Team 212 ROB 550 BotLab Report

HaoTsung Lee, KuanTing Lee, Xili Yi

***Abstract*—Mobile robots have been developed rapidly these year. They have the ability to move and explore the environment without human intervention. This paper demonstrates a large prototype implementations of SLAM and A* for a differential wheeled robot, MBot-Mini. Our project is broken down into three parts: 1) Motion Controller, 2) Mapping and Localization, and 3) Path Planning. By doing so, we built a autonomous robot navigation system, which can explore unknown maps. Our work won the first place in ROB 550 class competition.**

I. Introduction

IN this project we implemented a differential wheeled robot capable of exploring a maze autonomously and ultimately returning to its origin position. Self-driving vehicles have become one the most popular trend in the past few decades. These vehicles rely on autonomous navigation and map building to gather the information around the environment. To achieve these, we firstly designed the motion controller of our wheeled robot. We need to make our robot drive smoothly and follow the designated trajectory. Then we estimated the location of the robot and also drew the surrounding map based on odometry and laser scan data, and planed the path using the A* algorithm.

II. Methodology

This section describes the various methods and models used for the different tasks in the project.

A. Odometry

To control the robot's position, we need to be able to tell the robot how to move. One simple way to control the robot speed is to use encoder data to control the wheel speed using a feedback controller. We keep track of the robot's relative displacements by encoders with respect to a known initial position. If we assume no slippage occurs, the displacement of the robot can be formulated as a function of the encoders' readings. The followings is our odometry equations.

$$\triangle\theta = \frac{\triangle s_R - \triangle s_L}{b}, \triangle d = \frac{\triangle s_R + \triangle s_L}{2},$$
$$\triangle x = \triangle d\cos(\theta + \triangle\theta/2), \triangle y = \triangle d\sin(\theta + \triangle\theta/2)$$

$\triangle s$ can be calculated directly from encoders, then we can get displacements and heading angles from relative to initial pose.

Gyroscope also can be used to provide more robust odometry, which is called Gyrodometry. The idea is to fuse the rotational displacement with gyro and encoders. Consider the situation of driving very fast, which the robot may slip, or hitting a bump, relying only on encoders is apparently not enough. Thus, if gyro reads significantly different than encoders odometry, we should trust the gyro. Algorithm 1 shows the decision method. We choose the threshold as 0.000125.

Algorithm 1 Gyrodomertry

$\triangle_{G-O} = \triangle\theta_{gyro} - \triangle\theta_{odometry}$
if $|\triangle_{G-O}| > Threshold$ **then**
  $\triangle\theta = \triangle\theta_{gyro}$
else
  $\triangle\theta = \triangle\theta_{odometry}$

To better determine the pose of the robot, a robust calibration approach must be applied to the odometry. UMBark[1] is a method for measuring, comparing, and correcting dead-reckoning errors in mobile robots. We will use this standard approach to adjust robots' physical parameters such as baseline (lateral distance between the two wheels) and wheel diameter based on the error in turning and forward motion respectively.

B. Motion Control

The whole motion controller consists of 3 loops, the outer loop is motion controller loop, the second loop is velocity frame controller loop, and the inner loop is the motor controller loop. Finally We use these to implement a Rotation-Translation-Rotion(RTR) trajectory follower.

1) motion controller: We design our motion controller by proportional control. This controller takes target position as input, odometry that read from encoders as feedback and outputs setpoint velocity. The control law is to drive (1) distance to goal position, d, (2) angle between current heading and heading towards goal, θ, to 0. We also added a saturation filter to limit the maximum output velocity setpoints to avoid uncontrollable behavior. The following is our parameters K_P of the motion controller.

translation velocity	translation heading	rotation
1.5	1	1

2) velocity frame controller: We use proportional controllers for turning and forward velocities with feedback from odometry and then combine with feedforward of setpoints velocities. Next, we convert them to right and left wheel velocities respectively. We believe the feedforward part is more important than the PID control, so we only set K_P = 0.01. Additionally, we add a low-pass filter for setpoint forward velocity, to prevent rapid acceleration and jittering during the forward motion, which causes wheel slipping and therefore incorrect odometry result.

3) Motor controller: We use PID controllers for motor control, which take velocity from the velocity frame controller as input and output PWM signals to drive the motors. We set K_I much higher than K_P to reduce the rising time of motor rotational motion. The following is our parameters for both wheels.

K_P	K_I	K_D
1.5	20.0	0.01

We first tuned the value K_I to minimize the steady-state error. Than we adjusted K_P and K_D to make rising time and overshoot better.

C. Simultaneous Localization and Mapping (SLAM)

We begin by constructing an occupancy grid using known poses. Following that, we implement Monte Carlo Localization in a known map. Finally, we put each of these pieces together to create a full SLAM system.

1) Mapping: To navigate through a map, we need to avoid hitting obstacles and boundaries. Thus, we use laser scanning form a 2D LIDAR to provide such information. The goal is build a occupancy grid. For every cell in the grid, we calculate its log odd, which suggests whether the cell is occupied or free. To be more specific, we find end point of each ray and then determine cells along each ray that are currently free or occupied by Breshenham's algorithm. By calculating all the ray around 360 degrees, we can estimate the map.

2) Monte Carlo Localization: Monte Carlo Localization (MCL) is a particle-filter-based localization algorithm. Implementation of MCL requires three principle components: an action model to predict the robot's pose, a sensor model to calculate the likelihood of a pose given a sensor measurement, and function for finding the weighted mean pose of the particles and drawing samples.

a) Action Model: The action model determines the certainty our state after a given action applied. We add some Guassian errors to our translation and rotation functions for our RTR controller. The following is our equations for each action $u = [\alpha, \triangle s, \triangle\theta - \alpha]$.

$$\begin{bmatrix} \triangle x_t \\ \triangle y_t \\ \triangle \theta_t \end{bmatrix} = \begin{bmatrix} (\triangle s + \varepsilon_2)\cos(\theta_{t-1} + \alpha + \varepsilon_1) \\ (\triangle s + \varepsilon_2)\sin(\theta_{t-1} + \alpha + \varepsilon_1) \\ \triangle\theta + \varepsilon_1 + \varepsilon_3 \end{bmatrix}$$

Where:

$$\varepsilon_1 \sim \mathcal{N}(0,\, k_1|\alpha|)\,, \varepsilon_2 \sim \mathcal{N}(0,\, k_2|\triangle s|)\,,$$
$$\varepsilon_3 \sim \mathcal{N}(0,\, k_1|\triangle\theta - \alpha|)$$

The Guassian standard deviations of translation and rotation are proportional to their value by k1 and k2, respectively. In our model, k1 and k2 are set to following numbers because it gives us optimal SLAM quality.

k_1	k_2
0.01	0.01

Any smaller value can make the robot rely on odometry too much and therefore lead to a deviation from the true path. Any higher value will overly spread the particles and make the convergence of particles impossible.

b) Sensor Model: The sensor model finds the probability that a LIDAR scan matches the hypothesis pose given a map. We utilize a simplified likelihood field model to improve mapping quality. In this model, we increment the score by its original log odd, rather than just add a constant number. This makes sense because if a cell has higher log odd, we have more confidence that it is occupied. We not only increase the score if the ray endpoint is cast on an obstacle but also check cells before and after along the ray endpoint and increase the score by a fraction of 0.5 of its log odd value. We use Breshenham's algorithm, same as mapping, for only one step to find the cells before and after. This resulted in a smooth and robust map around the boundary of the obstacles in the map.

c) Particle Filter: The Particle Filter iteratively determines and updates the current pose of the robot, while the robot is moving. It assigns a set of potential poses with various weights, which represents the probability of guess of each pose. These poses with weights are called particles. We use 200 particles.

For each iteration, the particles are first redistributed based on the weight of each particle. We use low-variance resampling which is the approach to only select particles with higher weight for the later updates in particle filters. The mechanism behind the algorithm is to continuously increase a threshold value and accumulate particle weights at the same time. When the sum of particle weights is higher than the threshold, select the particle for future updates. Note that particles with high weight can be sampled multiple times.

Then, we use the redistributed particles to generate new ones via the action model and again estimate the weights by the sensor model. The weights of particles also need

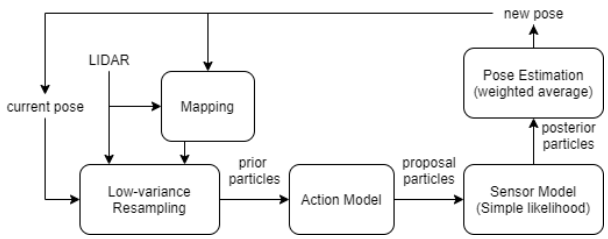


Fig. 1: SLAM architecture

to be normalized to sum to one. Finally, the average of the weighted particles is returned as the estimated pose.
3) Combined Implementation: Figure 1 shows the block diagram of our SLAM system.

D. Planning and Exploration

First, we implement A* algorithm for planning. Then, we devise a frontiers exploration method to make the robot drive toward unknown areas in the map.
1) Path Planning: A* is a path planning algorithm that can efficiently find an optimal path in a map with many obstacles. We adopt A* to drive toward the determined waypoints. Since the robot cannot get too close to obstacles, prior to planning, we use Brushfire Algorithm to build obstacle distance grid maps, where each cell is assigned a value that represents its distance to its closest obstacle. If the distance value of a cell is less than the robot's radius, A* should plan a path to avoid this cell, even this cell is free. In practice, to smooth the path we keep only one waypoint every 3 waypoints.
2) Exploration: To maximize the use of SLAM, a complete and high quality map is needed. The robot needs to explore through the whole map to generate a closed and complete map. We constructed a method to explore the map by continuously planning paths to the nearest feasible cell until no more frontiers can be found.
a) Frontiers Extraction: A frontier cell is defined as an unknown cell neighbored by a free cell on the Occupancy Grid, while a frontier is a set of adjacent frontier cells. In practice, we set a minimum frontier cell requirement to reduce too small frontiers or noise. For each iteration of finding frontiers, we used Breadth First Search(BFS) through the current map and trying to find frontier cells. Whenever a new frontier cell is found, we expand the frontier from this cell to obtain all the adjacent frontier cells in this frontier. If the size of this frontier meets the minimum frontier size requirement, this frontier is saved.
b) Robot feasible cells: The basic requirements for the robot to obtain a path using A* algorithm are: start cell and goal cell are both in the safe range in the ObstacleDistanceGird, and they can be connected by a series of continuous safe cells in the same ObstacleDistanceGird. We define all safe cells that can be connected to the current robot cell directly or indirectly as Robot feasible cells, the robot can plan a valid path to any Robot feasible cells. Considering the robot might be in a dangerous cell, we first used BFS from the robot current cell to find the nearest safe cell as the robot cell. Then we expand the robot cell using BFS and set every adjacent cell which is safe in ObstacleDistanceGird as a Robot feasible cell. Then we should choose a goal cell from these Robot feasible cells in the next step.
c) Nearest feasible cell: We should choose a goal cell from Robot feasible cells to explore the frontiers. Tao et al. (2007) provided a simple method by just choosing the geometry center of the nearest cell as the goal.[2] However, simply choosing the geometry center of the nearest frontier may lead to invalid goal or invalid path. To solve this problem, we used another BFS starting from the geometry center of the nearest frontier to find a cell in the aforementioned Robot feasible cells. The first cell found by this method is set to be the goal for planning.
d) Exploration algorithm: The robot explore the whole map by finding frontiers and execute the path planned given the goal as aforementioned, when there is no frontier, the exploration is done. To explore a single frontier, the robot doesn't have to reach the goal, but just being able to pass the obstacles is enough. We used a time-based method to avoid the robot to move too far. The robot will find all the frontiers and re-planning a new path after 8 second. If a frontier is explored in this 8 second period, this frontier will disappear in the map and the robot will plan a path to another unexplored frontier. If the frontier is not explored in this 8 second period, because the robot is moving to that frontier, the nearest frontier keeps the same, so the new path planned using A* will generally be the same.

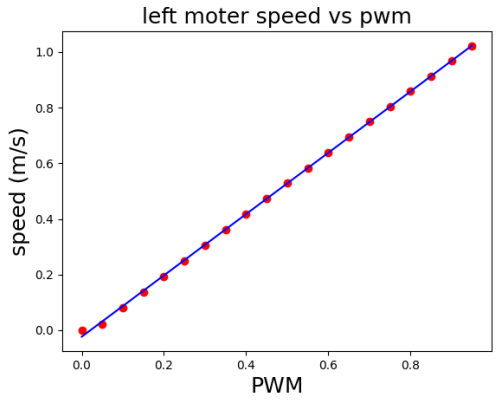
III. Results

The following section shows the result of different experiments to evaluate the performances of algorithms and methods implemented on MBots.

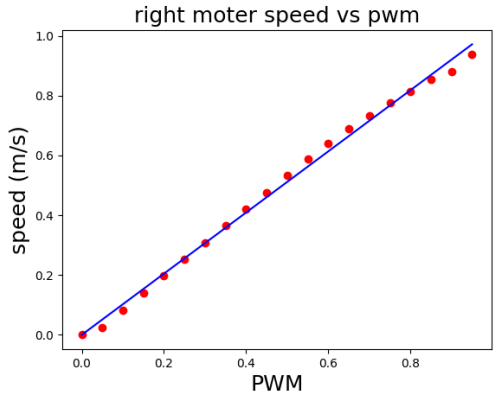
A. Motion and Odometry

a) Motor Calibration: To ensure consistent and stable performance of the motors, we set up an experiment to derive speed-PWM relation for the motors when they are running on a concrete floor. We sent PWM signals to the motors from 0 to 1 with a step of 0.05. The corresponding wheel speed was calculated based on the reading from wheel encoders. After applying linear regression to our data, the relationship between speed and PWM can be derived. Figure 2 shows the plots of

the motor speed vs. PWM for the loaded wheels and fitted curve. We calibrate two Mbots. Table I shows their fitted parameters, which are quite similar.



(a) left motor



(b) right motor

Fig. 2: Motor Calibration

	MBot	1	2
left	slope	1.01019	1.1019
left	intercept	-0.20232	-0.02319
right	slope	1.02344	1.02344
right	intercept	0.00048	-0.0004

TABLE I: Motors' fitted parameters

b) Frame Velocity Controller: Figure 3 shows the plots we used to test the frame velocity controller. The left column is plots of robot's forward position vs time for different step inputs. The right column is plots of robot frame heading vs time for various step inputs. Most of them look pretty good and linear, speed rising smoothly, except for the case of driving at full speed(1 m/s), which let the wheels slip.

c) Odometry: In the beginning, We test the odometry manually. Translation is validated with a ruler and Rotation is validated by turning 2π then seeing whether it could turn back to the original pose. The accuracy of the odometry is then tested by measuring the difference of initial and final pose after the robot executes an open-loop drive square command.

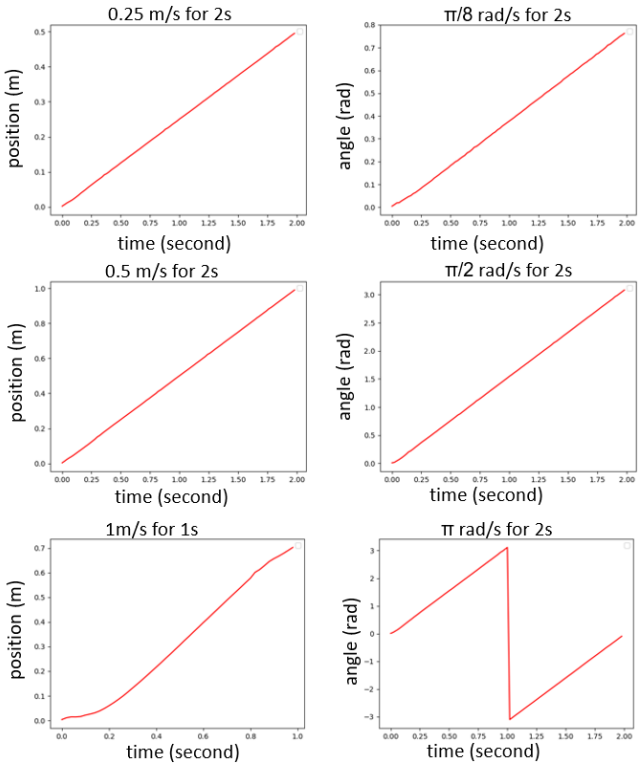


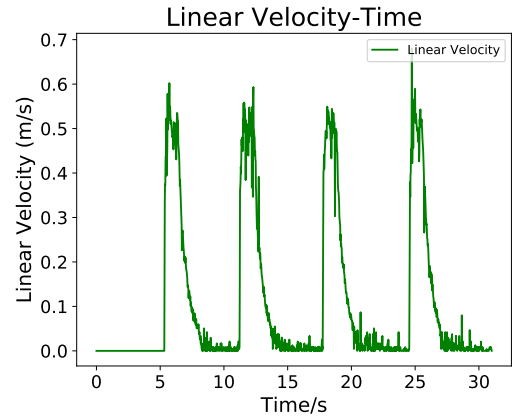
Fig. 3: Plots of robot frame velocity controller. The figures in the left column show position over time with desired speed of 0.25m/s, 0.5m/s and 1m/s. The figures inthe right column show angle over time with desired angular speed of $\pi/8$ rad/s, $\pi/2$ rad/s and π rad/s.

B. SLAM

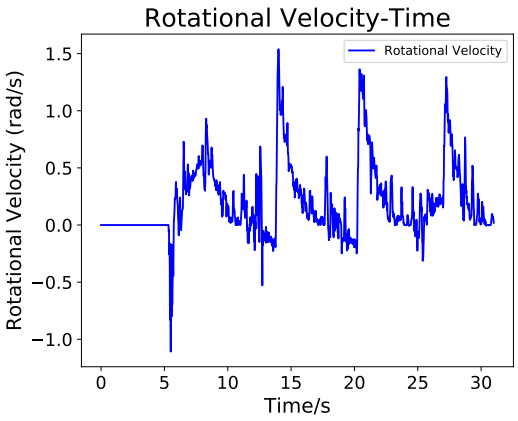
a) Occupancy Grid: An Occupancy Grid is generated with the observation in log file obstacle_slam_10mx10m_5cm.log. The Occupancy Grid is shown in Fig.6, where the black area denotes obstacles and the grey area is the unknown area.

b) Sensor Model and Particle Filter: By setting the number of particles used in SLAM process as 100, 300, 500 and 1000, the average time consumed by SLAM iteration and corresponding Root Mean Square (RMS) errors are shown in table II. The SLAM algorithm is tested on drive_square_10mx10m_5cm.log. The first 107 and last 40 data are abandoned because no motion is executed and the time consumed is meaningless for real SLAM scenario.

The time consumption is linear to number of particles, and the data above can be fit to Fig.7. The maximum



(a) Linear Velocity



(b) Rotational Velocity

Fig. 4: Plots of linear velocity and rotational velocity as MBot drives one loop around the 1 meter square. (a) shows the Linear velocity, (b) shows the rotational velocity.

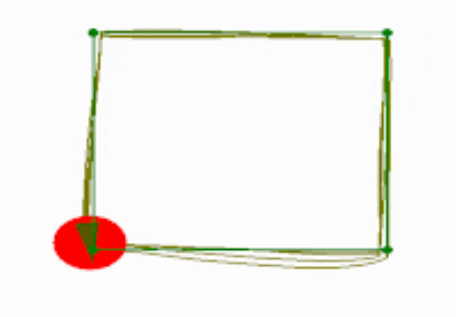


Fig. 5: Plot of the dead reckoning estimated pose as the robot is commanded to drive a 1m square 4 times

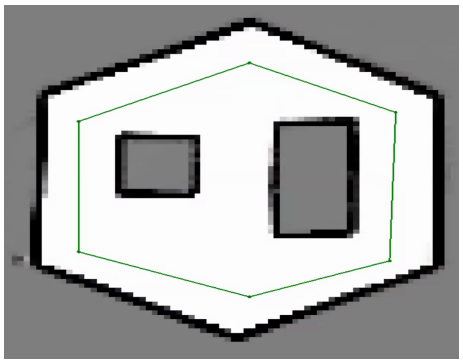


Fig. 6: The occupancy grid generated by the log file obstacle_slam_10mx10m_5cm.log

Number of particles	runtime average (μs)	runtime RMS (μs)
100	7425	3258
300	17461	6730
500	27252	9777
1000	48206	9877

TABLE II: Mean and RMS of SLAM iteration runtime for different number of particles

number of particles your filter can support running at 10Hz on the RPi is 140.

The error between SLAM Pose and Odometry is calculated and are shown on Fig.9. Those SLAM Pose and Odometry are obtained by using log file obstacle_slam_10mx10m_5cm.log with localization function only . The maximum distance error is 0.50 m and the maximum magnitude of orientation is 0.28 rad.

c) SLAM: The whole SLAM algorithm is tested using log file obstacle_slam_10mx10m_5cm.log, and the error between SLAM Pose and ground truth is shown in Fig.10. Both distance error and angular error are shown. The distance is obtained by calculating the Cartesian

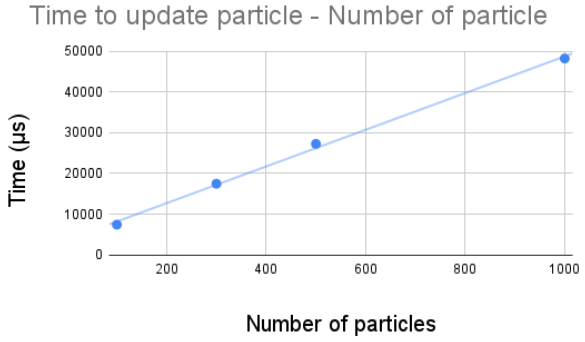


Fig. 7: Plot of the relationship between number of particles and their corresponding update time in SLAM

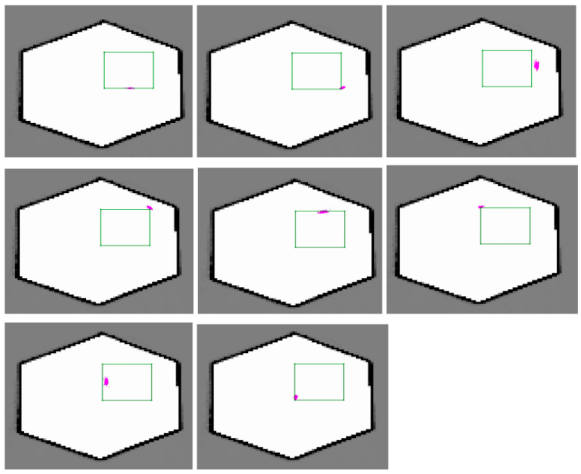


Fig. 8: Plot of 300 particles at the midpoint of each 1m translation and at the corners after having turned 90 degree

distance between SLAM Pose and ground truth.The RMS of distance error is 0.0616 m and the RMS of angular distance is 0.082 rad.

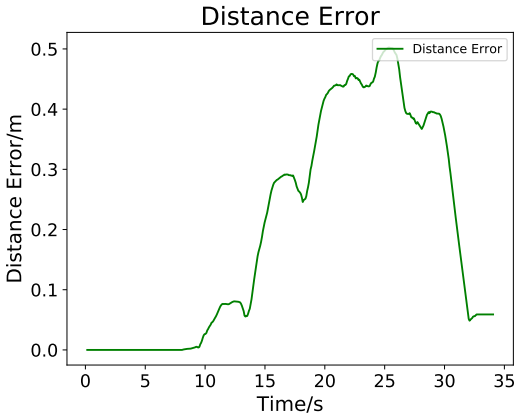
C. Planning and Exploration

a) A Planning:* A* planning algorithm is applied to both the test cases and the real environment. The planning results on astar_test_file are shown in Fig.11 and Fig.12. The algorithm passed most of the test cases. The A* algorithm is applied on MBOT to move in a maze, Fig.13 shows the planned path and MBOT's odometry. The green line and dots were the waypoints produced by the A* algorithm and the blue line is the path travelled by the robot.

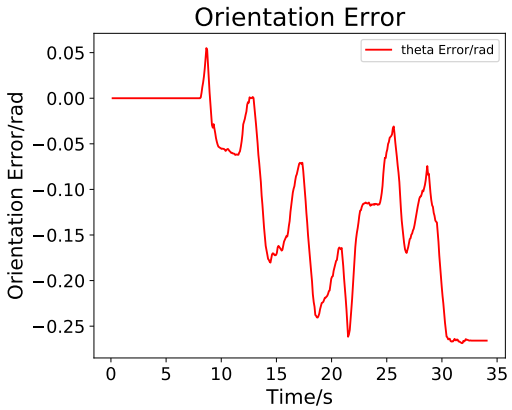
b) Exploration: The exploration result is shown in Fig.14. The red area are the grids that have obstacle distance smaller than the robot radius, which means the robot are not allowed to travel these grids. The pink area is the frontier, which are regions on the boundary between open space and unexplored space. The robot is programmed to visit the closet reachable area to the closet frontier in the map, which is pointed to by the mouse cursor.

IV. Discussion

This section discuss and interpret the results of different experiments.



(a) Distance Error

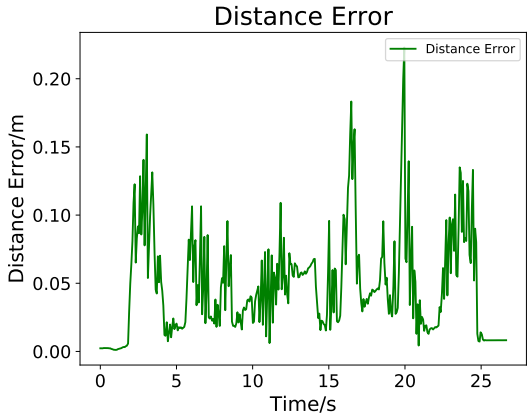


(b) Orientation Error

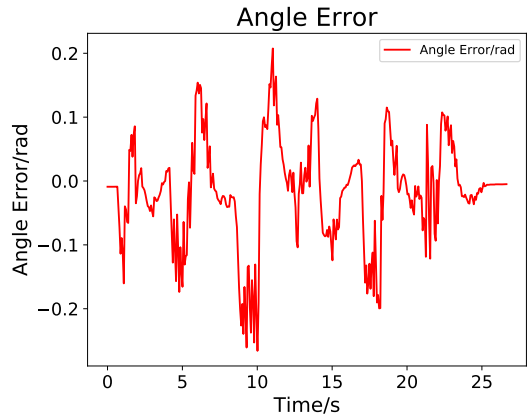
Fig. 9: Plots of Pose error between pure Odometry and SLAM Pose. (a) shows the Cartesian distance between Odometry and SLAM Pose, (b) shows the Orientation differences between Odometry and SLAM Pose.

A. SLAM

From the results of time consumption with respect to the number of particles used in SLAM, they have a linear relation ship, which means estimated time consumption can be determined by choosing a specific particle number. The difference of error between SLAM Pose and ground truth obtained with 1000 particles and 200 particles is less than 5%, considering time efficiency, 200 is chosen to be the particle number in the rest of the experiment. The error between SLAM Pose and ground truth can be treated as unbiased error while the error between Odometry and SLAM Pose is biased because of possible systematic error of physical parameters such as the length error of wheel base, the error of wheel diameter, or the drifting of gyro. Figure 15 shows our SLAM performance.



(a) Distance Error



(b) Angle Error

Fig. 10: Plots of Pose error between SLAM Pose and Ground Truth . (a) shows the Cartesian distance between SLAM Pose and Ground Truth, (b) shows the Orientation differences between SLAM Pose and Ground truth.

B. *Planning and Exploration*

a) A Planning:* There are two ways in generating the obstacle distance grid, using 4-way or 8-way connection, while they results in different types of boundary in which the path can be generated. Using 4-way connection will generate a wider obstacle, which will limit number of Robot feasible cells, meanwhile it can keep the robot from the obstacle. A* algorithm can also use 4-way or 8-way connection, 8-way connection can lead to a smoother path, and can generate a path that 4-way connection cannot. 4-way obstacle distance grid and 8-way A* algorithm is thus chosen to use in all the other experiments. There is a single test case that takes us around 5 seconds to determine that it is a failed planning attempts, which is the test_narrow_constriction_grid. Since the test case is an unbounded area, we set a maximum iteration times for the robots to explore. If the maximum iteration

```
Timing information for successful planning attempts:
test_convex_grid :: (us)
        Min :    166
        Mean:    184.5
        Max:     203
        Median:  0
        Std dev: 18.5
test_empty_grid :: (us)
        Min :    6062
        Mean:    9904.33
        Max:     15905
        Median:  15905
        Std dev: 4298.45
test_maze_grid :: (us)
        Min :    874
        Mean:    8325
        Max:     22451
        Median:  2260
        Std dev: 8547.19
test_narrow_constriction_grid :: (us)
        Min :    2667
        Mean:    3153
        Max:     3639
        Median:  0
        Std dev: 486
test_wide_constriction_grid :: (us)
        Min :    2944
        Mean:    3772.67
        Max:     5278
        Median:  5278
        Std dev: 1066.24
```

Fig. 11: Timing information for successful A* planning attempts

```
Timing information for failed planning attempts:
test_convex_grid :: (us)
        Min :    5
        Mean:    47.5
        Max:     90
        Median:  0
        Std dev: 42.5
test_empty_grid :: (us)
        Min :    6
        Mean:    21.5
        Max:     37
        Median:  0
        Std dev: 15.5
test_filled_grid :: (us)
        Min :    29
        Mean:    64.6
        Max:     95
        Median:  73
        Std dev: 26.5224
test_narrow_constriction_grid :: (us)
        Min :    10
        Mean:    1.769e+06
        Max:     5.30694e+06
        Median:  10
        Std dev: 2.5017e+06
test_wide_constriction_grid :: (us)
        Min :    128
        Mean:    128
        Max:     128
        Median:  0
        Std dev: 0
```

Fig. 12: Timing information for failed A* planning attempts

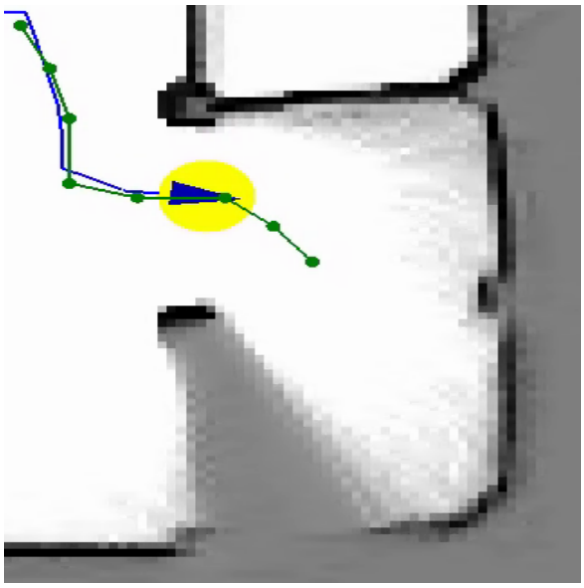


Fig. 13: The actual path driven by MBOT with a path planned using A* algorithm

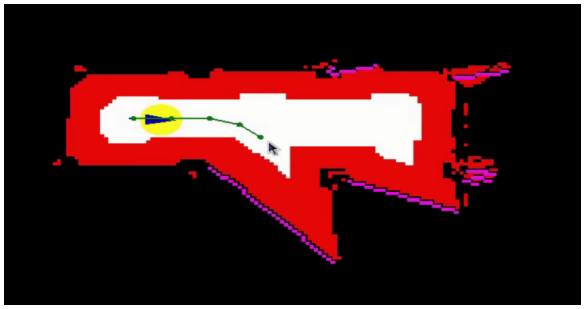


Fig. 14: Robot exploration in the botgui interface

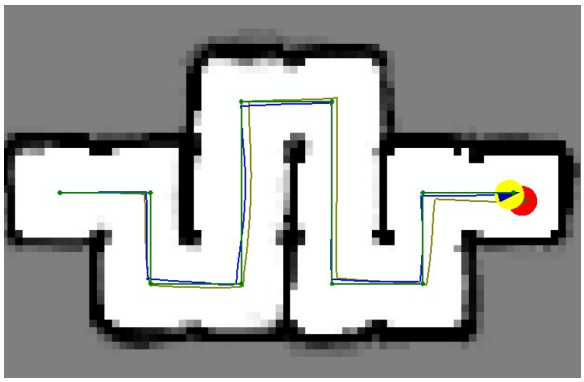


Fig. 15: SLAM in checkpoint 2 map

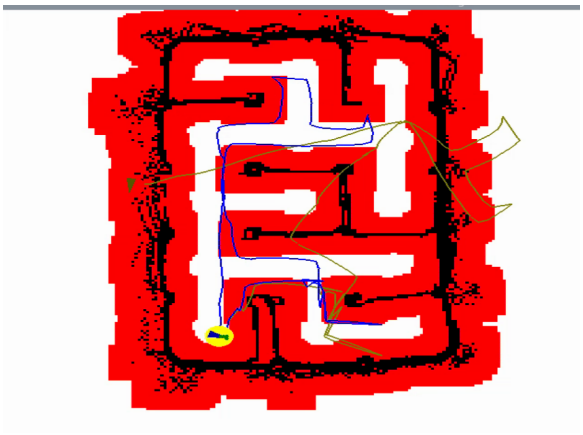


Fig. 16: Exploring performance in class competition

is exceeded, the program will return that it is impossible to find a path between the start and the end point.

b) Exploration: The experiments of exploration fails a lot without using the Robot feasible cells, because the A* planner cannot plan a valid path due to obstacles blocks all the possible ways from the starting cell to goal cell. Adding the robot feasible cells methods, the goal is always chosen to be feasible, which guarantees success in path planning while cost more time. This lead to latency in publishing a new path, and the robot usually has moved away from the starting waypoint of a path, then the robot turn around and go to a visited cell. This problem lowered the efficiency and makes the odometry not so smooth. A possible way to solve the problem is to remove the first one or two waypoints of each planned path. Figure 16 shows our result of exploring methods.

REFERENCES

[1] J. Borenstein and L. Feng, "Umbmark: A benchmark test for measuring odometry errors in mobile robots," in *Mobile Robots X*, vol. 2591. International Society for Optics and Photonics, 1995, pp. 113–124.

[2] T. Tao, Y. Huang, F. Sun, and T. Wang, "Motion planning for slam based on frontier exploration," in *2007 International Conference on Mechatronics and Automation*, 2007, pp. 2120–2125.

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

[4] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Wiley, 2005. [Online]. Available: https://books.google.com/books?id=wGapQAAACAAJ