Team 6 ROB 550 Botlab Report

Bo Xue, Wei Jian, Yeyang Fang

***Abstract*—The botlab works with the MBot. A waypoint navigation system using dead reckoning with odometry is implemented to estimate the location of the MBot. This is based on IMU sensor and wheel encoders. Also, using 2D lidar, a simple simultaneous localization system is implemented. An A-star path planing algorithm and an exploration strategy are also finished to navigate the robot.**

After all these work, the robot is able to explore the unknown environment and get the map of the environment. It also can localize itself using the map.

I. Introduction

RECENT advances in technology have led to an increasing number of study related to self-driving cars. One of the key topic related to this field is to figure out what is the environment around the vehicle and the location of the vehicle in a map.

Probability robotics [1] is a subfield of robotics concerned with perception and control, and can be used to solve the problem. The occupancy grid algorithm is used to map the environment and to figure out the environment around the MBot. Monte Carlo localization algorithm is used to figure out the location in the map. Combined these together, a SLAM algorithm is developed and an A-star algorithm is implemented to navigate the robot in the maze.

II. Methodology

A. Mapping

We use occupancy grid algorithm to map the environment. For a $N \times N$ grid, the cell is either occupied or unoccupied. This can be updated through lidar sensor data. The laser of the lidar will terminate when hit an obstacle. So each laser beam can be used to update the grid. The grid hit by a laser beam will be seen as occupied while the grids the laser beam goes through will be seen as unoccupied. The occupancy grid mapping algorithm steps are shown in Algorithm 1. If the grid is considered to be free in this scan, then the log odds is decreased by -1.0, if the grid is occupied, then the log odd is increase by +3.0, ideally the value should be +3.5 as in the following equations, but since log odd is stored as int8_t, whose values are integers varying from -128 to 127, we cast it to +3.0

$$\begin{aligned}\lambda(x = D|occ(i,j)) &= \frac{p(z = D|occ(i,j))}{p(z = D)|\neg occ(i,j)} \approx \frac{.06}{.005} = 12\\ \log_2 \lambda &\approx +3.5\\ \lambda(x > D|occ(i,j)) &= \frac{p(z > D|occ(i,j))}{p(z > D)|\neg occ(i,j)} \approx \frac{.45}{.9} = 0.5\\ \log_2 \lambda &\approx -1.0\end{aligned} \tag{1}$$

Algorithm 1 Occupancy Grid Mapping Algorithm

Require: $laserscan$, map, $pose$

```
for each laserray ∈ laserscan do
    pose_start, pose_end =
        moving_laser_scan(laserscan, pose);
    gridcells = Bresenham's (cell_0, cell_t, map)
    for cell ∈ gridcells do
        if cell is occuppied then
            log_odd(cell) = log_odd(cell) + 3.0
        if cell is free then
            log_odd(cell) = log_odd(cell) - 1.0
```

B. Localization

1) Action Model: The action model is based on probabilistic robotics textbook. This is a odometry based action model. The noise is generated based on four parameters α_1, α_2, α_3 and α_4, the first two mainly affect the turning angle while the latter mainly affects the transitional movement.

$$\begin{aligned}\delta_{rot1} &= \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}\\ \delta_{trans} &= \sqrt{\bar{y}' - \bar{y}, \bar{x}' - \bar{x}}\\ \delta_{rot2} &= \bar{\theta}' - \bar{\theta} - \delta_{rot1}\end{aligned} \tag{2a}$$

The nominal control inputs δ_{rot1}, δ_{trans}, and δ_{rot2} are computed based on the following equations. Once nominal controls and noises are given, we sample an actual control input by a normal distribution. Then we apply the control signals to the action model to propagate to the next state.

To choose the suitable uncertainty parameters, we tried different noise of particles, and we found that increasing particles noise really help the localization. This is because if the particles noise is too small, the particles would not be sparse enough to catch up with

the true path. Thus, we choose the parameters (i.e. αs) big enough to fulfill this.

$$\begin{aligned}
\hat{\delta}_{rot1} &= \delta_{rot1} - \text{sample}\,\alpha_1\delta^2_{rot1} + \alpha_2\delta^2_{trans} \\
\hat{\delta}_{trans} &= \delta_{trans} - \text{sample}\,\alpha_3\delta^2_{trans} + \alpha_4(\delta^2_{rot1} + \delta^2_{rot2}) \\
\hat{\delta}_{rot2} &= \delta_{rot1} - \text{sample}\,\alpha_1\delta^2_{rot1} + \alpha_2\delta^2_{trans}
\end{aligned} \tag{2b}$$

$$\begin{aligned}
x' &= x + \hat{\delta}_{trans}\cos\theta + \hat{\delta}_{rot1} \\
y' &= y + \hat{\delta}_{trans}\sin\theta + \hat{\delta}_{rot1} \\
\theta' &= \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}
\end{aligned} \tag{2c}$$

TABLE I: Action Model Noises

Parameters	Values
α_1	1.5
α_2	0.3
α_3	1.0
α_4	0.2

2) Sensor Model: We used a different sensor model from the lecture slides. The sensor model algorithm steps can be seen in Algorithem 1.For each laser scan, there are a number of laser rays each end at an endpoint. Ideally at the position of each endpoint there should be obstacle cell, that is a cell with positive log Odds. If the map is accurate each obstacle cell should has a log odd of 127. If the log odd is negative, then it is inconsistent with the belief that there should be an obstacle, and possibility of this ray is deliberately set to 0, any non-negative log-odd is normalized to 1.0, and the mean of probability of each ray is returned as the probability of this laser scan. This sensor model has robust performance but slower response.

Algorithm 2 Sensor Model

Require: $laserbeams$, map, $sample$
1: $weight = 0$
2: **for** each $laserbeam \in laserbeams$ **do**
3: $pose_{start}$, $pose_{end}$ =
4: **moving_laser_scan**$(laserbeam, sample.pose)$;
5: $grids$ = **Bresenham's**$(cell_{start}, cell_{end}, map)$
6: **for** $grid \in grids$ **do** weight = weight + $\frac{1}{127}\min(0, log\_odd(end\_point\_grid))$
return $weight$/number of laser beams

$$\begin{aligned}
P(z_t|x_t) &= \prod_{i=0}^{N} P(ray_i = d_i|x_t) \\
\log P(z_t|x_t) &= \sum_{i=0}^{N} \log P(ray_i = d_i|x_t)
\end{aligned} \tag{3}$$

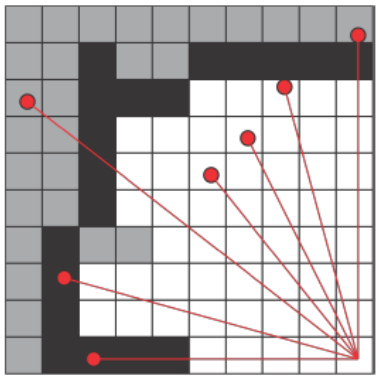

Fig. 1: sensor model illustration

3) Particle Filter: Fig 2 and Algorithm 3 shows a flow diagram of particle filter implementation. For each time step, we sample particles from initial particle pool with action model and weight them with sensor model. After this, resample the particles with weights given by the above steps.

Algorithm 3 Particle Filter Algorithm & Resampling

Require: χ_{t-1}, u_t, z_t
1: $\bar{\chi}_t = \chi_t = \emptyset$
2: M = Number of Particles
3: **for** each $m \in M$ **do**
4: sample ${x_t}^{[m]} \sim p(x_t|u_t, {x_{t-1}}^m)$
5: ${w_t}^{[m]} = p(z_t|{x_{t-1}}^{[m]})$
6: $\bar{\chi}_t = \bar{\chi}_t + \langle {x_t}^{[m]}, {w_t}^{[m]} \rangle$
7: **for** each $m \in M$ **do**
8: $r = rand(0; M^{-1})$
9: $w = 0$
10: $i = -1$
11: **while** $w <= r$ **do**
12: $i = i + 1$
13: $w = w + w_t[i]$
14: add $x_t[i]$ to $t[i]$
15: exponentiate and normalize χ_t
16: return χ_t

C. SLAM

Fig 3 shows a block diagram of SLAM system components. The former state x_{t-1} and command u_t predict the current state x_t by the action model. The sensor model gives the current measurement z_t and corrects the previous prediction and gives the estimate of current state. The current measurements from obstacles gives and updates the map information.

For implementation of localization, we tried different numbers of particles and different noise of particles, and

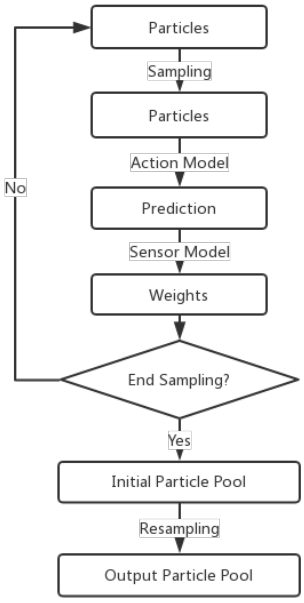

Fig. 2: Particle Filter block diagram

we found that increasing the number of particles does not have a significant improvement but increasing particles' noise does really help the localization. This is because if the particles' noise is too small, the particles would not be sparse enough to catch up with the true path. Thus, the tweak we choosed to improve the behaviour of localization is to increase the noise of particles.

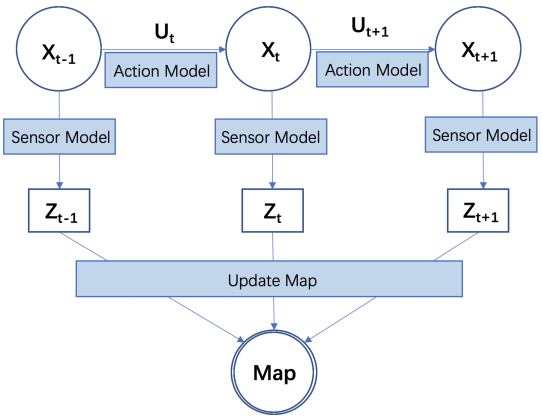

Fig. 3: SLAM block diagram

D. A^* Path Planning

A^* Path Planning aim to find a path to the given goal node having the smallest cost. During every iteration, the A^* will find the node with lowest cost and continue extending. The cost score f can be defined as the sum of g and h. The function g represents the cost of the path from the start node to n, which can be calculated by adding the g_cost of before node and the Manhattan distance between the calculated node and the before node. The function h represents the cost from calculated node to goal. It is calculated using Manhattan distance in our case. Then we have the expression for our f score:

$$f = g + h + Obstacle\_Compensate$$

The obstacle compensate score can be calculated by the following methods. If the minimum distance between a node and an obstacle is larger than the threshold `params.maxDistanceWithCost`, then it's 0. Otherwise it can be calculated by the following equation

$$Obs\_C = (maxDisWCost - dise(node))^{disCostExp}$$

Then we can get the f score for each node in the iteration.

The step for A* algorithm is shown in Algorithm 4.

Algorithm 4 A^* algorithm steps

Require: start,goal
1: **if** goal(start)=true **then return** Path(Start)
2: Openlist ← start
3: **while** Openlist ≠ ∅ **do**
4: sort(Openlist)
5: n ← Openlist.pop()
6: neighbors ← expand(n)
7: **for** nbor ∈ neighbors **do**
8: nbor.f ← nbor.g + nbor.h + obs_com
9: **if** goal(nbor)=true **then return** Path(nbor)
10: **if** nbor∩ Closelist=∅ **then** Openlist ← nbor
11: Closelist ← n
12: **return** ∅

E. Map Exploration

For the exploration part, we keep choosing the closest frontier point as the goal frontier. For each goal frontier, we find the closest valid goal point as the goal for each step of exploration. A valid goal point should satisfy the following requirements:

- The point is not in any obstacle areas (logOdds value should be negative).
- The distance from the closest obstacle point to this point is longer than the minimum obstacle threshold (`minDistanceToObstacle` in the codes).

After selecting the goal points, we use A^* planning method to navigate the robot. The above process is done several times until there are no frontier points found. For the actual implementation, we find that because the path generated by A^* only gives a serial of goal pose to the goal pose, the robot will not get any output path if all the possible path if blocked by the `DistanceGrid`, which is the enhanced obstacle area that is within some

given distance from obstacles. As a result, the robot may keep finding goal points but never stops and always ends up with no valid path. To handle this, we set the final state to be `fail to find valid path for 20 times` instead of `there are no frontiers detected`.

F. Localization with Unknown Starting Position

For the Kidnapping task, we detect the furthest obstacle as the moving direction and ensure the robot not hitting on any obstacle(always keep the distance from the closet obstacle further than a threshold). Before navigating the map, the particles are uniformly distributed in the open space of the map. With exploring in the given map, the robot will increase the certainty of its location in the map.

One important thing is that, because there are some locations in the map where the surroundings are similar, the robot may regard some place which is similar to the current location. In order to avoid this and enhance the accuracy of its detected location, the robot should explore the map as much as possible without hitting any obstacles it detected (not get from the given map) even if it has already a high certainty of its detected location. After this implementation, the robot will gradually recognize its location correctly.

III. Results

A. Mapping

We run our occupency grid algorithm on the logfile `obstacle_slam_10mx10m_5cm.log`. And we get the image of the map. We tried several times to test the performance. Fig 4 shows the image of the map without odometry and Fig 5 shows the image of another map with odometry.

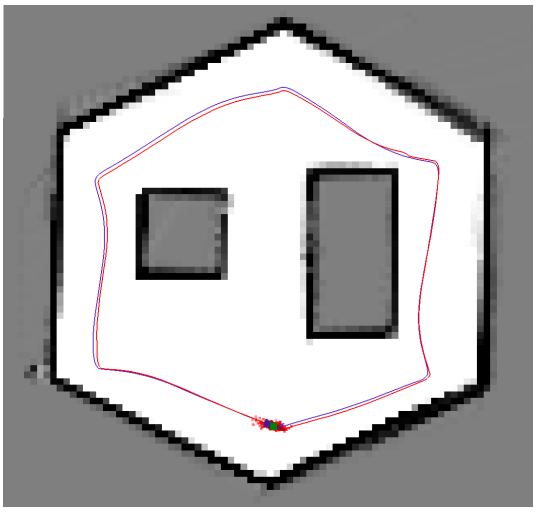

Fig. 4: The image of the map without odometry

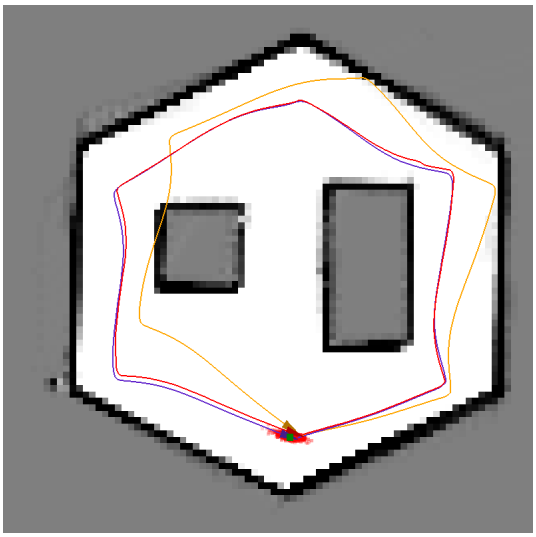

Fig. 5: The image of the map with odometry

B. Localization

We tested the performance of the system with different number of particles. It is obvious that more particle requires more time to simulate. The time required to update is shown in table II. We interpolate these data and is shown in Fig 6. And we estimate that the maximum number of particles our filter can support is approximate 4500 particles running at 10Hz on the RPi.

For the obstacle slam log and map, the errors is illustrated in 8, the upper subplot shows the distance between slam pose and ground truth pose from log in the 2D plane, while the lower subplot shows the difference in the heading angle. The distance between two poses oscillates during the entire process, but finally stabilized at 0.05m after one round, also the heading angle error almost totally disappeared after the circuit. The time stamp of ground truth and slam pose are almost always inconsistent, so we use the 1D linear interpolation to cast the ground truth according to the time stamps of slam poses and calculate the errors. This test uses 300 particles and the result is very satisfactory, actually our model can perform pretty well with only 10 particles.

Figure 9 shows the errors in drive square log in convex environment in simulation, we only have the ground truth

TABLE II: Time required to update the particles

Particle Number	Time[μs]
100	2.62
300	7.21
500	11.55
1000	22.38
4000	87.06

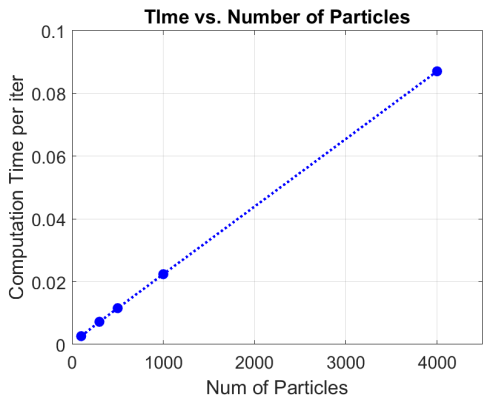

Fig. 6: Time required to update the particles

TABLE III: Error for SLAM Implementation

Maps	Mean of Error	Stdev of Error	Max Error
EucD in Obstacle	3.28cm	1.73cm	15.77cm
Head in Obstacle	-0.0114rad	0.0628rad	0.5191rad
EucD in Square	3.44cm	1.96cm	13.22cm
Head in Square	-0.0369rad	0.2973rad	-3.2076rad

We also tested the particle filter performance in simulator. We choose particle number equals to 300 for a fast and accurate simulation.The Fig 7 shows the particle distribution during the simulation process. The first image shows the distribution of particles at the beginning and the other images shows the distribution of the particles at the midpoint of each translation and the corners after turning 90 degrees. We can see from the simulation that our particle filter works very well.

C. SLAM

Fig 4 shows the result of SLAM in obstacle map. Table III gives the mean, standard deviation, and max of the pose error (the error between SLAM pose and ground truth pose). In the table, `EucD` means Euclidean Distance Error, `Head` means Heading Angle Error, `Obstacle` means the obstacle map, `Square` means the driving square map (here the error is calculated by simulation on MATLAB). From the table, we can see that in obstacle map, the error is tolerable although there is a big max error. Fig 8 shows the pose error evolving over time. The blue line shows the Euclidean Distance Error, while the red line shows the Heading Angle Error.

For the driving square test for 4 times, Fig 10 shows the odometry pose (The red line in the figure) and the slam pose (The yellow line in the figure) when we drive the robot following a square four times. Since the optitrack system is currently not working, we don't have the result for ground truth. By comparing the result we can see that the slam pose is similar to odometry pose. However, there is a still drive while the robot is driving. The reason is that our MBot two motors with different max velocity. So it will always turn right while driving along the straight line and modify the error at the end of the line. Despite of this, we can say our slam algorithm is good enough.

For the error calculation, because we do not have ground truth for real world implementation and there is only one-time driving square log file, we simply use the one-time driving square log file to calculate the error. The error is shown in Table III. And Fig 9 shows the pose error evolving over time. The blue line shows the Euclidean Distance Error, while the red line shows the Heading Angle Error. Here we can see that there is a very big max error, this is because the map is highly symmetric and there is a breakpoint of the Heading Angle change (the $+\pi$ and $-\pi$ discontinuity).

D. A^* Path Planning

Because the optitrack is not working, we use the slam pose as the ground truth and to test our A-star algorithm on robot. Fig 12 shows the process of map construction. In the A-star algorithm we use on the robot, the step is set to 3 for better robot control while in the A-star test we set step equals to 1 for validation. We manually right click the mouse to assign a destination on the map. The green dots and line shows the path calculated by our A-star algorithm, and the yellow line shows the moving trace of the slam pose, which can be seen as the path of the robot in the real maze. We can see from the result that the yellow line followed the path of green dots. The robot moving path (yellow path) is more smooth at the turning compared to the green path generated by the A-star algorithm because there is a threshold in the robot turning control.

Fig 11 shows the process of returning to the start point using A-star algorithm after finishing construction of the map. We can see from the image that the A-star algorithm generated an optimal path for robot to move and the robot followed the path accordingly. This shows the great performance of our A-star algorithm and the robot motion control.

We also tested our A-star algorithm using provided A-star test algorithm. Our A-star algorithm can pass all 6 tests in the A-star test algorithm. Table IV shows the time information of the successful attempt and Table V shows the time information of unsuccessful attempt. All data are in unit of us. The astar test program is ran on the laptop computer.

E. Map Exploration

Fig 13 shows the final results of map exploration. The yellow line shows the track of each step. The green points show the serials of goal points generated by A^*. We can see that with exploring the map, the robot has an

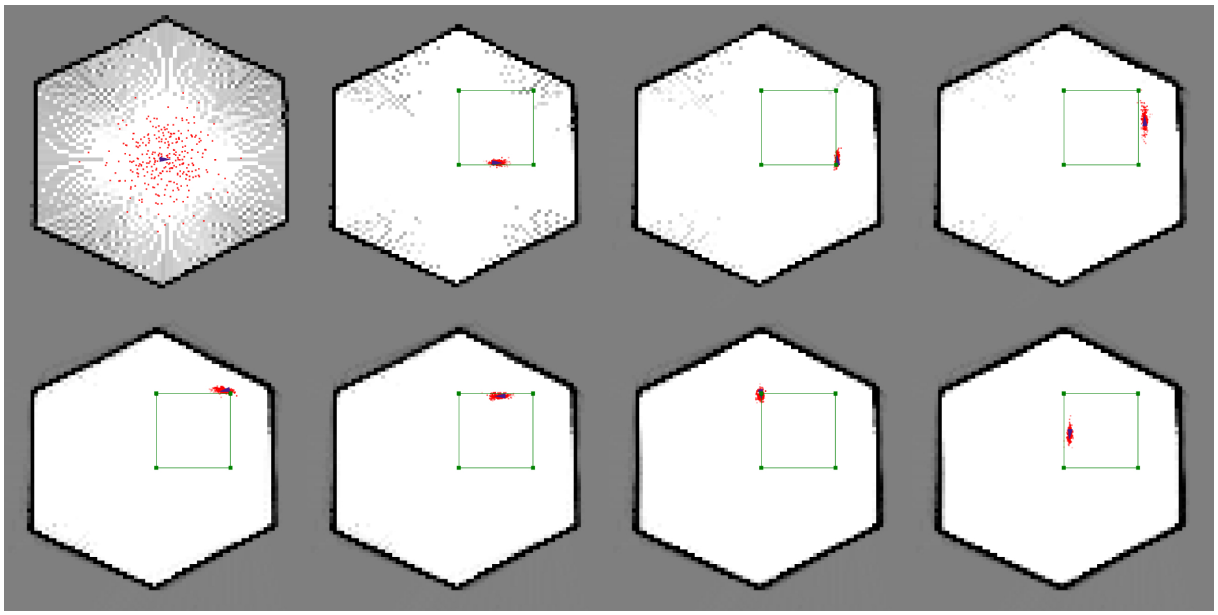

Fig. 7: Particle filter on drive square log

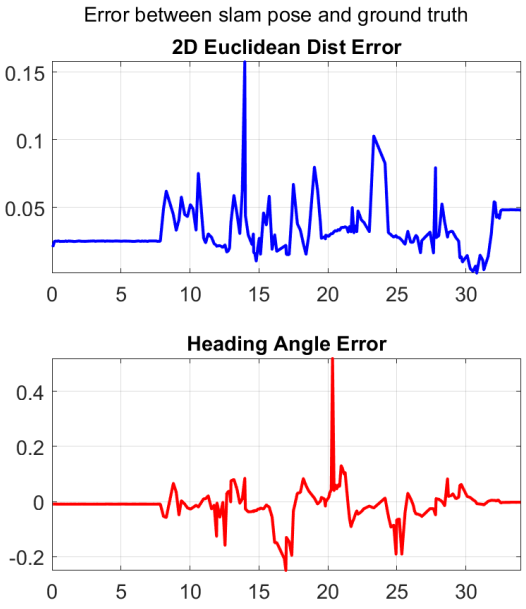

Fig. 8: Obstacle Slam log error

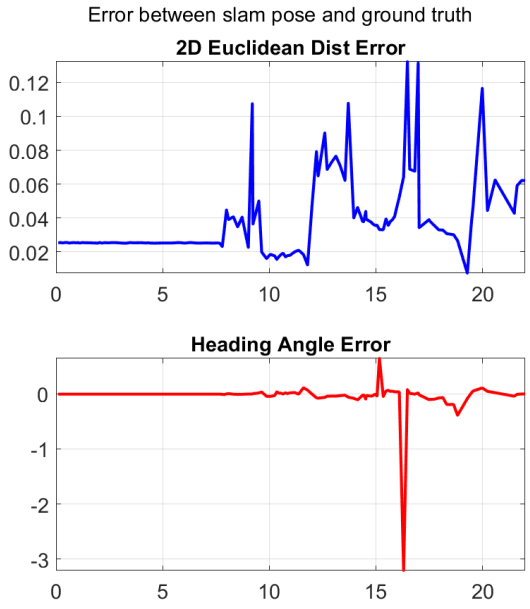

Fig. 9: Drive Square Convex Map

increasing certainty of the map. And there is no uncertain points in the map after the whole exploration.

Although the above result seems accepted, sometimes the robot cannot navigate to some frontiers although the frontiers are detected. This is because there does not exist an valid path.

F. Localization with Unknown Starting Position

Since the maze was not available while we are testing our algorithm, so we tested our algorithm in simulation. The localization process is shown in Fig 14. The full video can be found on canvas for extra credit.

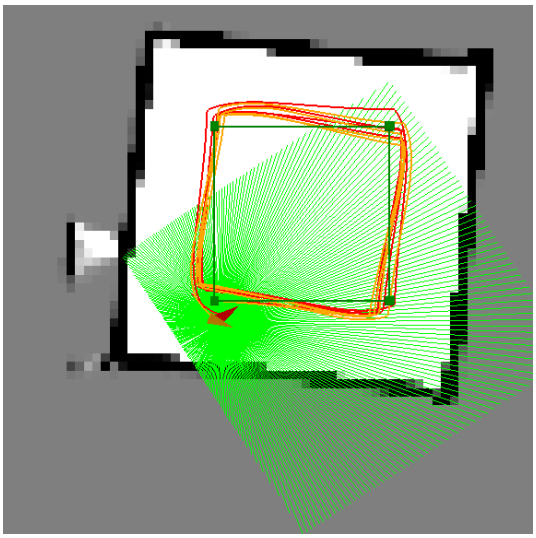

Fig. 10: Drive square result

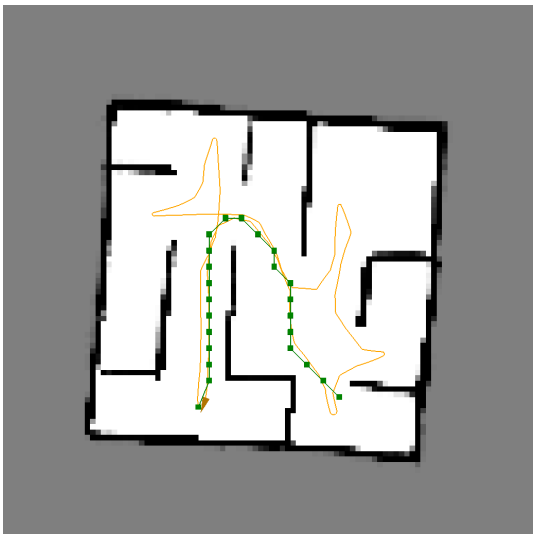

Fig. 11: The return process navigated by A-star

IV. Discussion

The overall implementation of SLAM and A-star path planning is successful. Besides, we can still make some discussions and there are still some ways to improve our algorithm.

TABLE IV: A-star test time information for successful attempt

Test name	Min	Mean	Max	Median	Std Dev
convex	1416	1490.67	1687	1453.06	91.82
empty	3403	3736.03	4382	3656.07	269.68
maze	1534	1644.3	1966	1601.89	117.998
narrow	3334	4392.35	5521	4060.94	842.59
wide	3321	4084.7	5570	3872.25	804.02

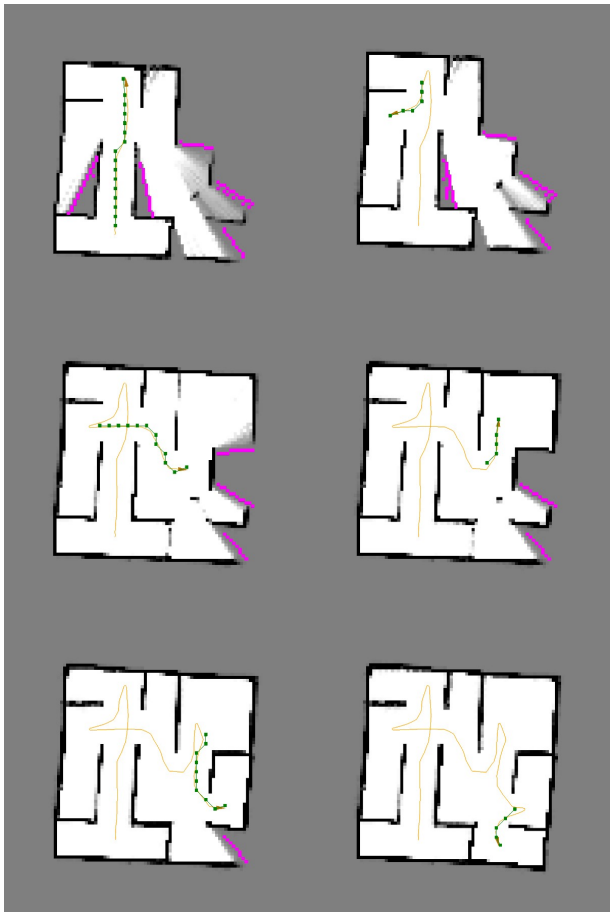

Fig. 12: Map construction navigated by A-star (manual control)

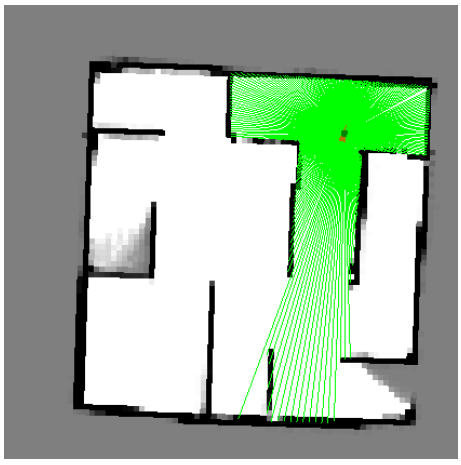

Fig. 13: The map of the competition maze

TABLE V: Time information for unsuccessful attempt

Test name	Min	Mean	Max	Median	Std Dev
convex	1	57.13	1679	8.29	296.13
empty	1	3.35	18	2.48	4.08
maze	0	2.78	18	2.00	3.2881
narrow	1	8048.87	21770	8265.64	9095.81
wide	1	3.6	13	2.25	4.05

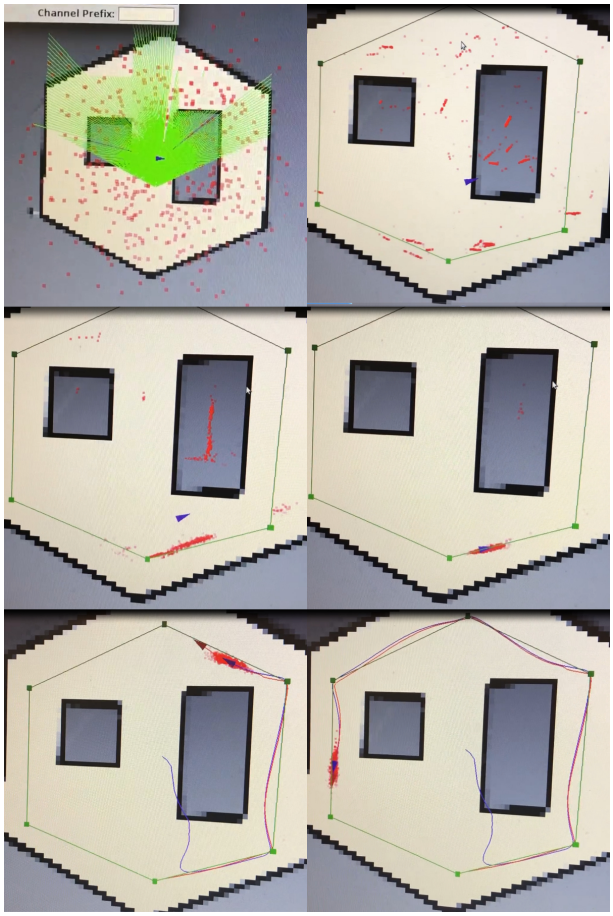

Fig. 14: Localization result

In our slam algorithm, the major modification is the action model, in the original lecture model there are only two parameters, but we use four $alpha$s. I tuned the parameters based on the homework of EECS 568 mobile robotics. The noise cannot be too small, otherwise we will not be able to sample poses near the ground truth, and as a result the slam pose will deviate from the ground truth and approach to the odometry poses. In our test our model can handle very large noises and initial uncertainty, as we can see our model can robustly estimate the location in simulation. Also our model can perform well even with a very small number of particles with little deviation. We tried 10 and 20 particles and the results are very satisfactory. The second main issue is the modification of then sensor model. Instead of using the lecture sensor model, I chose the sensor model which solely depends on the end points of each laser beam. Since the beam should always end at an obstacle. This sensor model is very easy to implement and performs very well, the only drawback it has a relatively slow response compared to other teams' sensor model. But as we can see from the SLAM and localization results. But slower response also leads to a much smoother behaviors. If the diversity of particles diminish too fast, the slam trajectory will become very noisy, in our case the estimated path is always very smooth.

In A-star path planning algorithm, though our A-star algorithm is fast and accurate enough for the competition, there are still many ways to improve the performance. For example, we can use the priority queue data structure during iteration to find the node with minimum f cost. We can also use methods like BFS search to accelerate the process of finding the minimum distance between a node to obstacles.

In our kidnap application, Our particles are sampled in the whole space. We can improve the performance of this algorithm by sampling the particles inside the maze instead of the whole space.

REFERENCES

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.