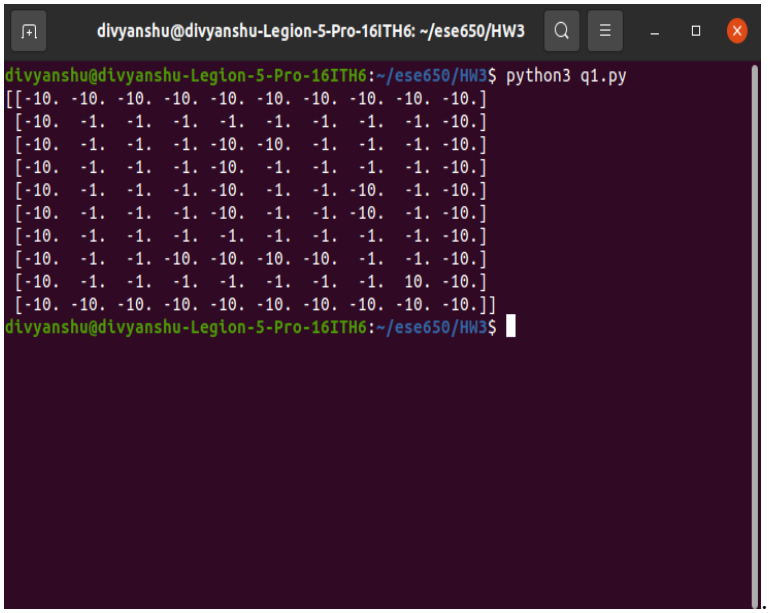
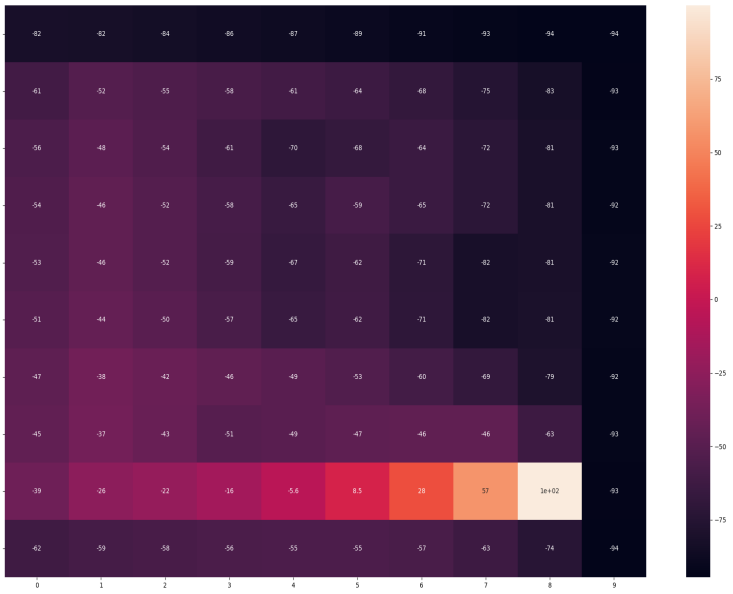
ESE 650 HW3

Divyanshu Sahu

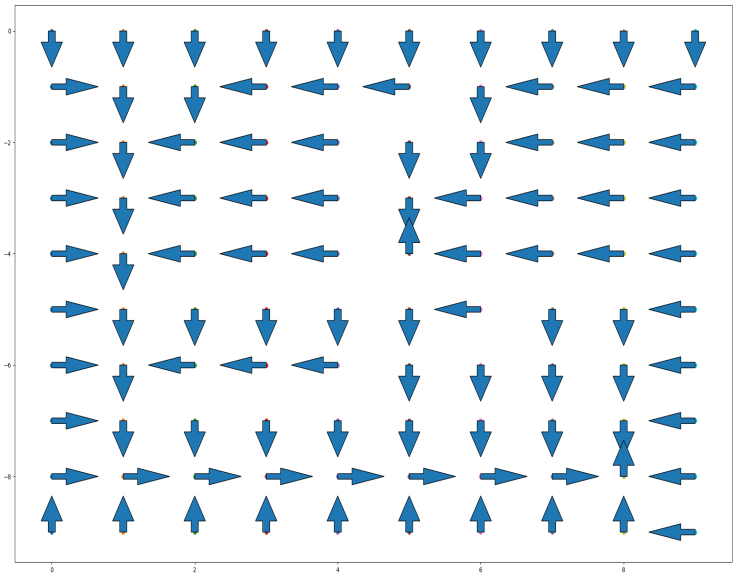
1. (a) Plot of the environment is as follows:



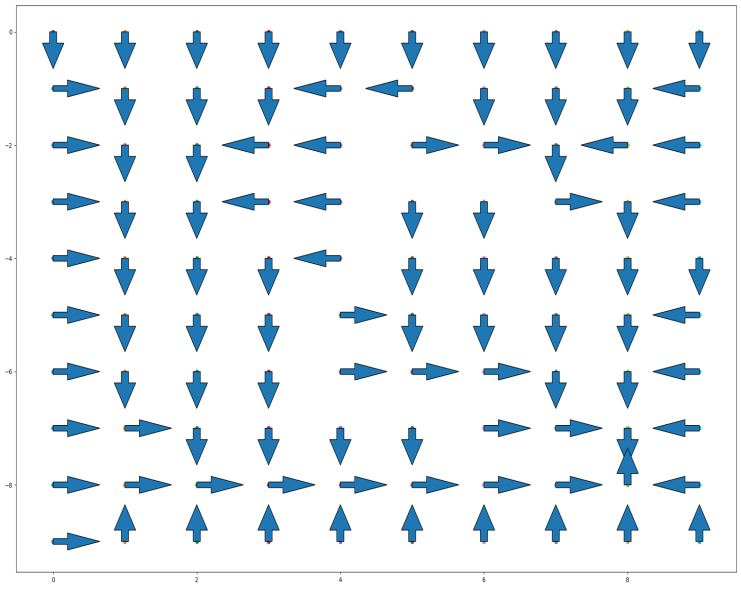
(b) Heatmap of the first policy evaluation is as follows:



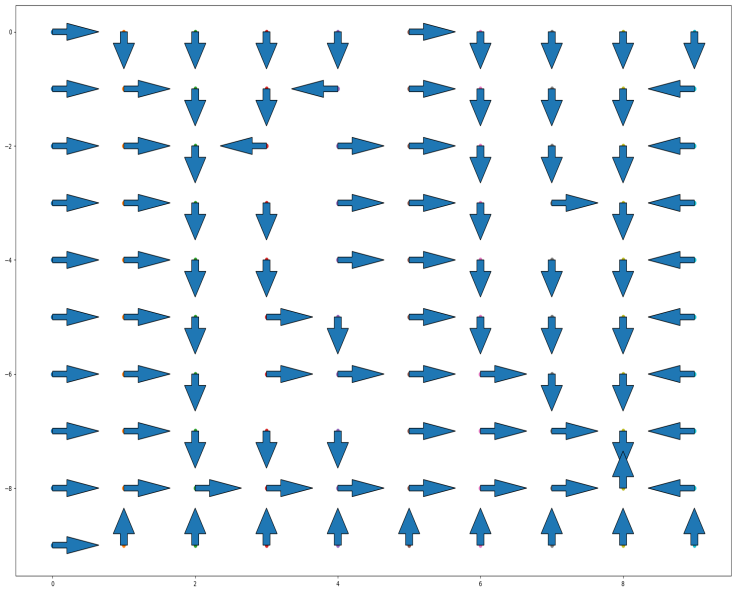
(c) The arrow graphs of the 4 iterations and the final heatmap is as follows:



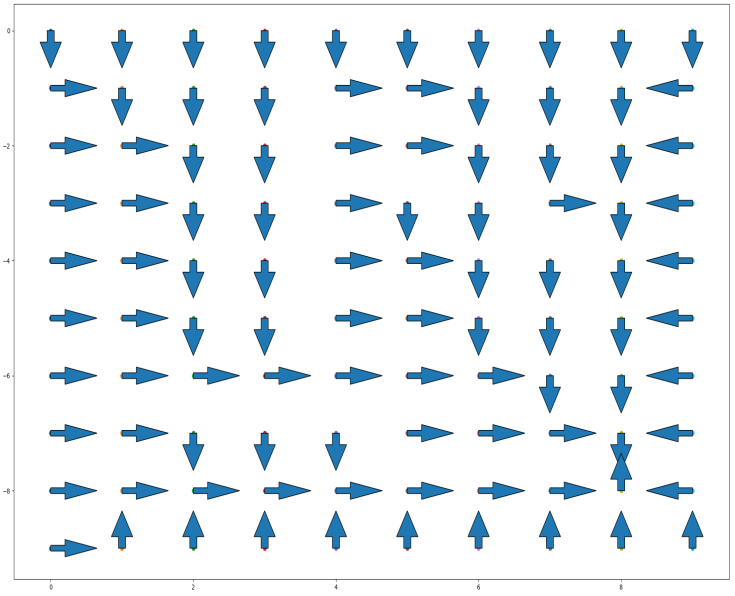
.



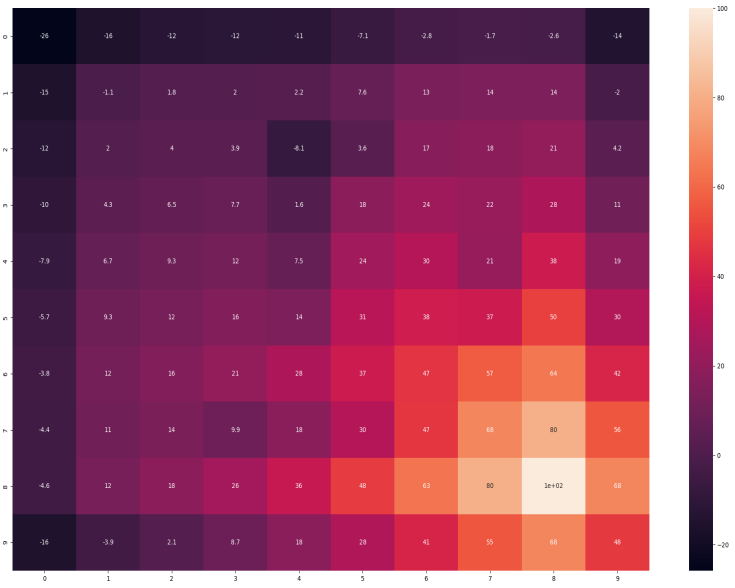
.


0
−2
−4
−6
−8
0
2
4
6
8

.


0
−2
−4
−6
−8
0
2
4
6
8

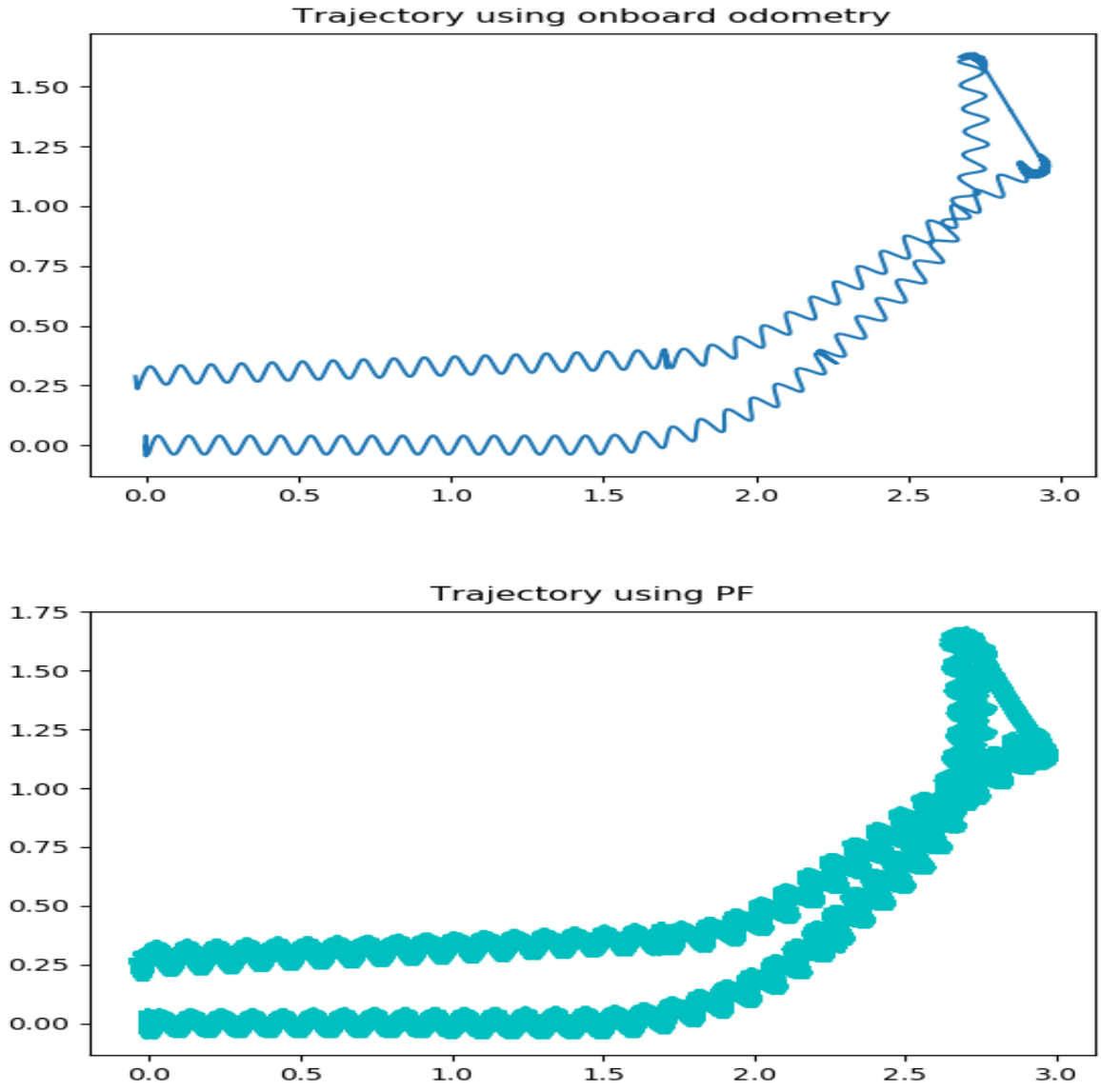
.




```
divyanshu@divyanshu-Legion-5-Pro-16ITH6:~/ese650/HW3$ python3 q1.py
[['S' 'S' 'S' 'S' 'S' 'S' 'S' 'S' 'S' 'S']
 ['E' 'S' 'S' 'S' 'E' 'E' 'S' 'S' 'S' 'W']
 ['E' 'E' 'S' 'S' 'E' 'E' 'S' 'S' 'S' 'W']
 ['E' 'E' 'S' 'S' 'E' 'S' 'S' 'E' 'S' 'W']
 ['E' 'E' 'S' 'S' 'E' 'E' 'S' 'S' 'S' 'W']
 ['E' 'E' 'S' 'S' 'E' 'E' 'S' 'S' 'S' 'W']
 ['E' 'E' 'E' 'E' 'E' 'E' 'E' 'S' 'S' 'W']
 ['E' 'E' 'S' 'S' 'S' 'E' 'E' 'E' 'S' 'W']
 ['E' 'E' 'E' 'E' 'E' 'E' 'E' 'E' 'N' 'W']
 ['E' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N']]
divyanshu@divyanshu-Legion-5-Pro-16ITH6:~/ese650/HW3$
```

2. (a) Given the current timestep, I first calculate the controls from the get-controls function using the lidar data from the current and previous timestep. As stated in the comments, the controls is simply the difference between these the values of x,y,th at these time steps. Hence, I use the smart-minus-2d function to calculate the difference.

   Now that I have the controls, I use it for each particle in the particle filter and add the controls. After this, I add the necessary noise using the np.random.multivariate-normal function. I use the smart-plus-2d function to add both the noise and the controls. The final trajectory plots of the odometry and the particle filter trajectory is as follows



(b) Given the particle in the particle filter and the time step, I first calculate the nearest time step of the joint data from the lidar data using the find-joint-t-idx-from-lidar function. Using this timestep, I calculate the head and the neck angle at that timestep. I use the current timestep to get the distance array of the lidar sensor.

For each particle I first convert the lidar distances to world coordinates assuming that the robot is at the given particle location. Now that I got the lidar world coordinates for each particle, I convert the world coordinartes into the grid coordinates by implementing the grid-cell-from-xy function.

Initially when the first observation comes in, a single particle is used to initialise the map. Hence for that single particle, I first update my map.cells and map.log-odds using

the gird coordinates that I got from that single particle. After this, for each particle, I use the map.cells from the preceding iteration to calculate observation log odds which is simply the scan matching result of the current lidar grid coordinates with the previous map.cells. The higher the match, the higher the log odds and hence higher the weight of that particle.

After calculating the observation log-odds, I use the update-weights to update the weights of the particles. Using this updates weights, I take the argmax to get the particle with the highest weight and update the map log odds and the map cells using the lidar grid cell coordinates of the particle with the highest weight. It is important to note that if I decrease the log probability of the obstacle everywhere, the drift could become significant and the trajectory would end up going through the obstacle itself. This is because, the current lidar observation shows the current obstacles and using this to decrease the log probability everywhere else would render the old obstacles to be declared as free space. To avoid this, after calculating the particle with the highest weight, I only declare the space free between the grid location of the particle and the obstacles that the particle encountered considering the robot to be at that particle location. This helps to keep an accurate track of the old obstacles as well and helps the map to be updated correctly overtime.

Lastly, I use the resample-particle function to resample the particle and get the updates poses and weights of the particle. This function uses the stratified-resampling function which I implemented using the resampling algorithm given in the notes. The generated graphs for the slam results is as follows

Using all these features, I use the implemented the slam algorithm to iteratively update the dynamics and observation. It is also necessary to note that I have increased the yaw noise because this helps me get more diverse particles and hence there is a better probability of me matching the current scan with one of the 100 particles and hence get a better estimate of my current pose.

To plot, I keep a track of all the free spaces by making a copy of the original grid and only updating the free spaces in that map. For checking the obstacles, I use the log odds map and rest I declare as unexplored spaces. The figure below shows the result. Green is the odometry, red is the filtered trajectory, black are the obstacles detected, white is the free space and gray is the unexplored space.

```
divyanshu@divyanshu-Legion-5-Pro-16ITH6:~/ese650/HW3$ python3 main.py --m
e observation
INFO:root:> Reading data
INFO:root:> Particles
: [[-4.64670052e-10]
 [ 4.05781452e-04]
 [ 0.00000000e+00]]
INFO:root:> Weights: [1.]
INFO:root:

INFO:root:> Particles
: [[2.  0.2 3. ]
 [2.  0.4 5. ]
 [2.7 0.1 4. ]]
INFO:root:> Weights: [5.24288566e-22 1.00000000e+00 5.24288566e-22]
divyanshu@divyanshu-Legion-5-Pro-16ITH6:~/ese650/HW3$
```

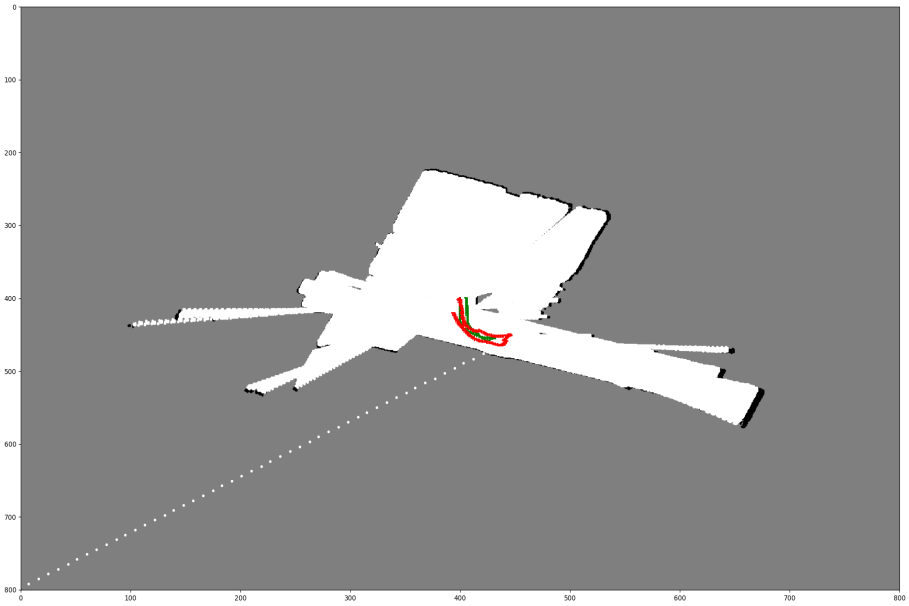


Figure 1: Dataset 1

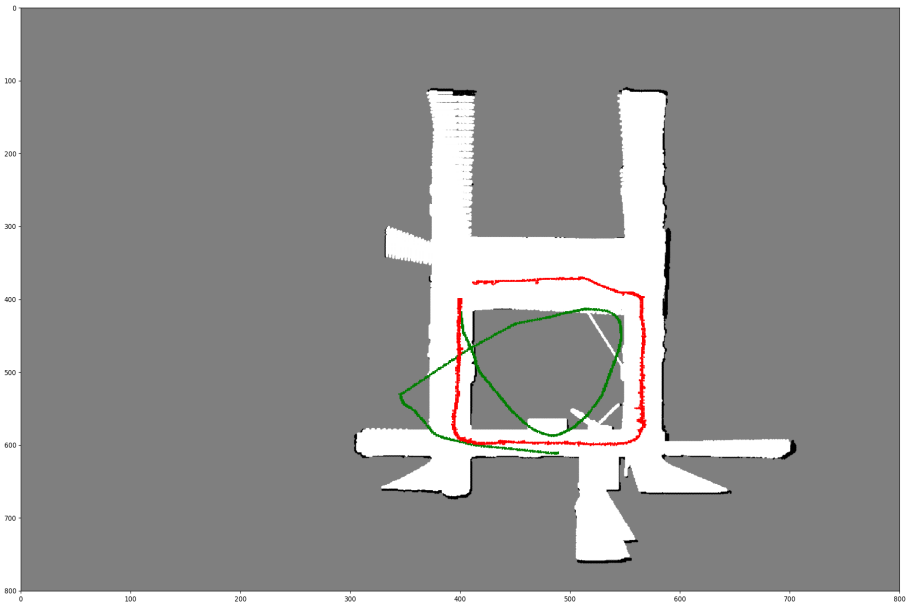


Figure 2: Dataset 2

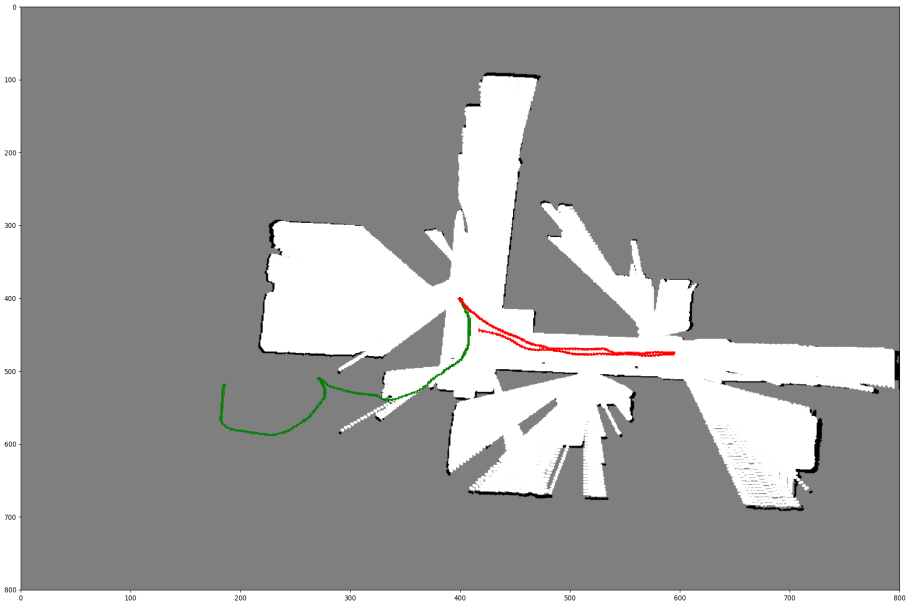


Figure 3: Dataset 3

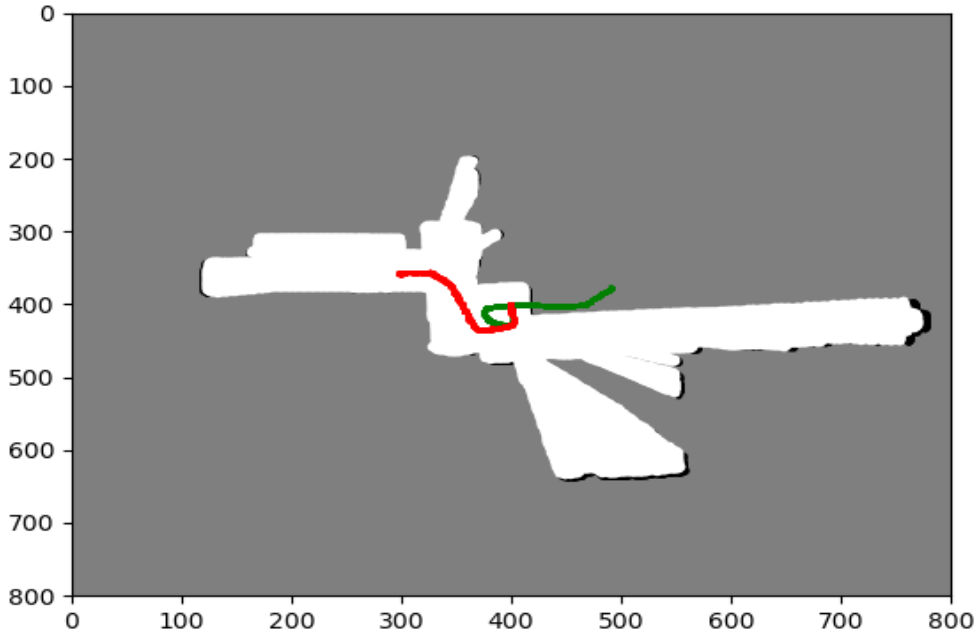


Figure 4: Dataset 4