low.

Part 2: Motion Planning

Tasks

For these tasks, I rounded pi = 3.1416 and pi/2 = 1.5708. No preprocessing was performed for any of the maps. I did the tasks in the order of 3, 2, 1, (did not do 4), so that is the order I will present them here.

3. Given sparse_obstacles.yaml and the initial pose (8.5, 5.5, 0), I achieved the pose (14.4,12.5, π/2).

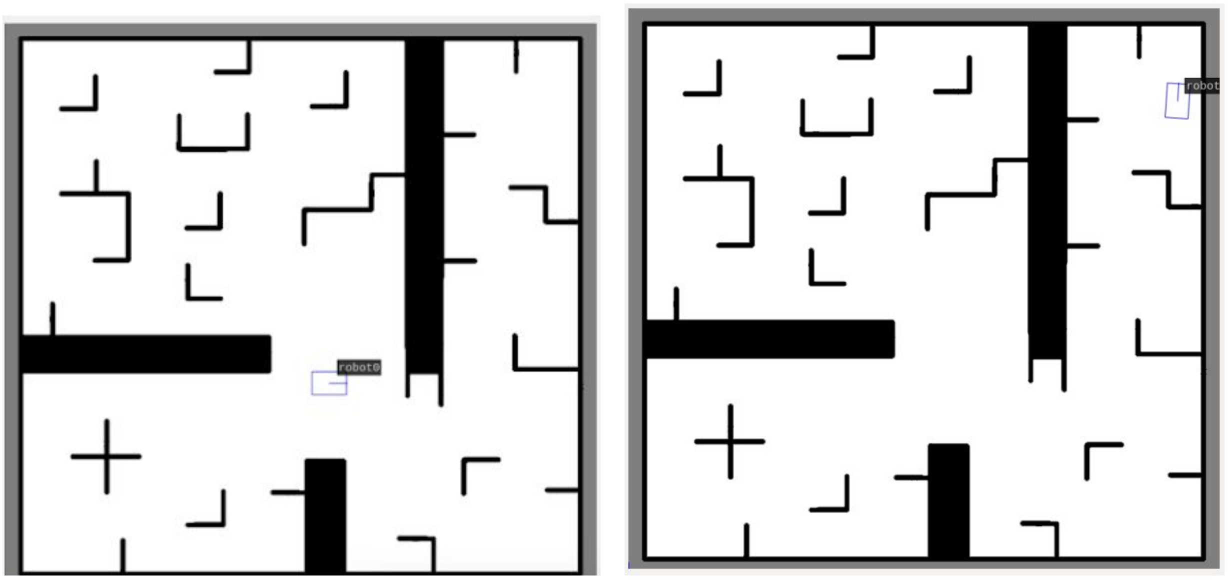
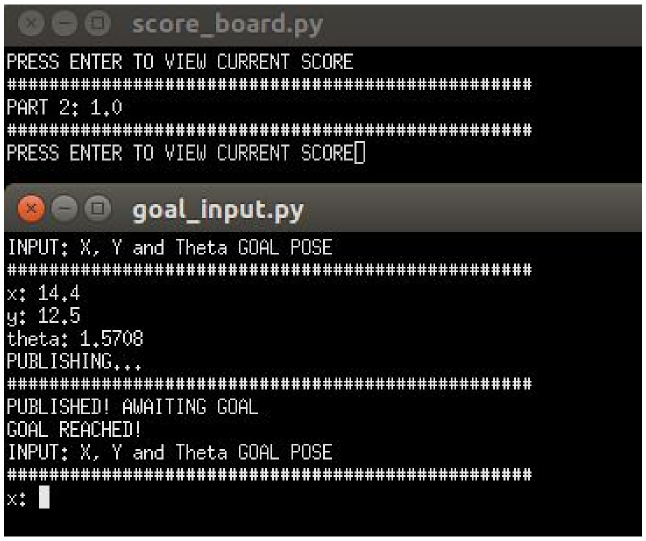


Figure 2.1: *The starting position of the robot (14.4, 12.5, pi/2), and the ending position.*



```
PRESS ENTER TO VIEW CURRENT SCORE
#####################################################
PART 2: 1.0
#####################################################
PRESS ENTER TO VIEW CURRENT SCORE
```

```
INPUT: X, Y and Theta GOAL POSE
#####################################################
x: 14.4
y: 12.5
theta: 1.5708
PUBLISHING...
#####################################################
PUBLISHED! AWAITING GOAL
GOAL REACHED!
INPUT: X, Y and Theta GOAL POSE
#####################################################
x:
```

Figure 2.2: *The scoreboard and input x-terminal windows.*

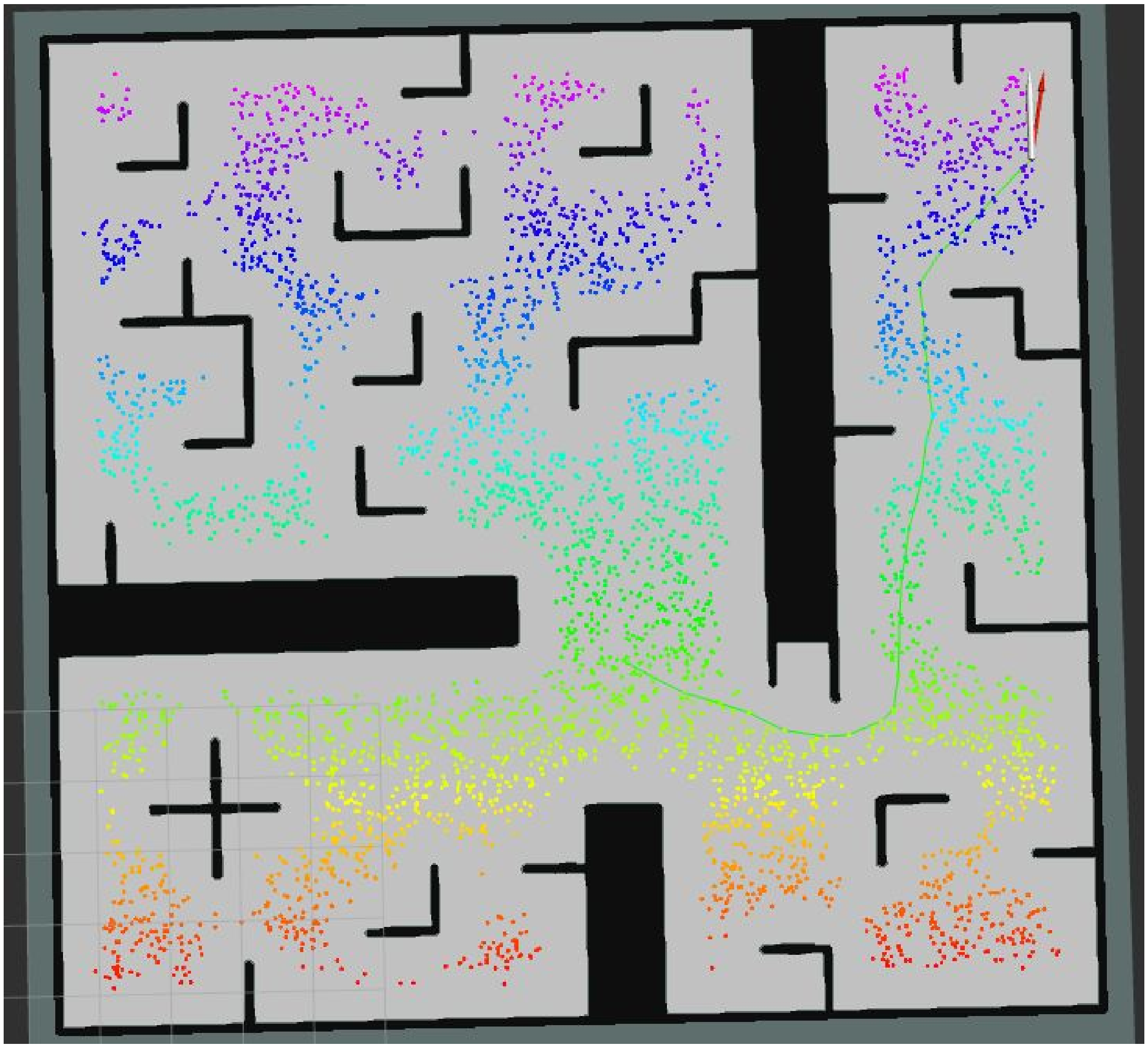


Figure 2.3: *The randomly generated points and path found and executed by the robot.*

2. Given simple_rooms.yaml and initial pose (3, 7.5, 0), I achieved the pose (18, 7.5, π).

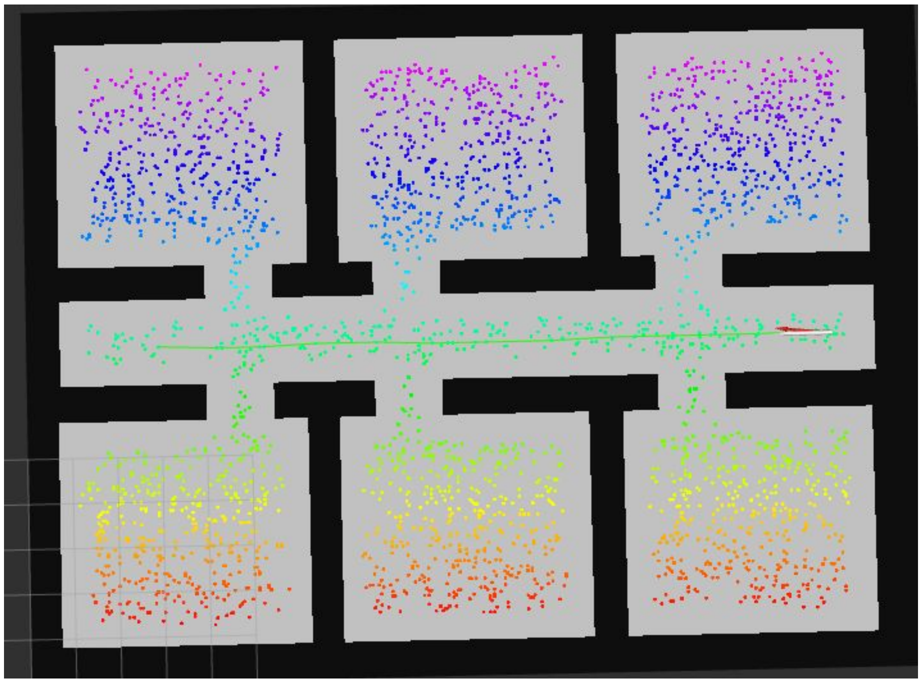
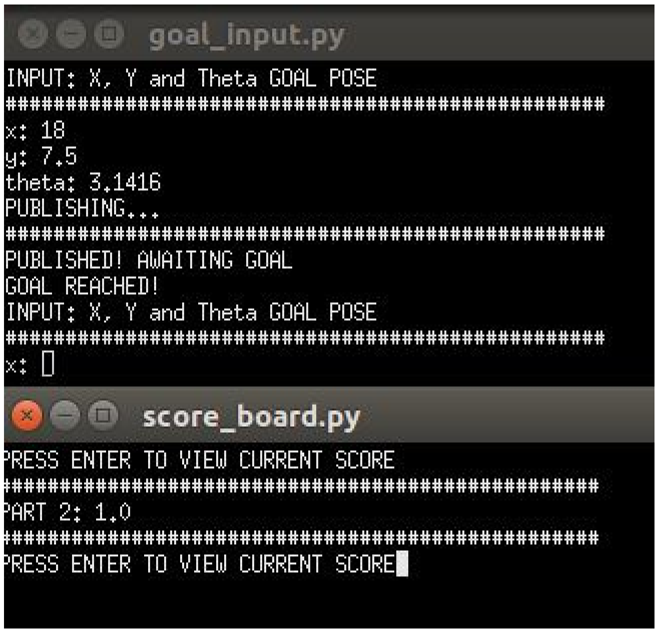


Figure 2.4: *Points and path generated for Task 2.*



goal_input.py

```
INPUT: X, Y and Theta GOAL POSE
####################################################
x: 18
y: 7.5
theta: 3.1416
PUBLISHING...
####################################################
PUBLISHED! AWAITING GOAL
GOAL REACHED!
INPUT: X, Y and Theta GOAL POSE
####################################################
x:
```

score_board.py

```
PRESS ENTER TO VIEW CURRENT SCORE
####################################################
PART 2: 1.0
####################################################
PRESS ENTER TO VIEW CURRENT SCORE
```

Figure 2.5: *Scoreboard and goal input for Task 2.*

1. Given hospital_section.yaml and an initial pose (1.5, 20, 0), I achieved the following poses: (4, 15.4, 0), (50, 15, 0), (38, 10.5, π). I did not achieve the last 2 poses, (31.75, 1, π/2) and (30.8, 20, −π/2). One thing I did differently in this graph from the previous tasks was since that map was so big, I only added points that were more than 10 cells away from any other points. This slightly increased the time necessary to create the points but drastically reduced the time necessary to make the graph and path.

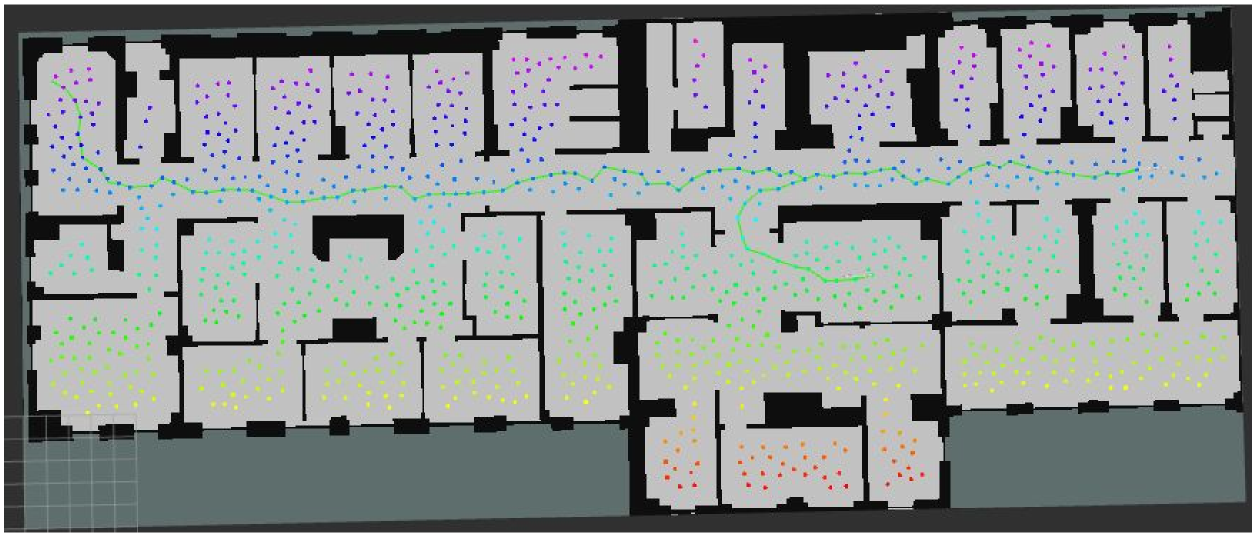
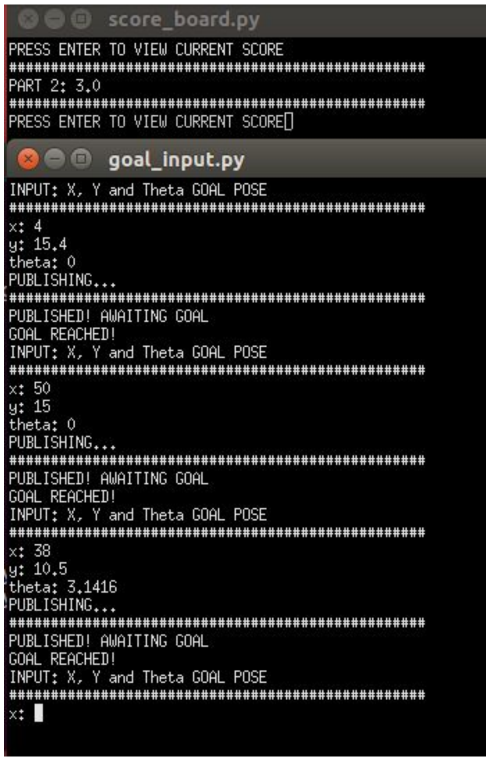


Figure 2.6: *Points and paths generated for Task 1.*



```
score_board.py
PRESS ENTER TO VIEW CURRENT SCORE
##################################################
PART 2: 3.0
##################################################
PRESS ENTER TO VIEW CURRENT SCORE
```

```
goal_input.py
INPUT: X, Y and Theta GOAL POSE
##################################################
x: 4
y: 15.4
theta: 0
PUBLISHING...
##################################################
PUBLISHED! AWAITING GOAL
GOAL REACHED!
INPUT: X, Y and Theta GOAL POSE
##################################################
x: 50
y: 15
theta: 0
PUBLISHING...
##################################################
PUBLISHED! AWAITING GOAL
GOAL REACHED!
INPUT: X, Y and Theta GOAL POSE
##################################################
x: 38
y: 10.5
theta: 3.1416
PUBLISHING...
##################################################
PUBLISHED! AWAITING GOAL
GOAL REACHED!
INPUT: X, Y and Theta GOAL POSE
##################################################
x: 
```

Figure 2.7: *Scoreboard and goal input for Task 1.*

Questions

1. I implemented PRM as my planning algorithm. It is a simple algorithm to implement and the one I understood best. The drawbacks are that it is quite slow; however I was able to keep it within a reasonable time (1-3 minutes for most computing).

2. I did not really do any preprocessing beforehand, but I could've performed the random point generation and graph generation beforehand. Then the only thing to do would be perform the search algorithm.

3. None really - the only difference was I did not command the robot to go to the selected theta angle for each point. Instead, I had only had it go to the selected theta for the last point. This made the path much smoother, and since the path had many points, the theta values were generally in the correct direction (and were feasible) anyways.

4. I used Dijkstra's algorithm, because it was the simplest to implement and I knew it best. To move along the path planned, I used the bike control laws we have developed in previous programming assignments - the one without a specific theta for all the points except the last one, which had the control law for the specific theta.

5. None really for navigation. I could've used BFS or DFS to get a path - just didn't choose those because I found them personally a little harder to implement since I have worked with them less.

Part 3: Search and Rescue

Tasks

1. I had my robot drive around to 10 different random goal locations. It covered the map decently - could've had more area on the upper left. The green paths show the places the robot traversed, and the white dots show the places the robot had as a goal location.

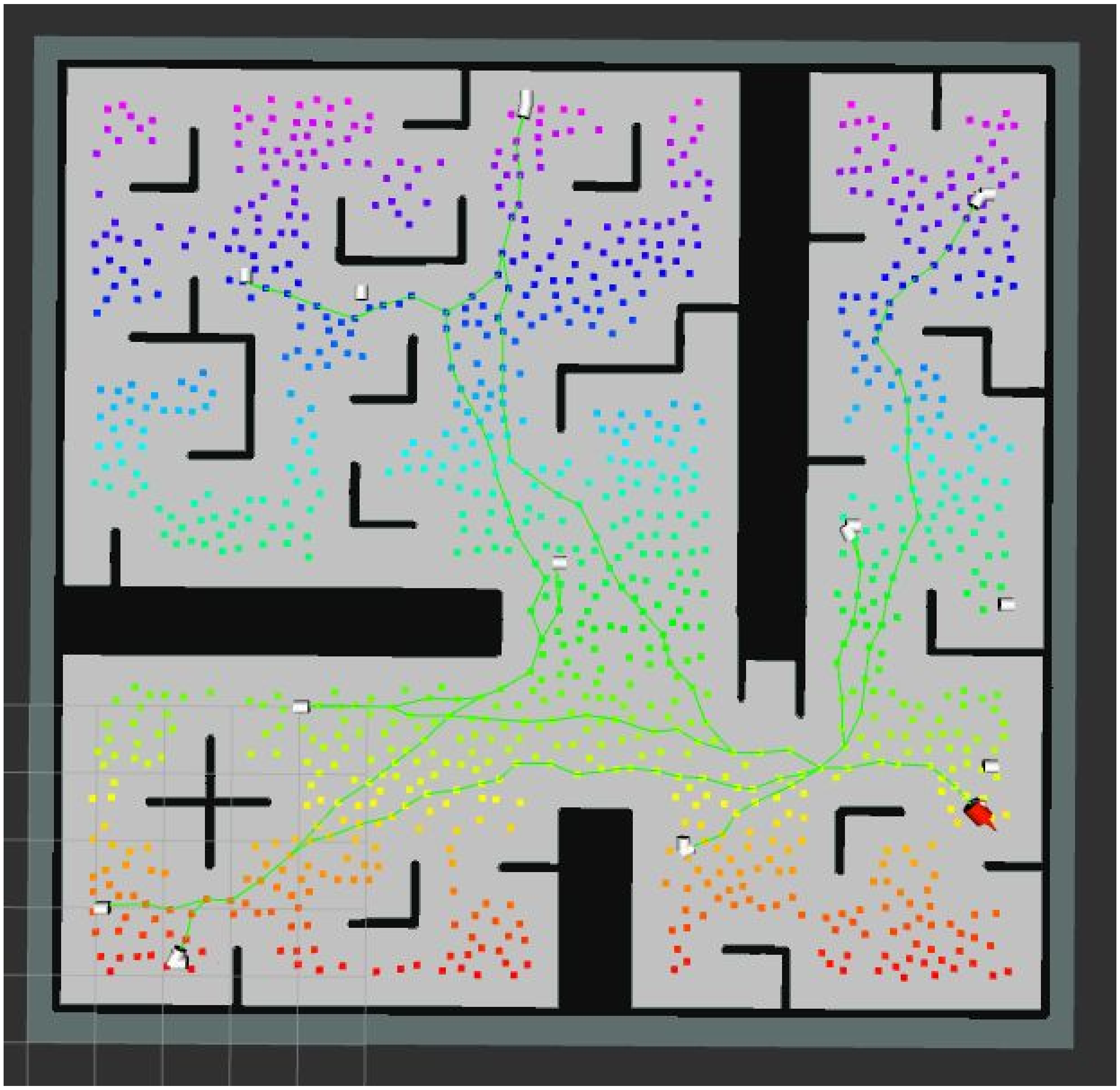


Figure 3.1: *The robot drives to several locations autonomously around the map.*

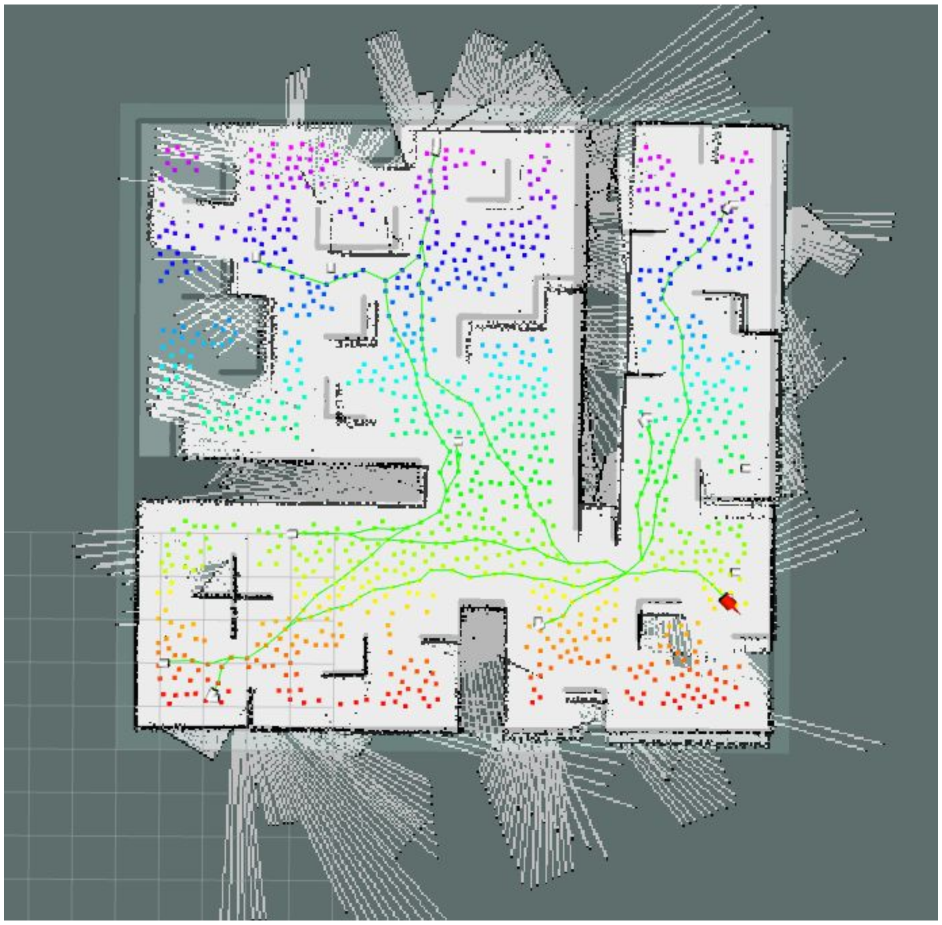


Figure 3.2: *The map that gmapping made using the randomly generated points. It's not stellar, but as you can see the robot was able to "see" most places on the map - i.e., given victims all over the map, the robot would've most likely come into contact with at least some of them at some point.*

2. I had my robot start driving towards a random point. Once it saw a victim, it stopped and attempted to find the average location of the victim. Then it generated a path towards it and executed the path.

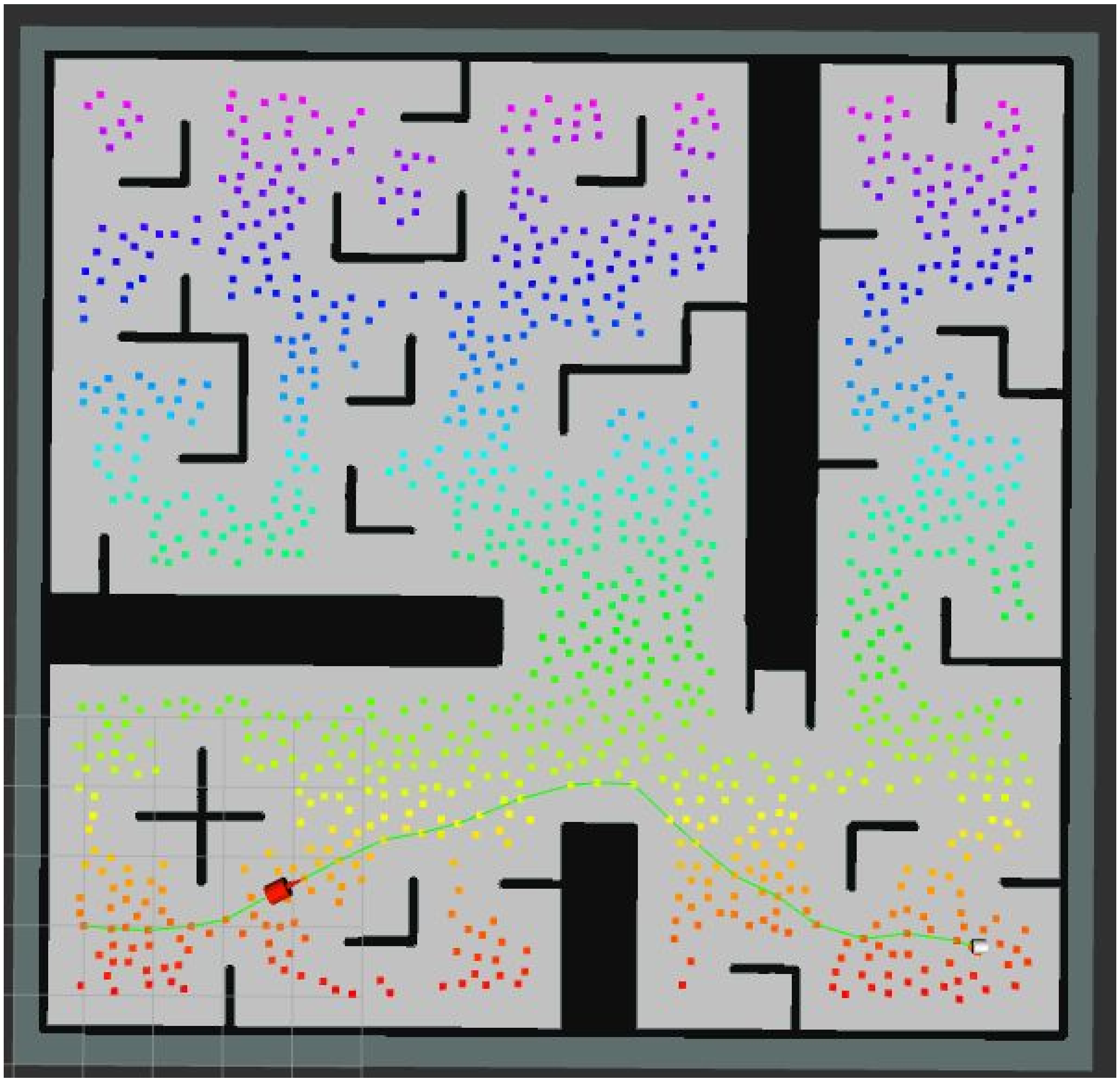


Figure 3.3: *The robot starts heading towards a random point.*

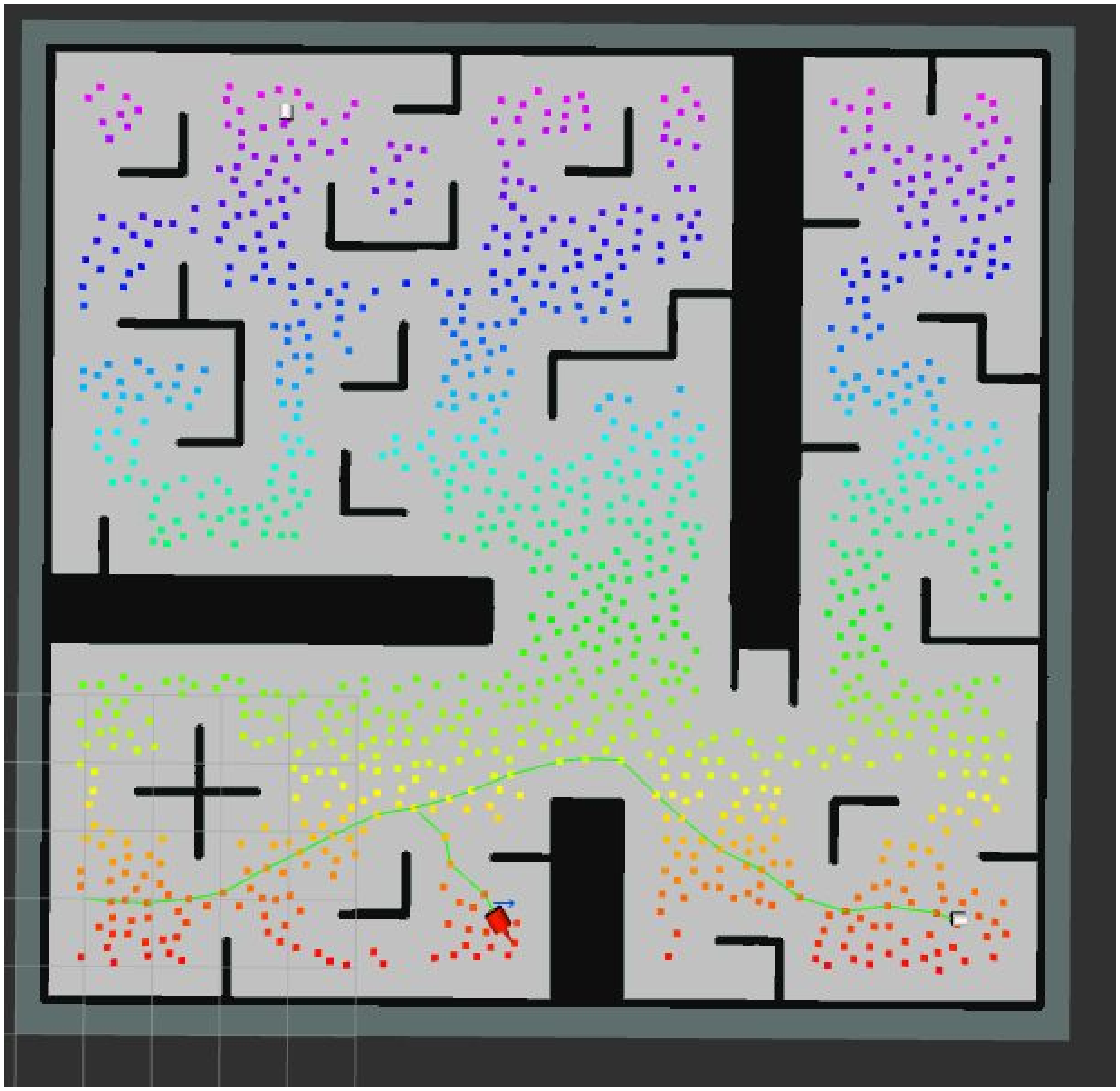


Figure 3.4: *Along the way, it sees the victim (blue) with noise. It stops, averages the position for a second, and then drives to the victim.*

Questions

1. A sensor that is not on the robot that could help is a collision detection sensor. For example, the Kobuki robot base (http://kobuki.yujinrobot.com/about2/) has a front bumper that senses when it is pushed. Then the robot knows it is “stuck” and can command backwards velocities to get itself unstuck before trying to find a new path forward. As it stands, if robot0 gets stuck, that’s it - the only information that really tells it that it isn’t stuck is the fact that the landmarks are not getting closer/farther away.

2. Each time we find a victim, we can keep track of that known victim in a list. Whenever we see a new victim published on /victims, we can compare that location to the victims in our list. If that new victim's location is within a certain euclidean distance of any of the victims on our found known victims list, we ignore that victim publication. The distance threshold would most likely have to be determined by trial and error, but it would be based on the noise of the victim's location publication. If this were a real life scenario, we could assume that the victim is of a certain size (person size). Thus if a new victim is published on /victims and is within, say, a 2ft radius of an old victim, then it is probably the same victim. We could make that a little smaller for noise considerations, and then once we get closer we will be able to differentiate (or not) between known and unknown victims.

3. If no map at all was given, I would resort to a bug algorithm, probably tangent/vis bug. It makes use of the short range of the lidar. When a victim is in sight, it uses bug mode. When there are no victims in sight, it heads towards an area of the map where there is lots of open space (i.e., many grid cells with values of -1). Thus the robot will be able to find victims that are in unmapped areas.

4. The disadvantages are that the bug algorithm could find a not-so-great path, one that is very long, for instance. It also can only update so fast, so it requires a lidar scan each time. The advantages are that it is simple and works in a unknown environment. You cannot use PRM if you can't see any of your map.

5. The advantages are that it generally works and converges, and plans a smooth and .It may have a hard time working with obstacles (at least the way that I set up part 2). It also doesn't really allow for easy processing in terms of a partially known map - if it can't find a feasible path, it basically needs to move towards an unmapped area before attempting to find a new feasible path.

Other things I learned

- I learned a lot about the synchronization of subscribers - between the laser scan and the odometry/Kalman filter/dead reckoning, and found several different ways to correct for the misalignment of these messages.
- I learned a lot about PRM, and was able to implement my version of the algorithm. I also found some places where my algorithm, and PRM in general, fails.
- I learned a ton about EKFs, and this project really solidified my understanding of it from what I learned in class and for the exam.
- I learned a lot about RVIZ - I was able to really configure RViz so that I could use it as the proper debugging tool it is, instead of just struggling around with the settings for awhile. I was able to create different publishers so I could view my algorithm working (such as with PRM) and overlay different algorithms so that I could compare them working in real-time, such as with the odometry, dead reckoning, and kalman filter.

- I customized my launch files a lot for part 1, so I learned a lot about arguments in launch files, XML syntax, etc.
- In general, I brushed up a lot on my numpy and python knowledge, as well as ROS message types.

Guide to the Code

Part 1:

For the tasks: slam.launch

- Different args turn on different nodes (static vs. dynamic map, gmapping, kalman and dead reckoning)
- kalman_filter.py does the kalman filtering
- dead_reckoning.py does the dead reckoning
- mapping.py makes static map (currently subscribes to kalman filter as pose estimator)
- dynamic_mapping.py makes dynamic map
- gmapping can also be used
- slam.rviz

For the demos:

- Section_1_demo_1.launch and section_1_demo_2.launch
  - Uses landmarks_demo.py
  - Have defaults set correctly for demos
  - Can set landmark_noise:= false for no noise at all (the other noises are default true)

Part 2:

- prm.py contains the path planning algorithm.
- Change self.K to change number of points plotted
- Change self.max_rad to change how close points need to be to get connected by the graph
- Change self.far to change how far away a point has to be from other points to get added to the graph
- All uses navigation.rviz (toggle different things on the left side)
- When launching navigation.rviz, change the map name and x, y, theta arguments appropriately

Part 3:

- Task 1: roslaunch search_rescue_task1.launch
  - Uses drive_around_task1.py and path_follower_task1.py as the PRM/goal generation node and path following node, respectively
- Task 2: roslaunch search_rescue_task2.launch

    - Uses drive_around_task2.py and path_follower_task2.py to start heading towards some goals and change route to head to victim once it sees a victim. Then stop.
- Demo: roslaunch search_rescue_demo.launch
    - Uses drive_around_demo.py and path_follower_demo.py to go towards random goals. If it sees a victim, it stops and averages the victim position, and then goes to the victim. Once it reaches it, it adds it to a line in the csv file. Then it starts heading to another random goal, and has a 30 second buffer where it ignores victims as it drives so that it doesn't just end up heading back to the same victim it was just at.