Team 7 ROB 550 BotLab Report

Prototype Implementations of SLAM and A* for a Wheeled Robot

Elizabeth Miranda, Sahib Dhanjal, Zaid Ashai

***Abstract*—Using a stable two-wheeled robot and lidar, we implemented odometry, motion control, SLAM, A* path planning, and Yamauchi's Frontier Exploration algorithm with the goal of exploring an unknown area. We analyzed our odometry accuracy, compared mapping results to log files, compared localization results to Optitrack ground truth data, and analyzed efficiency of A* and exploration.**

I. Introduction

CONFIDENCE in a robot's location is fundamental to many robotics applications. In order to demonstrate a set of techniques that can be applied to localization, we implemented localization, mapping, and exploration algorithms on a two-wheeled differential drive robot equipped with a 2D Scanse Sweep Lidar, Raspberry Pi 3, Beaglebone Green, and Mobile Robotics Cape.

Our prototype implementations of seven tasks are elaborated on in the report: 1) motor control and basic movement, 2) navigation using dead reckoning based on odometry, 3) map creation based on lidar data, 4) Monte Carlo localization consisting of an action model, sensor model, and particle filter, 5) integration of (3) and (4) into a simultaneous localization and mapping (SLAM) algorithm, 6) path planning using the A* algorithm, and 7) autonomous exploration of unknown portions of a map.

We evaluated the performance of our system by comparing our dead reckoning estimated pose to the ground truth pose while driving in a square, generating a map using a known LCM log file, generating particle filter distributions at various points while driving in a square, comparing SLAM estimated poses to ground truth poses while driving a specified circuit, and comparing a path planned with A* to the actual path executed.

II. Methodology

The main portions of the system are the odometry, motion controller, SLAM, and exploration/planning.

A. Odometry

To convert encoder ticks to distance in meters (line 59 in `mb_odometry.c`), we used the following equation: $D = \frac{c*d*\pi}{e*r}$, where c = number of encoder ticks, d = 0.08m wheel diameter, e = 48 ticks per shaft revolution, and r = 34 shaft revolutions per wheel revolution. The encoder resolution, e, and the gear ratio, r, were determined from the given specification. We tested our odometry by placing a meter stick on the floor, moving the robot from one end of the stick to the other with the motors powered off, and recording the distance calculated through encoder values. We divided the actual distance by our calculated distance to determine the ratio by which to scale our calculated distance. We speculated that the cause for this discrepancy stems from an offset in the axes of the wheel and the motor shaft, which in turn stems from deformities in the wheel adapter.

To update the heading angle, $\Delta\theta_{odo}$, from encoder values (line 69 in `mb_odometry.c`), we used the following equation: $\Delta\theta_{odo} = \frac{D_r D_l}{B}$, where D_r was the distance traveled by the right wheel, D_l was the distance traveled by the left wheel, and B was the baseline, which we 0.21m from the specifications. To update the heading angle, $\Delta\theta_{gyro}$, from the gyroscope (line 70 in `mb_odometry.c`), we subtracted the current heading reading from the previous heading. We added the current heading by $\Delta\theta_{gyro}$ if $|\Delta\theta_{gyro}\Delta\theta_{odo}| > \Delta\theta_{thresh}$ or by $\Delta\theta_{odo}$ otherwise (lines 73 to 80 in `mb_odometry.c`). After examining raw gyroscope heading data, we determined that a $\Delta\theta_{thresh}$ of 0.00001 eliminated noisy gyroscope readings the best.

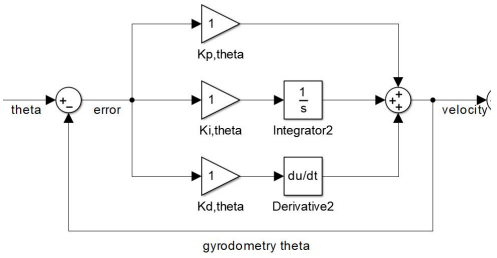


Fig. 1: PID diagram for controlling heading angle.

TABLE I: Table of PID parameters for heading θ. Included are the canonical P, I, and D gains, linear bias B, and constant bias C.

Variable	P	I	D	B	C
θ	0.3	0.0	0.3	0.0	0.0

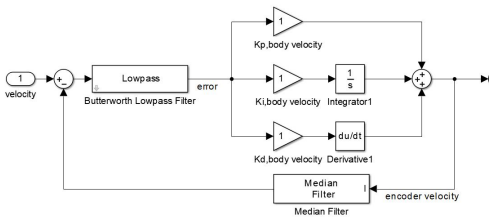


Fig. 2: PID diagram for controlling body velocity.

TABLE II: Table of PID parameters for body velocity v. Included are the canonical P, I, and D gains, linear bias B, constant bias C, median filter length n, and derivative 4th order low pass Butterworth filter cutoff frequency f in Hz (- denotes a filter is not used).

Variable	P	I	D	B	C	n	f
θ	4.5	0.3	1.5	0.0	0.0	16	6

B. Motion Controller

The PID loop for heading angle, θ, is shown in Figure 1 and the gains are shown in Table I. We tested the sensitivity of the gains by manually turning the robot 45deg and 90deg from the desired heading angle and observing the robot's response. When the gains were too large, the robot would overshoot the desired heading and continue oscillating around the desired heading or continue spinning forever. When the gains were too small, the robot would not reach the desired heading at all. Once the robot adjusted reasonably well to changing in heading at zero forward velocity, we tested its performance under nonzero forward velocity. We found that the robot would occasionally jiggle while moving, indicating that it was too sensitive to small changes in heading. Once we adjusted the gains down to remove the unwanted behavior, we were left with the gains found in Table I.

C. Simultaneous Localization and Mapping (SLAM)

We began by building a map of an area based on provided LIDAR data, then implemented Monte Carlo Localization using a provided map. Finally, we integrated these into a SLAM system.

1) Mapping: We implemented mapping by adjusting the log odds values of grid cells representing a 10m x 10m space around the robot; each square grid cell had a side length of 0.05m. The (x,y) origin of each LIDAR ray was adjusted using the movingLaserScan class found in `moving_laser_scan.*`. Next, rather than iterating through the entire grid, each ray was iterated along using Breshenham's Line algorithm [1] marking the endpoint cell as more likely occupied and the prior cells along the way as more likely vacant. Log odds greater than 0 represented a cell that is probabilistically more likely occupied and log odds less than 0 represented a cell that is more likely vacant, with 0 representing unexplored space.

2) Monte Carlo Localization: Monte Carlo Localization (*MCL*) is a particle-filter based localization algorithm which represents a robots belief as a set of weighted hypotheses, which in turn approximates the posterior under a common Bayesian formulation of the localization problem. Essentially, it is a recursive Bayes Filter derived from the *Markov Family* [2] of algorithms for localization.

Given the previous state of the robot(X_{t-1}), the control input(u_t), and the sensor measurements(z_t), MCL estimates the next state of the robot(X_t) as shown in Algorithm 1. For the case of a differential drive robot *(as in our case)*, X_t is given by the position and orientation of the robot (ie. $< x, y, \theta >$), and we use this notation through-out.

MCL relies on three key components: Measurement Model, Sensor Model, and a Particle Filter.

Algorithm 1 Monte Carlo Localization

1: **function** MCL(X_{t-1}, u_t, z_t)
2: $\widetilde{X_t} = X_t = \emptyset$
3: **for** m = 1 to M **do**
4: $x_t^{[m]}$ = action_model (u_t, $x_{t-1}^{[m]}$)
5: $w_t^{[m]}$ = sensor_model (z_t, $x_t^{[m]}$)
6: $\widetilde{X_t} = \widetilde{X_t} + < x_t^{[m]}, w_t^{[m]} >$
7: **for** m = 1 to M **do**
8: Draw $x_t^{[m]}$ from $\widetilde{X_t}$ with probability $\propto w_t^{[m]}$
9: $X_t = X_t + x_t^{[m]}$
10: return X_t

a) Measurement Model: The Measurement Model (also known as *Action Model*) predicts the next state of the robot based on the control input and the previous state, given by *p($x_t|u_t, x_{t-1}$)*. Two of the popular models used for differential drive robots are: (i) Velocity-based Model, and (ii) Odometry-based Model.

The Odometry Model is used in systems equipped with wheel encoders, whereas the Velocity Model is used in systems without wheel encoders and considered less accurate based on past experiments[3][4].

We used the Odometry Model *(Appendix F :Algorithm I)* since our robot was equipped with wheel encoders. Assuming the robot moves from state $x' :< x', y', \theta' >$ to state $x :< x, y, \theta >$, $u_t =< x, x' >$, represented the true state of the robot, and $x_{t-1}^{[m]} =< \bar{x}, \bar{x}' >$ represented the odometry values. The true state was the actual position of the robot based on MCL, whereas odometry values were the position of the robot calculated from the wheel encoders. The odometry values were represented as $\bar{x} :< \bar{x}, \bar{y}, \bar{\theta} >$ & $\bar{x}' :< \bar{x}', \bar{y}', \bar{\theta}' >$.

The approach to choosing the values of the arbitrary constants (α_{1-4}) is discussed in the results section.

Algorithm 2 Odometry-Based Action Model

function `action_model`(u_t, $x_{t-1}^{[m]}$)
  $\delta_x = \bar{x}' - \bar{x}$
  $\delta_y = \bar{y}' - \bar{y}$
  $\delta_{tr} = \sqrt{\delta_x^2 + \delta_y^2}$
  $\delta_{r1} = atan2(\delta_y, \delta_x)$ - $\bar{\theta}$
  $\delta_{r2} = \bar{\theta}'$ - $\bar{\theta}$ - δ_{r1}
  $\hat{\delta}_{tr} = \delta_{tr}$ + `normal(0,` $\alpha_3\delta_{tr}+\alpha_4(|\delta_{r1}|+|\delta_{r2}|)$ `)`
  $\hat{\delta}_{r1} = \delta_{r1}$ + `normal(0,` $\alpha_1|\delta_{r1}| + \alpha_2\delta_{tr}$ `)`
  $\hat{\delta}_{r2} = \hat{\delta}_{r2}$+ `normal(0,` $\alpha_1|\delta_{r2}| + \alpha_2\delta_{tr}$ `)`
  $x' = x + \hat{\delta}_{tr}\cos(\theta + \hat{\delta}_{r1})$
  $y' = y + \hat{\delta}_{tr}\sin(\theta + \hat{\delta}_{r1})$
  $\theta' = \theta + \hat{\delta}_{r1} + \hat{\delta}_{r2}$
  return $< x', y', \theta' >$

function `normal`(μ, σ)
  return $\exp(\frac{-\mu^2}{2\sigma^2})/\sqrt{2\pi\sigma^2}$

b) Sensor Model: The Sensor model assigned a weight to each of the particles created by the measurement model. The weight depended on the likelihood of the particle given the measurements from the sensor. The method used to compute the weight of each of the particles is described in *(Appendix F :Algorithm I*.

When sensing its environment, the robot updated the particles to reflect where it can be. It computed the probability of a sensor reading given the state of the particle and assigned a weight to the particle based on the probability. Particles consistent with the sensor readings were more likely to be chosen and were alloted a higher weight, whereas the inconsistent ones were rarely chosen. Hence, with subsequent sensor readings, the robot tended to converge towards a better estimate of the robot's state making the robot increasingly confident of its position.

c) Particle Filter: The particle filter represented the distribution of likely states *(position & pose of the robot)*, with each particle representing a possible state. The algorithm started with a uniform random distribution of particles over the configuration space, meaning the robot has no information on its position and assumes it's equally likely to be at any point in space.

As the robot moved, the particles shifted to predict its new state after the movement. Upon sensing a change, the particles were re-sampled based on recursive Bayesian estimation *(non-parametric form of Bayes Filter(Appendix G :Algorithm IV)*. After several iterations, the particles converged to the actual location of the robot.

Algorithm III in *Appendix F* describes the methods we implemented to represent the posterior $bel(x_t)$ by a set of random state samples drawn from this posterior. After every iteration of the particle filter, we re-sampled the particle set based on the weights assigned to each of the particles in the first step. By doing this, we eliminated the unlikely states and densified the more likely ones.

3) Combined Implementation: Upon completion of mapping and localization, we integrated the functionality together in order to run SLAM.

D. Planning and Exploration

A* path planning and a frontier exploration method were implemented in order to plan paths and explore new environments.

1) Path Planning: We implemented an A* path planner in order to plan navigation between specified locations in an environment based on [5]. Source code can be found in `astar.*` and `obstacle_distance_grid.*`. Our methodology used a heuristic of Euclidean distance between the start and goal locations. Rather than implementing an obstacle_distance_grid of true distances to obstacles, we filled the grid with three values indicating if the cell is: 1) an obstacle itself, 2) within a robot radius of an obstacle, or 3) not within a robot radius of an obstacle.

2) Exploration: We implemented *Yamauchi's frontier exploration algorithm* [6]. The algorithm initially set the value for all the cells in the occupancy grid equal to the initial (prior) probability. The value of the prior was set to 0.5 for all the experiments. Each grid cell was then placed into one of the three classes:

- **open**: $p_{occupancy} < prior$
- **unknown**: $p_{occupancy} < prior$
- **occupied**: $p_{occupancy} > prior$

Three lists were maintained for the frontiers throughout the execution of the algorithm: 1) open_frontier_list, 2) inaccessible_frontier_list, and the 3) occupied_frontier_list. The open_list maintained the unexplored frontier cells, the occupied_list maintained the visited frontier cells, and the inaccessible_list maintained the inaccessible cells.

Once initialization was complete, a 360deg sensor sweep was performed, using information from which we extracted the boundary between the open and unknown spaces. Any open cell adjacent to an unknown cell was labeled as a frontier edge cell and added to the open_list. Adjacent edge cells were grouped into frontier regions and regions with a size of the robot's width were considered a frontier.

The A* path planning algorithm was chosen for navigation in the exploration case. We chose this planner because we assumed a static environment in our exploration case with no new obstacles. Once the frontiers have been detected, the robot navigated to the nearest accessible unvisited frontier using the path planner. This frontier cell was then added to the occupied_list. The robot performed a sensor sweep again to update the evidence grid with the new occupancy probabilities. With

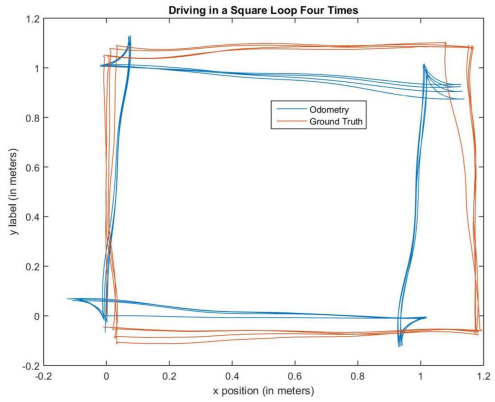


Fig. 3: A visual graph of the robot driving in a 1m by 1m square four times. Odometry data is shown in blue and ground truth data from Optitrack is shown in red.

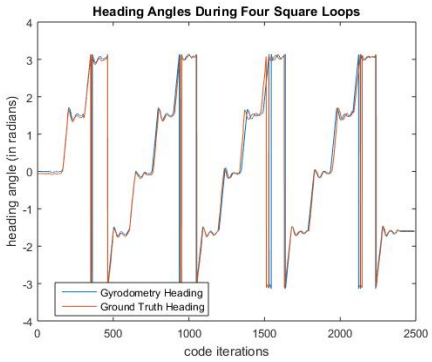


Fig. 4: A graph of the heading values when driving in a square four times. Odometry data is shown in blue and ground truth data from Optitrack is shown in red.

the newly added frontier cells, the robot navigated to the closest frontier cell in the open_list. This process was repeated until no more unexplored frontier cells were left *(ie. the open_list was empty)*.

While executing the exploration algorithm, the robot might find unreachable frontier cells. If the robot determined this destination cell to be inaccessible, the location was added to the list of inaccessible frontiers and the robot performed a sensor sweep from its current location. Instances where multiple frontier cells were at the same distance from the robot also occurred. In such cases, we chose the frontier cell which required the robot to change its heading the least amount.

III. Results

The results of our implementations of the odometry, motion controller, SLAM, and exploration/planning are described below.

A. Odometry

To validate the accuracy of our gyro fused odometry, we implemented RTR control to autonomously drive the robot in a square path four times (as shown in Figure 3). When using the motion capture data as the ground truth, we found the average positional error was 0.15 meters with a standard deviation of 0.09 and the average heading error was 0.23 radians with a standard deviation of 0.27.

B. Motion Controller

We tested our motion controller by physically disturbing the robot's heading and observing its correction behavior and by commanding various velocities from 0 to 1 m/s. We initially observed unusual oscillations in our forward velocity PID which led us to use the motion controller provided by the professor. We used that motion controller for the rest of the lab and observed no unusual behaviors. We changed the gain constant variables in motion controller to reflect those we tuned in our heading PID. We quantified our results in the graphs shown in Figure 3 and Figure 4 and shown in error graphs in the Appendix.

C. Simultaneous Localization and Mapping (SLAM)

We successfully implemented both localization and mapping but due to time constraints performed limited testing of SLAM as a system.

1) Mapping: Increments used to increase and decrease the value of each cell were tuned based on qualitative comparison of each generated map to provided sample map; the values chosen were 5 and 2 for kHitOdds and kMissOdds, respectively. The resultant map built from the "sweep_obstacle_slam_10mx10m_5cm.log" LCM log is shown in 5; additional generated maps with different characteristics can be found in the Appendix.

2) Monte Carlo Localization: The performance of each of the sub-parts of MCL are briefly stated below.

a) Measurement Model: The measurement model was responsible for producing a sparse distribution of states given the previous state and control input. The constants α_1, α_2, α_3 and α_4 used in the equations played a major part in the measurement model and were tuned by trial and error. As seen in the equations, the constants α_1 and α_4 were converted to angles (in radians) whereas α_2 and α_3 were converted to distances.

TABLE III: Table of values for α

α_1	α_2	α_3	α_4
0.05	0.3	0.3	0.05

Since the changes in the angles were smaller than the ones in distances, we started with a value of 0.01 for α_1

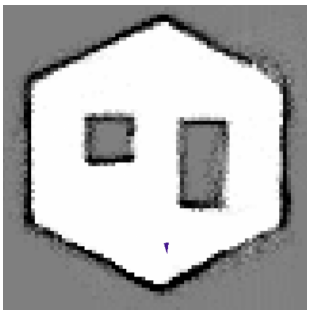


Fig. 5: The occupancy grid map generated from sweep_obstacle_slam_10mx10m_5cm.log using kHitOdds of 5 and kMissOdds of 2; white represents a likely vacant space, black represents likely occupied space, and gray represents unexplored space.

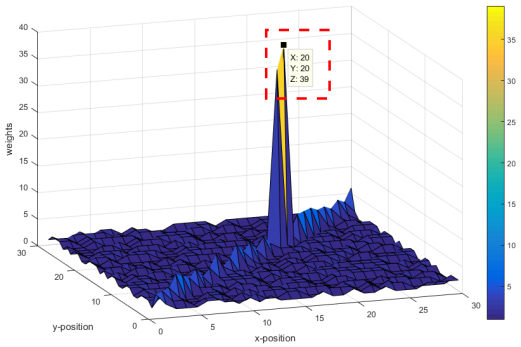


Fig. 6: Sensor Model weights over a 3m Lidar ray: the maximum weight is 39 in accordance with our algorithm

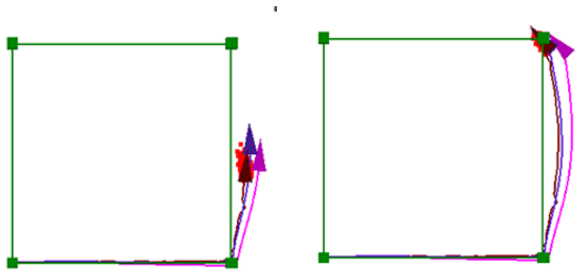
and α_4, and 0.1 for α_2 and α_3. Based on the observed distribution of the particles created, we doubled each of the values until all the particles of the distribution converged to a circle with radius $1cm$ on the screen. Once achieved, we fine-tuned the values to achieve a distribution within a circular area with diameter $1cm$ on the screen. Figures *5,6,7* in Appendix C show the different distributions we obtained in the process of tuning the values. The final values are listed in Table III.

b) Sensor Model: The sensor model performed well given the accuracy of MCL, taking into consideration the inaccuracies in the odometry. The accuracy for this model depended on the choice of the threshold. The threshold *($odds_{threshold}$)* was chosen based on the accuracy desired for the robot positioning (0.9). Too large a threshold (> 0.98) resulted in the robot having low confidence in any of the particles generated. Depending on the number of particles generated, there might be cases where no particle generated would lie within the thresholded region, which resulted in all particles having the same low probability. This resulted in the robot having low confidence in any of the sampled states. A lower threshold (< 0.7) resulted in too wide a region being accepted. Hence many particles would have higher weights causing the robot to have high confidence in multiple states, which in turn resulted in lower accuracy while localizing. The distribution of weights based on our sensor model is represented in Fig.*6*.

c) Particle Filter: The time taken for the algorithm to localize from an evenly distributed sample space to the position of the robot varied with the number of particles sampled. Table IV represents the times taken for some scenarios. Given the 300 particles, the filter had an acceptable performance and was able to localize the robot within 137 seconds on average. As the number of particles increased, the time for the filter to update also increased.

$n_{particles}$	$time(ms)$
100	75.44 ± 15.05
300	136.89 ± 13.76
500	292.89 ± 38.44
1000	405.44 ± 106.48

TABLE IV: Time taken to update particle filter for 100, 200, 500 and 1000 particles



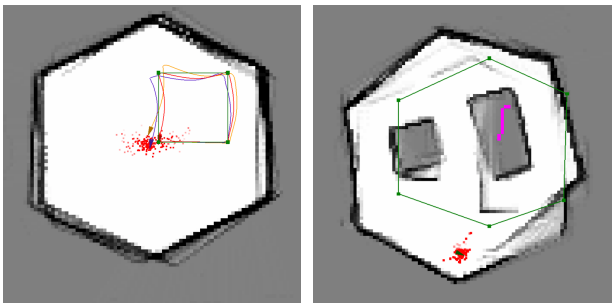
(i) (ii)

Fig. 7: Particles generated by the particle filter on the (i) center of one of the edges, and (ii)corner of the square. Particles for all the other positions are in *Appendix C*

An issue we faced was repeated re-sampling when the robot was stationary. This meant that when the robot was in two similar parts of a map *(parts with similar features as sensed by the robot)*, it tended to eventually converge to particles in just one of the two parts after repeated re-sampling. Because of this, particles in the other part weren't considered as a state until the robot moved. To fix this temporarily, we increased the number of particles from 300 to 1000. Adding more particles increased the number of re-samples converging to a single state.

3) Combined Implementation: Our combined SLAM implementation requires further tuning, however, our

current system determined the robot's location and maps the surrounding space relatively well as shown in 8. Due to time constraints ground truth pose error data was not collected and analyzed.



(i) (ii)

Fig. 8: Results of SLAM for driving in a square and around obstacles.

D. Planning and Exploration

We implemented an A* algorithm and successfully generated optimal paths around obstacles, taking into account our robot's width. We published intermediate poses to the robot and tracked the ground truth of the robot's motion.

1) Path Planning: Our A* algorithm was tested by using provided test files and by observing paths planned around log-generated obstacles in the BotGUI. The path our A* code generated differed from the actual path driven by our robot as shown in 9. An example path of navigation around an obstacle can be found in the Appendix.

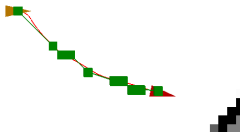
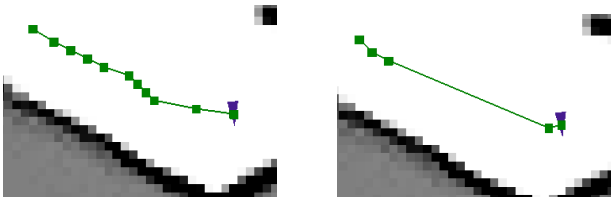


Fig. 9: A comparison between the A* generated navigation path and the actual path driven by our robot.



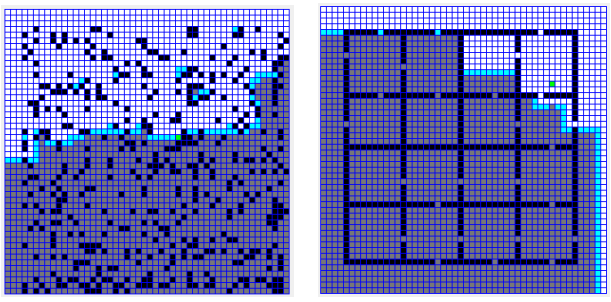
(i) (ii)

Fig. 10: Comparison of A* without and with diagonal pose algorithm.

Our A* algorithm returns a path consisting of a series of poses. Initially, a new pose was added for every cell location along the desired route that involved a change in both the x and y coordinates (i.e. not a vertical or horizontal line), resulting in diagonal lines consisting of many poses. In order to improve the algorithm and return a path consisting of fewer poses, we implemented a comparison between $cos(\theta)$ of a tentative start location of the prior cell and the tentative start location of the current cell. Our goal was to add a new pose to the path when the arc length between the prior line and the new line differed significantly. This implementation successfully resulted in paths consisting of fewer poses as shown in Figure 10.

Our path planning execution times for the test cases found in the `astar_test_files` binary can be found in the Appendix, where test cases were run ten times. Notable characteristics included mean execution times on the order of 30ms. The standard deviations of execution times for each test case were large and the maximum execution time for any test case was around 80ms.

2) Exploration: Terrains of various dimensions and obstacle distributions were used for simulating Yamauchi's Frontier Exploration algorithm. The completion time and number of unvisited cells, starting at random initial locations, were recorded for evaluation. We assumed the laser scan spanned a length of just one grid cell for the simulation, which in turn, was also equal to the size of the robot. This meant that the robot could sense just the grid cells adjacent to its location. Refer to *Appendix E* for the layouts of each map we used and the state of exploration after a certain number of iterations. Figure **??** shows the average completion times for some of the maps shown. The error-bars in the graph indicate standard deviation when each map was run 25 times with random initial locations for the robot.



(i) (ii)

Fig. 11: 500 iterations in the (i) Outdoor World and (ii) Office World (*50 x 50 grid*). Cell colors depicted are as follows: light blue - open cells, white - occupied cells, green - robot, navy blue - obstacles, and rest - unknown

As can be seen in the Appendix, approximately 1114

iterations on average were required to explore a 50 x 50 outdoor grid *Fig.(11)* completely. The exploration algorithm performed satisfactorily in simulation, since it visited all the grid cells in all cases. We tried implementing this algorithm on the robot but couldn't do so due to lack of time.

IV. Discussion

The odometry of our robot was accurate to approximately 15mm and the motion controller adequately controlled our low-speed use case. Our SLAM worked as a cohesive system, but could be greatly improved through the analysis of ground truth pose error data collection and implementation of loop closure. A* path planning worked both in simulation and on the physical robot but could be further tuned to improve accuracy and made more efficient. Exploration methods have been proposed and simulated but should be further tested using the physical robot in order to verify the results of our simulations.

References

[1] "Breshenham's line algorithm," https://en.wikipedia.org/wiki/Bresenhamslinealgorithm, accessed: 2017-12-13.

[2] D. Fox, *Markov Localization for Mobile Robots*, 1999, http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume11/fox99a-html/node1.html.

[3] W. Burgard, *Probabilistic Motion Models - Lec #06*, 2017, http://ais.informatik.uni-freiburg.de/teaching/ss17/robotics/.

[4] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2006. [Online]. Available: http://www.probabilistic-robotics.org/

[5] "Introduction to a*," https://www.redblobgames.com/pathfinding/a-star/introduction.html, accessed: 2017-12-13.

[6] B. Yamauchi, "A frontier-based approach for autonomous exploration," 1997. [Online]. Available: http://ieeexplore.ieee.org/document/613851/

[7] W. Burgard, S. Thrun, D. Fox, and F. Dellaert, "Monte carlo localization: Efficient position estimation for mobile robots," 1999. [Online]. Available: robots.stanford.edu/papers/fox.aaai99.pdf

[8] S. Koenig and M. Likhachev, *D* Lite Path Planner*, 2002, $http://robotics.usc.edu/\ geoff/cs599/D_Lite.pdf$.

ROB 550: BotLab Appendix

Elizabeth Miranda Sahib Dhanjal Zaid Ashai

December 20, 2017

Appendix A Odometry

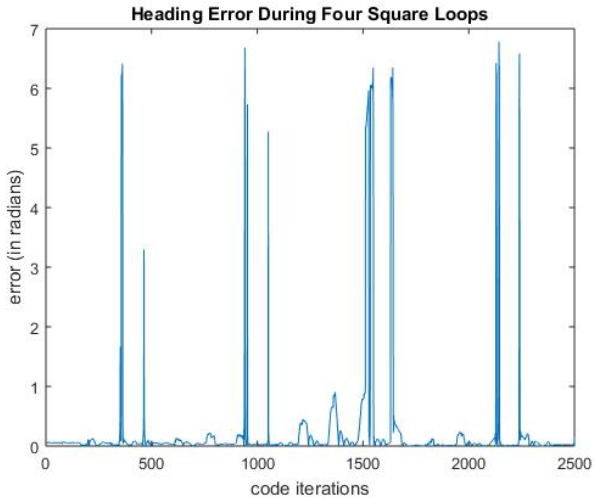


Figure 1: Heading error when driving in a square four times. Note that the spikes in error occur when the robot is at a heading of 2π radians and thus have no real significance.

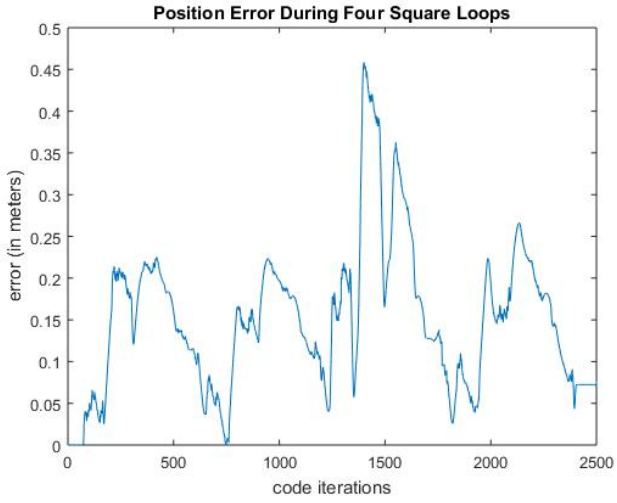


Figure 2: Position error when driving in a square four times.

Appendix B Mapping

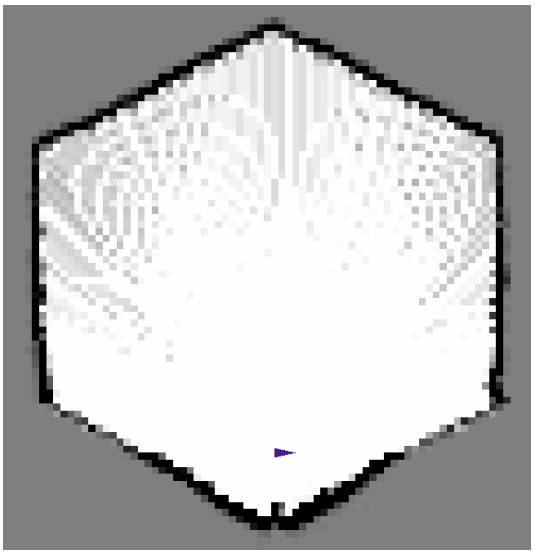


Figure 3: Results of our mapping algorithm run using an LCM log of a convex environment with a stationary, off-center robot with hit odds of 5 and miss odds of 2.

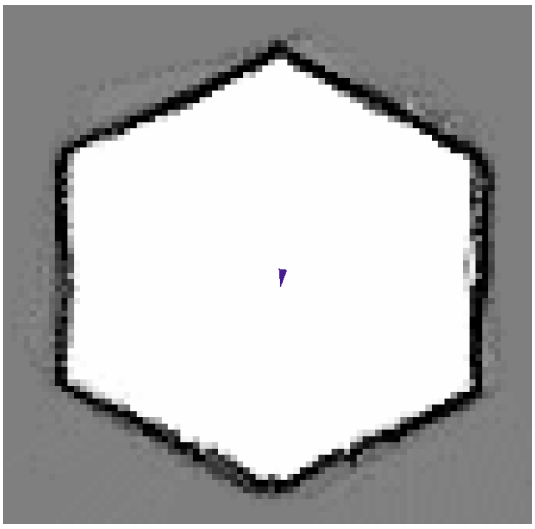


Figure 4: Results of our mapping algorithm run using an LCM log of the robot driving in a 1m x 1m square with hit odds of 5 and miss odds of 2.

Appendix C Localization

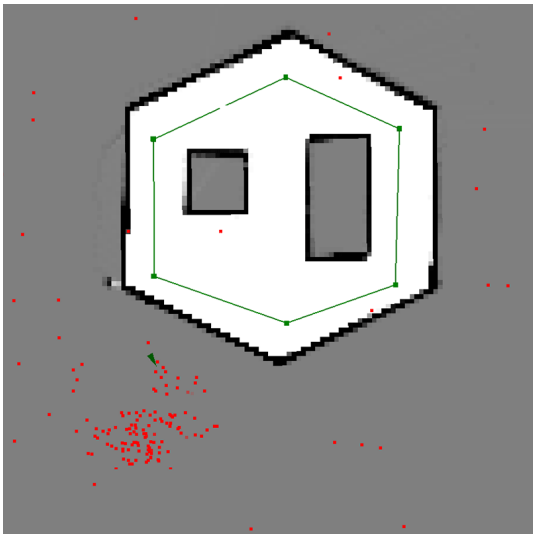


Figure 5: Tuning Action Model constants: $\alpha_1 = 0.01, \alpha_2 = 0.1, \alpha_3 = 0.1, \alpha_4 = 0.01$

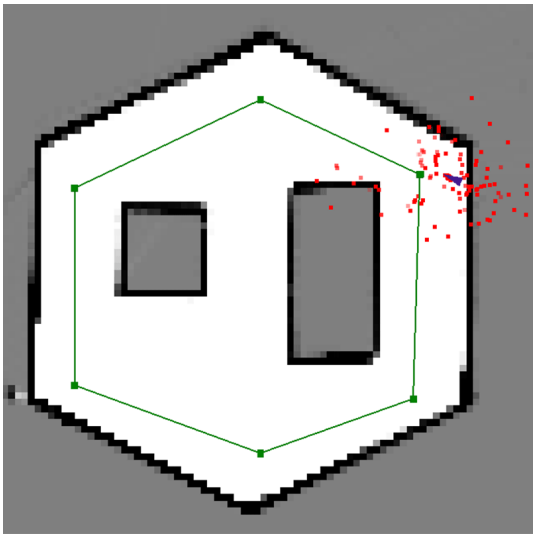


Figure 6: Tuning Action Model constants: $\alpha_1 = 0.04, \alpha_2 = 0.3, \alpha_3 = 0.3, \alpha_4 = 0.04$

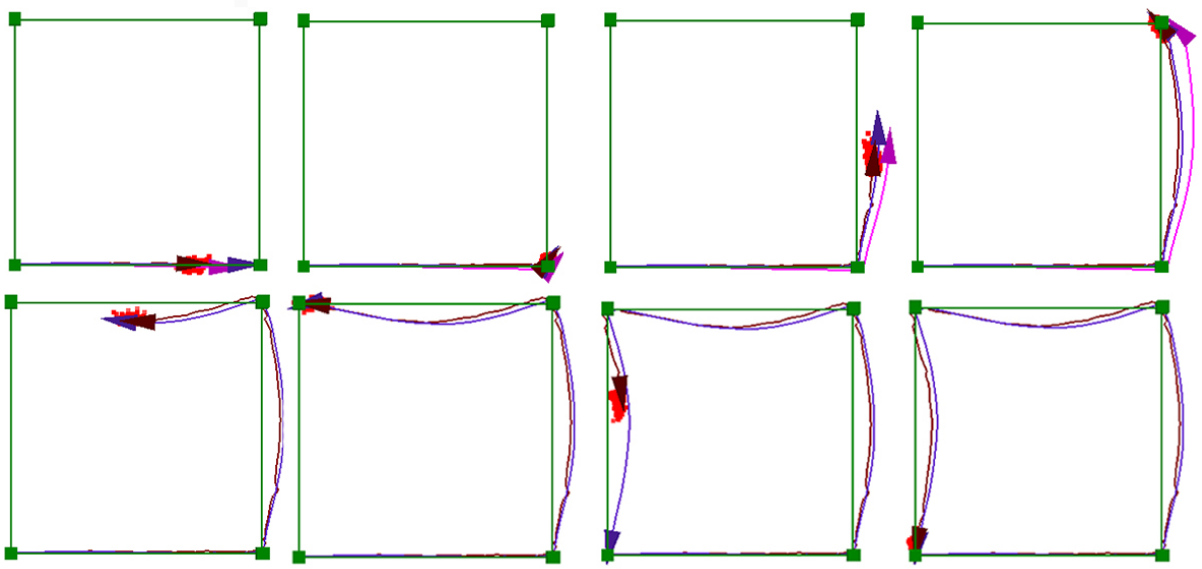
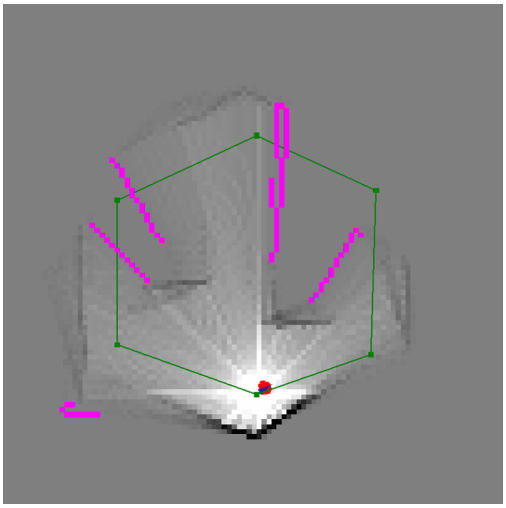
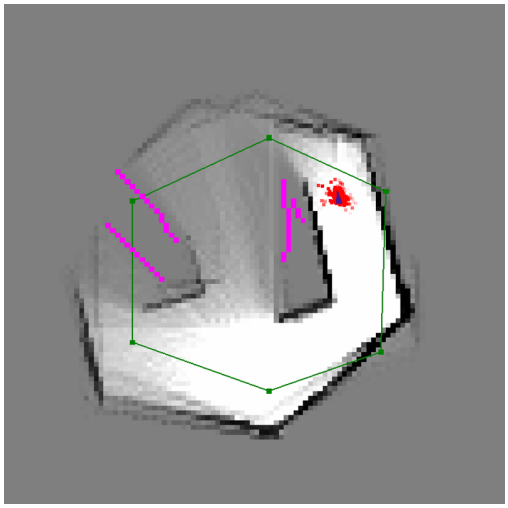


Figure 7: Tuned Action Model Performance on the drive_square log file

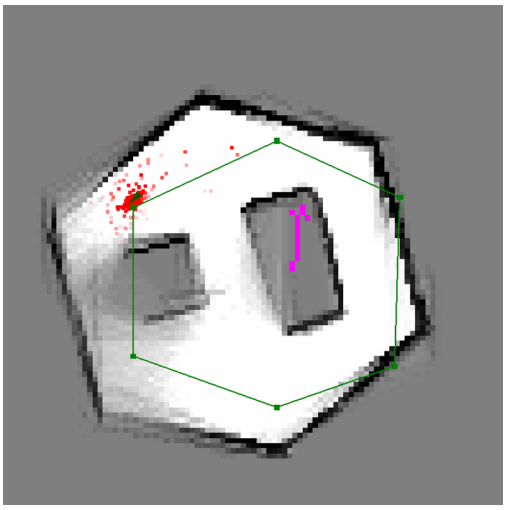
Appendix D SLAM



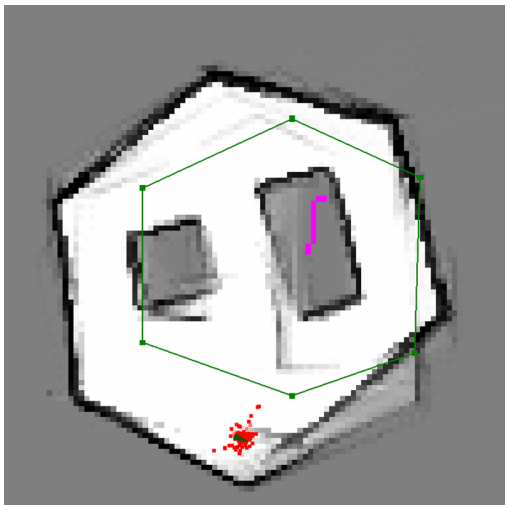
(i)



(ii)



(iii)



(iv)

Figure 8: Progressive maps made while running SLAM on the obstacle_map log file. The green lines represent the commanded positions.

Appendix E Path Planning

Table 1: A* Path Planning Execution Time Statistics (in milliseconds)

Test Case	Convex	Empty	Maze	Narrow	Wide
Min	1.9	5.0	2.2	4.6	5.1
Mean	2.1	6.7	2.7	25.1	25.7
Max	2.4	11.8	3.4	78.4	79.9
Median	2.0	5.2	2.6	8.9	9.3
Std dev	0.09	2.5	0.33	29.7	30.2

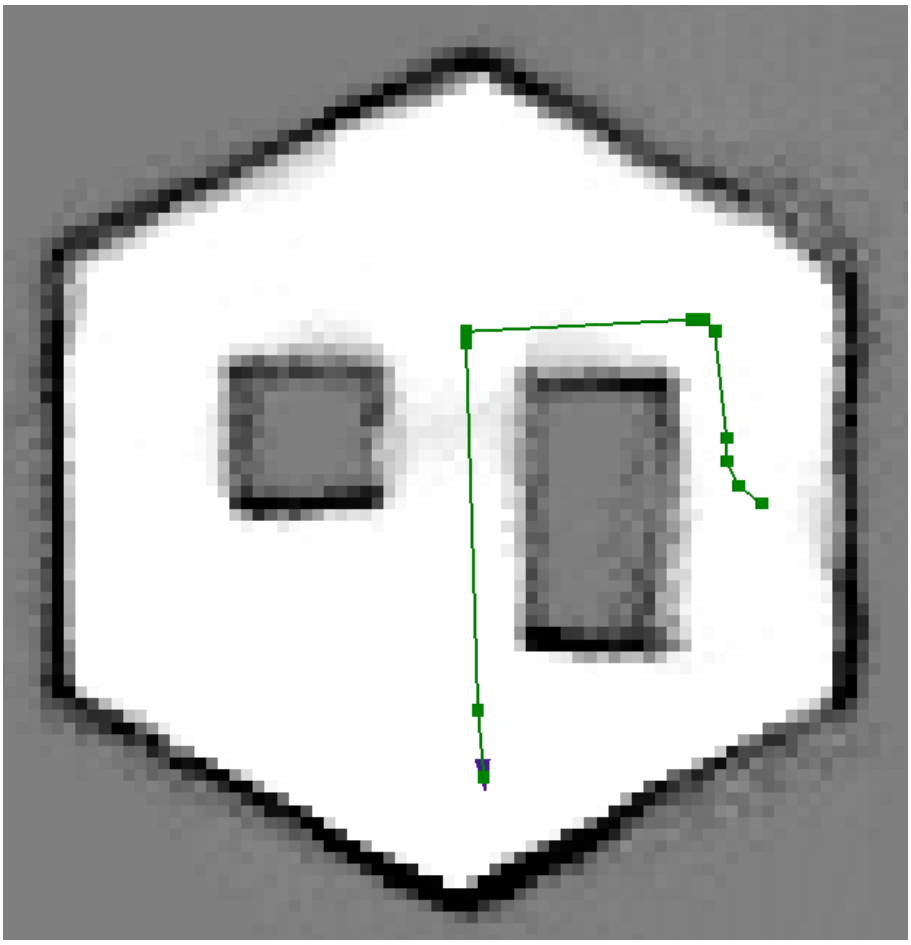
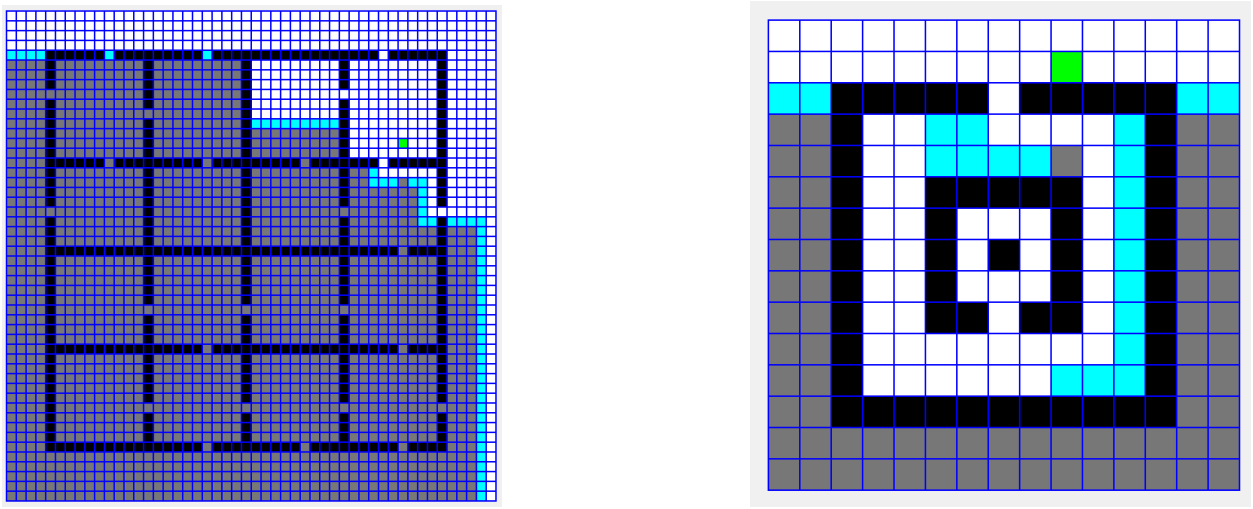


Figure 9: Example of A* algorithm planning a path around an obstacle, avoiding an opening deemed too narrow to pass through based on a specified robot radius and generating diagonal path segments.

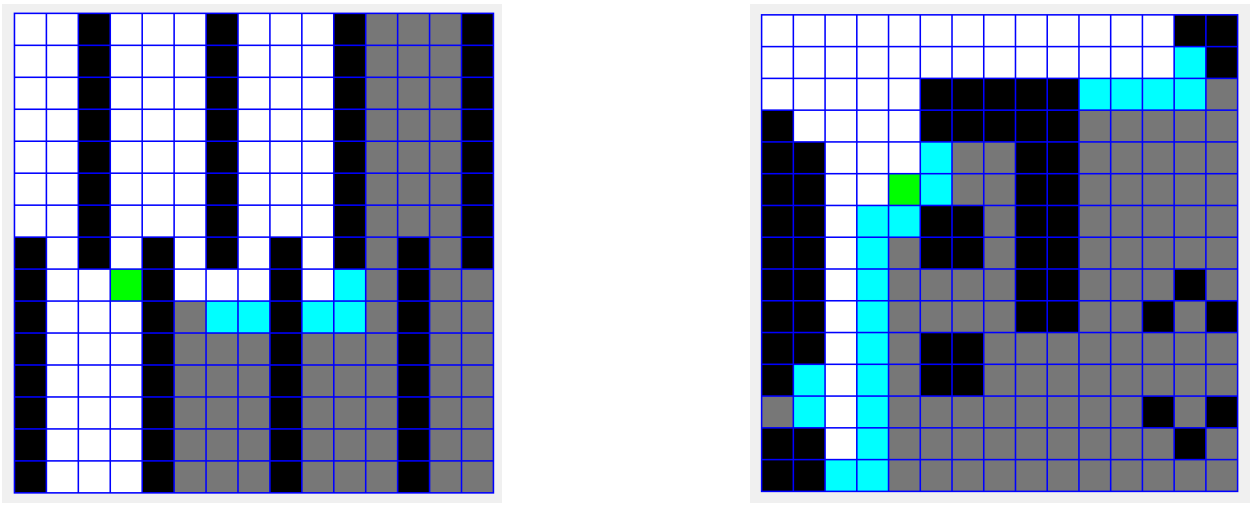
Appendix F Exploration

In all of the figures referenced below, the following color scheme is being used:
(i) white - occupied cells , (ii) green - robot current position, (iii) light blue - open cells, (iv) navy blue - obstacle cells, and (v) others - unoccupied/ unexplored cells



(i) (ii)

Figure 10: 500 iterations of Yamauchi's Algorithm for the simulated worlds: (i)Office World(50 x 50 grid) and, (ii) Circular Maze World (15 x 15 grid)



(i) (ii)

Figure 11: 75 iterations of Yamauchi's Algorithm for the simulated worlds: (i)Bar Maze World and, (ii) Living Room World, both of which are 15 x 15 grids

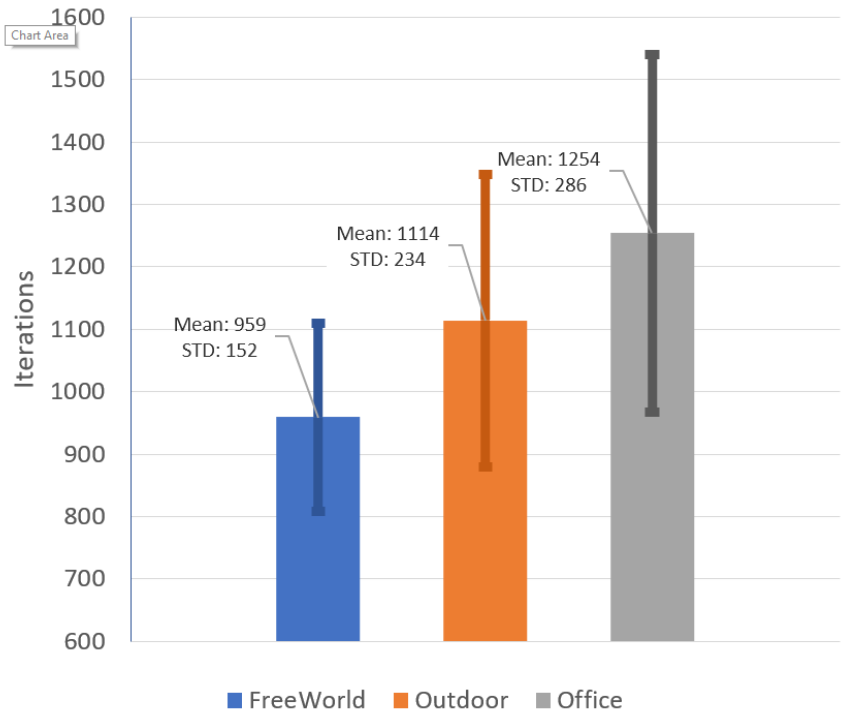


Figure 12: Performance of Yamauchi's exploration algorithm in Simulated 50 x 50 Grids

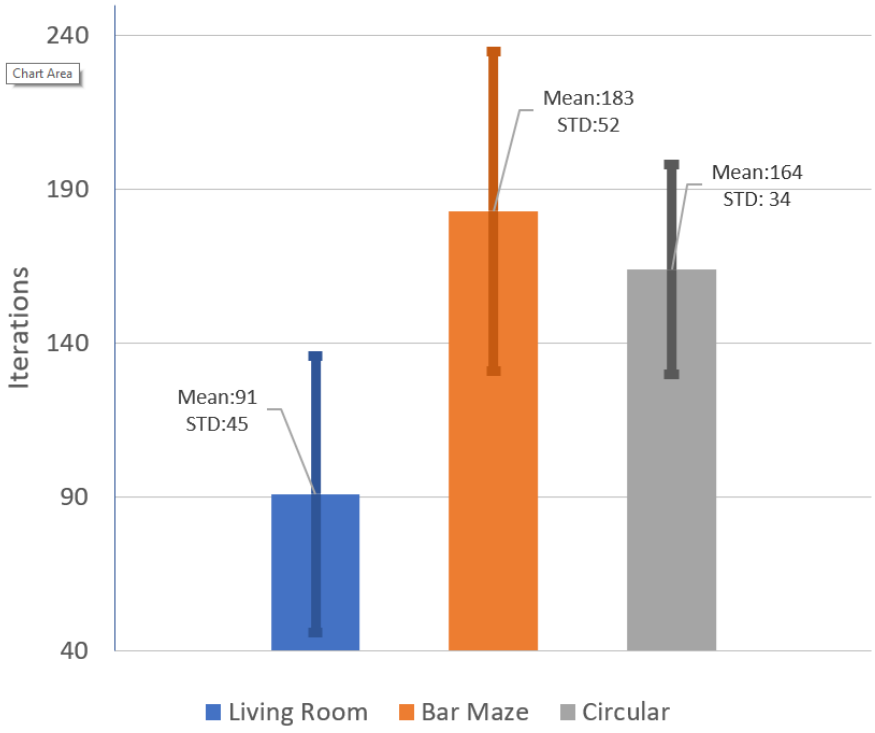


Figure 13: Performance of Yamauchi's exploration algorithm in Simulated 15 x 15 Grids

Appendix G Algorithms

Algorithm 1 Sensor Model

function `sensor_model`(z_t, $x_t^{[m]}$)
  $weight = 0$
  for i *in* z_t **do**
    x_i = Scan_to_Map(i_x) ▷ convert x_{scan} to x_{map}
    y_i = Scan_to_Map(i_y) ▷ convert y_{scan} to y_{map}
    compute log odds for each cell using ray casting
    $odds$ = log_odds(x_i, y_i)
    if $odds > odds_{thresh}$ **then**
      $weight++$ ▷ increment weight by 1
  return $weight$

Algorithm 2 Particle Filter

function `particle_filter`(X_{t-1}, u_t, z_t)
  $X_t = \emptyset, \eta = 0$
  for m `= 1 to n` **do**
    Sample index $j(i)$ from discrete distribution given by w_{t-1}
    Sample x_t^i from $p(x_t|x_{t-1}, u_t)$ using $x_{t-1}^{j(i)}$ and u_t
    $w_t^i = p(z_t|x_t^i)$
    $\eta = \eta + w_t^i$
    $X_t = X_t \cup < x_t^i, w_t^i >$ ▷ *add to new particle set*
  for m `= 1 to n` **do**
    $w_t^i = w_t^i/\eta$ ▷ *normalizing weights*
  return X_t

function `resampling`(X_t, W_t, n)
  $\tilde{X}_t = \emptyset, c = w_{[t]}^1, r = rand(0; n^{-1})$
  for m `= 1 to M` **do**
    $U = r + (m-1).M^{-1}$
    while $U > c$ **do**
      $i = i + 1$
      $c = c + w_t^{[i]}$
    add $x_t^{[i]}$ *to* $\tilde{X}_t$
  return $\tilde{X}_t$

Algorithm 3 Bayesian Filter

1: **function** `particle_filter`($bel(x_{t-1})$, u_t, z_t)
2: **for** `all` x_t **do**
3: $\bar{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$
4: $bel(x_t) = \eta p(z_t|x_t) \bar{bel}(x_t)$
5: return $bel(x_t)$