Team 15 ROB 550 BotLab Report

Nam Gyu Kil, Tianyi Liu, Yingxue Wang

***Abstract*—Mobile robots are becoming increasingly capable and more prevalent thanks to the amount of computing resource a mobile platform can carry today. The mobile robots are capable of mapping out an unknown environment and localize itself within the map, then plan a trajectory to explore an unseen region of the environment. The purpose of the Botlab is to investigate the implementation detail of mobile robot algorithms and deliver software that can navigate the robot safely and accurately.**

I. Introduction

AUTONOMOUS mobile robots have been investigated extensively in the field of Robotics. To enable the mobile robot to navigate in an unknown environment, the robot needs to achieve position control, simultaneous mapping and localization (SLAM), and path planning. The team researched and developed algorithms to support the robot's control, SLAM, and planning aspects. The goal of the lab is to enable the robot to map an unknown environment, localize itself, and plan safe trajectories to explore an unvisited area on the map.

II. Methodology

A. Odometry and Controls

The task of this section is to derive the robot's global frame odometry information, and design a motor controller that is capable of driving to a given 3-Degrees of Freedom (Dof) pose in the global frame.

1) Odometry with Gyro Sensor Fusion: The odometry of the Mbot consists of global x, y coordinate, and θ angle of Mbot heading direction. Two wheel encoders and an IMU were used to obtain odometry data. With the encoder data and the dimension of the wheels, the distance traveled can be calculated. The change in x, y, and theta in each time step can be calculated:

$$\Delta d = \frac{\Delta s_R + \Delta s_L}{2} \tag{1}$$

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{b} \tag{2}$$

$$\Delta x = \Delta d cos(\theta + \Delta\theta/2) \tag{3}$$

$$\Delta y = \Delta d sin(\theta + \Delta\theta/2) \tag{4}$$

Where Δs_L is the change in displacement in the left wheel and Δs_R is the change in displacement in the right wheel. All angle calculations are clamped within $-\pi$ to π. With the change in each term, the odometry can be updated with:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} \tag{5}$$

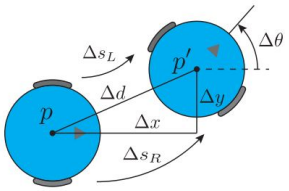

Fig. 1: Mbot wheel displacement and odometry

One of the disadvantages of only relying on encoders is error accumulation on the orientation readings when wheels slip. We incorporate the sensor reading obtained from the gyroscope and use the yaw angle to fix the orientation. During each iteration of pose update, if the difference d of the angles obtained from dead reckoning θ_1 and gyroscope θ_2 is greater than an experimentally chosen threshold T, we trust the gyroscope and use θ_2. Otherwise, we choose θ_1.

2) Control Architecture: The ultimate control design goal is to enable the robot to drive to a given pose $(x, y, \theta) \in SE(2)$. The control architecture of the Mbot was developed in three levels. On the low-level, a wheel speed controller takes desired individual wheel speed input and outputs PWM to each wheel motor. On the mid-level, a robot frame velocity controller receives a desired robot forward and turning velocity and computes required individual wheel velocity. On the high-level, a motion controller receives target pose information and commands a robot forward and turn velocity.

a) Motion Controller: The motion controller receives a series of target poses $(x, y, \theta) \in SE(2)$ as a discretized representation of a trajectory. The controller needs to drive to each target pose and assign itself to the next pose in order to follow the trajectory. Given a target pose and current pose of the robot, a feedback control algorithm was implemented as the motion controller. Fig. 8 illustrates the relationship between target pose and current robot pose in polar coordinates with the target as the origin. ρ is the euclidean distance between the robot and the target. θ is the angle between the robot's heading direction and the target heading direction. α is the angle

between robot's heading direction and the vector of ρ. β is the angle between vector of ρ and target heading direction.

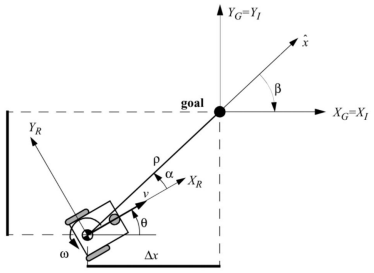

Fig. 2: Mbot kinematics in $SE(2)$

Driving the robot to the target pose is equivalent to controlling the system from $[\rho, \alpha, \beta]$ to [0, 0, 0]. The robot uses the control law, where V is the robot forward velocity and ω is the turn velocity:

$$V = k_\rho \rho \tag{6}$$

$$\omega = k_\alpha \alpha + k_\beta \beta \tag{7}$$

The V and ω are passed down to the robot frame velocity controller. Once the pose error value is within a certain threshold, the robot is considered to have reached the target pose.

b) Robot Frame Velocity Controller: The function of the robot frame velocity controller is to convert desired robot forward velocity V and the turn velocity ω to left and right wheel velocity v_L and v_R. Using the same differential drive motion model with wheelbase b, it can be obtained:

$$v_L = V - \frac{1}{2}\omega b \tag{8}$$

$$v_R = V + \frac{1}{2}\omega b \tag{9}$$

The v_L and v_R values are then passed down to the wheel velocity controller.

c) Wheel Speed Controller: Fig. 3 demonstrates the layout of the wheel speed controller. A feed-forward controller that maps the wheel speeds to the according PWM value under no load was implemented. To reject any disturbance or change in resistance, A feed-back controller was added to the control loop.

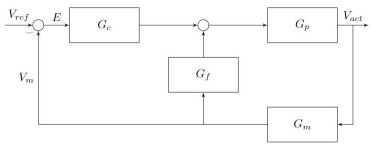

Fig. 3: Wheel speed controller block diagram

B. Simultaneous Localization and Mapping (SLAM)

The SLAM algorithm for the botlab is comprised of the following sub-modules: localization and mapping. Fig(4 shows a diagram of the interaction of the sub-modules with the inputs (wheel encoder, IMU, and lidar).

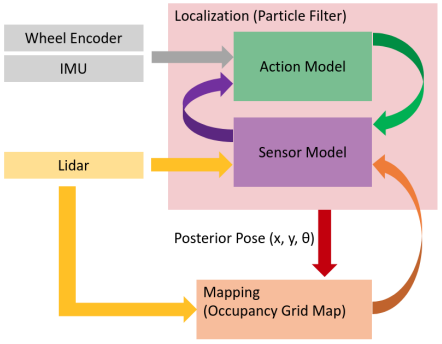

Fig. 4: Block diagram of how the SLAM system

The localization sub-module is composed of two components, the action model and sensor model, which is explained in the sections below. Essentially, these two components first disperse our probability distribution and resample the distribution to obtain a better pose estimate. The pose estimate then gets sent to the mapping sub-module which will map the environment based on the pose estimate and lidar scans received.

1) Mapping: The general mapping task is to compute the most likely map given sensor data and position of the robot and can be solved:

$$m^* = argmax_m P(m|x_{1:t}, z_{1:t}) \tag{10}$$

Because our robot uses a 2D sensor, our choice of map was the occupancy grid map, which is a M x N matrix of cells. Each cell is either occupied or unoccupied and is represented as a binary random variable that models this occupancy. $p(m_i) = 1$ represents occupied cell, $p(m_i) = 0$ represents unoccupied cell, and $p(m_i) = 0.5$ represents unknown cell. Aside from the assumption that the map can be represented as a binary state, two additional simplifying assumption of the environments are made. One is that the environment is static and the other that the cells are independent of one another.
Therefore with these assumptions, the joint probability of the map can be represented as the product of all the marginal probabilities of each cell being occupied given the position of the robot and sensor traces.

$$p(m|x_{1:t}, z_{1:t}) = \prod_i p(m_i|x_{1:t}, z_{1:t}) \tag{11}$$

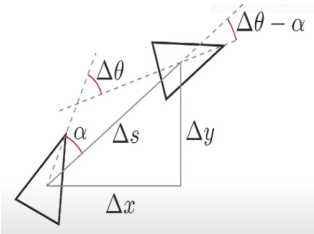

Fig. 5: Figure illustrating the action for robot to reach next pose

Using the concept of odds ratio into the probability and using the log-odds notation we get the following log-odds equation

$$l_{t,i} = invsensormodel(m_i, x_t, z_t) + l_{t-1,i} - l_0 \quad (12)$$

Eq(12), which refers to the new state of a cell at i and a time t, is made up of the inverse sensor model, the recursive term, and the prior in that order.

2) Monte Carlo Localization: The localization sub-module is a recursive state estimator in the form of the particle filter which has two components. The first is the action model which will propagate the particles based on the wheel encoders and IMU, increasing the probability distribution of our state. The second component is the sensor model which will take the lidar scan at current time t and output a score based on how well it fits with the current map. The better the fit, the higher the weight of a particle. A new pose with a lower probability distribution of our state will be calculated from these particles and the new pose will be used to map the environment with the lidar scans. Simultaneously, the new pose will also recursively enter back into the action model for the particles to be propagated and these particles to the sensor model.

a) Action Model: The purpose of the action model is to look at the certainty of the state given a certain action. Given our odometry readings, x, y, and θ, the action to propagate our distribution, α, Δs, $\Delta\theta$ - α can be derived by the following equations:

$$\alpha = atan2(\Delta y, \Delta x) - \theta_{t-1} \quad (13)$$

$$\Delta s^2 = \Delta x^2 + \Delta y^2 \quad (14)$$

$$\Delta\theta - \alpha = (\theta_t - \theta_{t-1}) - \alpha \quad (15)$$

As it can be seen in figure above, the parameters used for the action model is simply modeling the rotation, translation, and rotation of the robot as it reaches the next pose. To model the uncertainty of the action with errors we introduce some error terms ϵ_1, ϵ_2, and ϵ_3:

$$\epsilon_1 \sim \mathcal{N}(0, k_1|\alpha|) \quad (16)$$

$$\epsilon_2 \sim \mathcal{N}(0, k_2|\Delta s|) \quad (17)$$

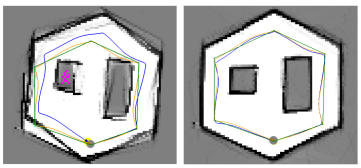

Fig. 6: Side by side comparison with low values of ks and higher value of ks

$$\epsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|) \quad (18)$$

Putting all the equations together, we get the following action model:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} (\Delta s + \epsilon_2)cos(\theta_{t-1} + \alpha + \epsilon_1) \\ (\Delta s + \epsilon_2)sin(\theta_{t-1} + \alpha + \epsilon_1) \\ \Delta\theta + \epsilon_1 + \epsilon_3 \end{bmatrix} \quad (19)$$

Through many iterations of testing the overall localization algorithm on the robot, we find the following values of k_1 and k_2 in table I.

TABLE I: Final values of k_1 and k_2

k_1	k_2
1.25	0.5

Fig. 6 shows the SLAM system working with different values of k_1 and k_2. Initially, both k_1 and k_2 were tuned to extremely low value (0.075 and 0.01 respectively) as a low value of k_1 would give a lower noise value associated with rotation and k_2 for displacement. However, though this worked for straight paths, when the robot would rotate, these values would fail and the localization as well as the mapping would incorrect. This was due to the fact that the particles were not being propagated enough for the distribution to capture the actual motion of the robot. Therefore, in order to capture both straight line and rotation movements for localization, a higher value was needed, especially for k_1.

b) Sensor Model: The purpose of the sensor model is to find the probability that a certain particle's pose and lidar scan matches the map. The sensor model gets as input the particles that have been propagated by the action model and scores each particle based on its fit with the map. Rather than using the beam model which has an ensemble form composed of the Gaussian, exponential, and two uniform distribution and ray casting to get a probability, we used the simplified likelihood field model. This was the method of our choice because ray casting is much more computationally expensive when comparing to a method that only checks the endpoints of the ray. For all the rays in a ray scan, if the endpoints of the ray lie where an obstacle is in the map, the score for the particle is incremented by the log odds of the map cell. These scores are then assigned as weights to an individual particle. After normalizing the weights,

the particles are then re-sampled. Essentially this is the survival of the fittest strategy where particles with higher weights are kept in the distribution as we recursively call the whole particle filter.

C. Planning and Exploration

The goal of this section of the lab is to allow the robot to autonomously explore an unknown environment using $A*$ path planning algorithm, draw a map using the previously described SLAM algorithm, and localize itself from an unknown location given knowledge of the map. This section explains our implementation in detail.

1) A^* Path Planning: The A^* algorithm is an informed search algorithm formulated in terms of weighed graphs, which in our case, is the obtained 8-way connectivity SLAM occupancy grid map. The mbot is able to plan path not-only in the up, down, left, right directions, but also along the diagonal directions of the grid. This will output a smooth path to the motion controller.

Starting from the initial location, A^* determines which grid cell of its path to extend to based on the true cost of the current path, $g(n)$, and an estimate of cost required to reach the goal, $h(n)$, where n represent the current grid cell. A^* then use a best-first search to select the path that minimizes the f score,

$$f(n) = g(n) + h(n) \tag{20}$$

This is done by maintaining all the expanded node in a priority queue, in which the node with lowest f score is positioned at the front and will be pop out and expanded first. The algorithm maintains a tree of paths originating from the start node and extends the paths until the termination criterion is satisfied, which in this case a target location is reached.

The pseudo-code of our implementation is provided here:

Algorithm 1: A*

```
Input: start, goal(n), h(n), expand(n)
Output: path
if goal(start) = true then return  makePath(start);
open ← start
closed ← ∅
while open ≠ ∅ do
    sort(open)
    n open.pop()
    neighbors ← expand(n)
    for neighbor ∈ neighbors do
        neighbor.f ← (n.g + 1) + h(neighbor)
        If goal(neighbor) = true then
        Return makePath(neighbor)
        If neighbor ∉ closed
        then open ← neighbor
    end
    closed ← n
end
Return ∅
```

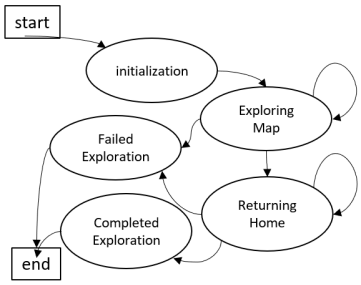

Fig. 7: Finite State Machine for Exploration

a) Incorporating Obstacle Distance to $A*$: The optimal paths obtained from path-planning algorithms such as A^* tends to follow obstacles and the uncertainties in robot position and map pose a risk of robot traveling too close or running into them. Therefore, we incorporated the distance from each grid cell to nearest obstacle into the g_score of A^* to penalize the path that is too close to the wall. This is done by adding the following term to the g_score equation for each cell in the grid:

$$\beta[(d\_max - d)^\gamma] \tag{21}$$

where d_max is the maximum distance with cost, d is the distance of current grid cell to the nearest obstacle, and γ is the Distance Cost Exponent. We multiply the term with a constant β as a hyper-parameter to adjust the amount of distance penalty we want to incorporate.

2) Map Exploration: We implemented a frontier exploration algorithms that utilizes A^* to autonomously navigate the environment and plot a map using SLAM. Frontiers are defined as the boundaries between the free space and explored space. The code for finding the frontiers were provided to us, returns a list of frontier clusters.

The exploration algorithm is implemented by a Finite State Machine(FSM) as shown in Figure 7.

a) Initialization state: The robot stays in the idle state until it receives an empty slam map and enters the $Initialization$ state to initialize the required state variables such as setting time-stamps and home pose.

b) Exploring Map state: For each iteration of the exploration state, we find the centroids of all frontiers and plan to the closest one using $A*$. However, since the centroid of a frontier is not guaranteed to fall onto navigable regions in the map, we implemented a Breadth-First-Search algorithm, $MakeValidGoal()$, to search within the goal cell's neighbors to find the closest valid cell and set it as the $A*$ target. A Valid goal has to satisfy the following requirements:

- The cell is inside of the grad map.
- The cell is free, i.e. known and not occupied.
- The cell is at least one robot-radius away from an obstacle or wall.

Moreover, since we do not need to have the robot proceed all the way to the target frontier centroid because the area will normally be fully explored before that. We therefore set a large distance margin so the path following algorithm is terminated early and the robot will be given a new path to follow. The mbot exits the $Exploring\ Map$ state when there is no frontier left to explore, i.e. we have complete knowledge of the known environment.

c) Returning Home state: The robot state changes to $Returning\ Home$ once the exploration is done. It uses $A*$ to plan to the initially set home pose, which should the the origin $(0, 0)$. In case the home location is non-reachable or too close to the wall, we apply the aforementioned $MakeValidGoal()$ function to plan to the closest valid cell to the home position. Once the robot arrives within a distance threshold to the home location, the robot exits the $Returning\ Home$ state.

d) Completed Exploration and Failed Exploration state: The mbot enters the $Completed\ Exploration$ state when it successfully returned home and exits the FSM. If the mbot encounters any issue such as failing to plan a valid path to a target cell during the $Exploring\ Map$ or $Returning\ Home$ state, it enters the $Failed\ Exploration$ state and exits the FSM.

3) Map Localization with Unknown Starting Position: The famous "kidnapped robot problem" is commonly used to test a robot's ability to recover from catastrophic localization failures. To solve the problem, we made adjustments on the SLAM algorithm and particle filters, and used the percentage of scan-matches to determine if the robot has correctly localized itself.

We first start with distributing the particles about an uniform distribution across the whole map. This is different then distributing about a normal distribution around the robot pose for our normal SLAM algorithm. Each individual particle is then propagated using the action model and resampled in the sensor model of the localizaton sub-module. To gather more information in the surrounding environment, we defined a $RandomWalker()$ function that takes in the odometry pose and set waypoints in nearby regions safely. The way points are assigned to move away from the closest wall and towards the furthest open space based on lidar scan. When the variance of particles' (x, y) falls within a certain threshold the robot will determine that it has localized itself.

However, the robot may localize to the wrong location due to the symmetric nature of the environment. To ensure that the localization result is correct, we check the ends of each lidar scan ray about the localized pose and compare that to the occupancy map. We count the number of correct rays and divide it by the total number of rays to get a percentage. If the percentage is greater than a threshold, we determine that the localization has succeeded. Otherwise, we re-distribute the particles across the whole map and re-propagate them until it correctly localizes itself.

TABLE II: Wheel velocity controller parameters

K_p	K_i	K_d	Tf (s)
1	0.97	0.01	0.08

III. Results

A. Odometry and Controls

1) Odometry: The calibration and error measurement for odometry was done by manually moving the robot forward for 50 cm and compare with odometry reading from Mbot. The same procedure was done for turning $\pi/2rad$. The error turns out to be within 5 cm for linear and 0.06 rad for a total of 5 test runs.

2) Control Architecture:

a) Wheel Speed Controller: Although the same control logic described in Methodology applies to all team robtos, The wheel speed controllers tuning differ slightly between each. The rest of the discussion will be based on Mbot-1508 tuning. The team implemented the PID controller from Robot control library, which has a low-pass filter integrated into its derivative term. The final tuning gains and low-pass filter parameters are listed in TABLE II.

The controller design goal was to achieve smooth wheel speed response, eliminate oscillation, and drive at low velocity with load. With the help of the feed-forward controller, the system's response time was rapid. The K_p gain was designed to be low to reduce steady-state oscillation. The K_i gain was set high to correct for steady-state error especially at low speed, where the motor PWM response is non-linear due to DC motor dead zone. Thanks to the high K_i gain, the controller is able to achieve a constant minimum wheel velocity of 0.01 m/s. The fastest speed was limited by motor power to 1 m/s. The step response of the wheel velocity controller to a 0.5 m/s input is shown:

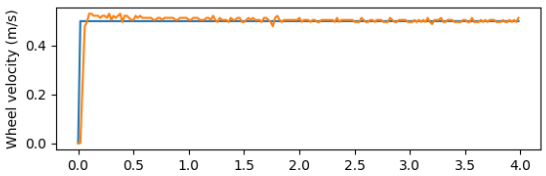

Fig. 8: Wheel controller response to 0.5 m/s step input

b) Robot Frame Velocity Controller: Being only an open-loop controller, the robot frame velocity controller behavior was very predictable. Fig. 9 illustrates the performance of the robot frame velocity controller in driving

TABLE III: Motion controller parameters

K_α	K_ρ	K_β
3	1.5	-0.5

forward and turning. The desired forward velocity was set to $0.25m/s$, $0.5m/s$, and $1m/s$ and desired turning velocity $\pi/8rad/s$, $\pi/2rad/s$, and $\pi rad/s$ from top to bottom. The change in robot's world frame x location was very linear with slight overshoot. This overshoot could be caused by accumulation of velocity error in wheel speed controller.

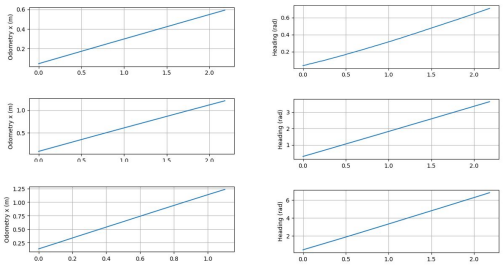

Fig. 9: Odometry x and θ at different turn velocity setpoints

c) Motion Controller: The motion controller was tuned with gains shown below. The controller was able to correct error in α and ρ term well, driving the robot to its target x, y coordinate within certain threshold. However, the β control only performs well with high $k_\alpha, k_\rho, andk_\beta$To ensure optimal SLAM results, the team kept k_β to a low gain and added a constant speed turn control logic when the robot has reached its x, y coordinate.

Fig. 10 shows the motion controller's behavior when commanding the robot to follow a square trajectory for four laps. The controller was able to follow most way points within threshold, yet it struggles when target is at π, as seen on the upper edge of the square trajectory. Fig. 11 displays the linear and angular velocity commands VS time in a 1 m forward, πrad turn, then 1 m forward trajectory. The change in linear velocity was as expected overall. The angular velocity had some unwanted fluctuation even when the robot is driving straight without a change in $beta$. The team believe that this issue only occurs at $\theta = \pi$ and could be caused by inconsistency in Euler angle wrap methods used in the code.

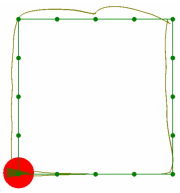

Fig. 10: Change in pose throughout a square trajectory

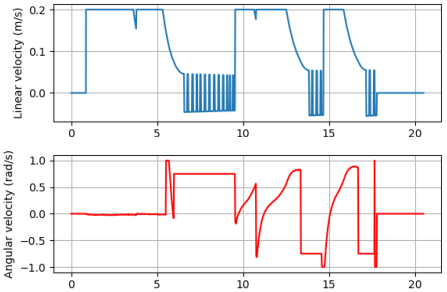

Fig. 11: Speed command in a forward-turn-forward maneuver

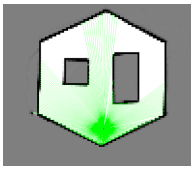

Fig. 12: Example plot of the Occupancy Grid Map

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: Using the occupancy grid to map the environment, the quality of the map heavily depended on several key factors. One was the resolution of the map which was defaulted to 0.05 meters. Fig. 12 is an example of the occupancy grid map generated for an obstacle course with the default resolution.

Another key factor in the quality of the map was also accuracy of the localization. Fig. 12 shows some blurs and shadows of edges that are non-existent in the actual environment, but mapped because of some error in the pose. Throughout multiple iterations of our slam system and mapping, the error associated with the orientation of the robot had the most drastic affect in the quality and accuracy of our maps.

2) Monte Carlo Localization: Fig 13 shows the 300 particles as it goes through the action model and sensor model of the particle filter. The layout of the particles correspond to the position of the robot as it navigates a 1m x 1m route starting from the bottom left corner and traveling counter clockwise. The particles that are plotted in the corner of the figures are the state of particles after applying the sensor model and the particles that are midpoint between the corners are the state of particles after the propagation from the action model.

a) Action Model: Looking at the plots in the midpoints of each 1 meter segment in fig 13, we can see the distribution of the particles to be much more spread out then its previous corner plots. This is due to the fact that we have modeled error for rotation and translation based on the action of the robot at current timestep t. A higher action term, in this example a Δs, would introduce more

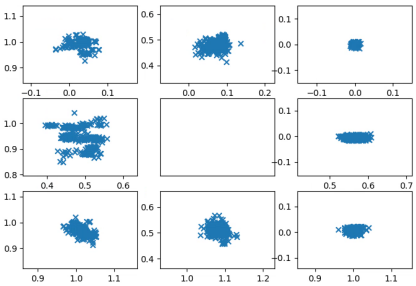

Fig. 13: 300 particles at midpoint of 1m transition and at corners.

TABLE IV: Computation speed for one iteration for varying particle numbers

Number of particles	Update in seconds
100	0.035
300	0.107
500	0.157
1000	0.321

noise to the system which disperses the distribution as can be seen in fig 13.

b) Sensor Model: The sensor model on the other hand, if implemented correctly, should take the propagated particles and resample the particles to make the distribution of the particles smaller as particles with higher weights are picked out. This can be seen again from fig 13 in the corner plots where the distribution are less spread out then the particles in the midpoint sections.

c) Particle Filter: Though the particle filter is very useful in representing distributions of the state of the system and more particles can result in more precise distributions and calculations for localization, the system is limited by the computation resources available. Table IV shows average time it takes for one iteration of the particle filter for varying numbers of particles.

As we can from the table IV, more particles have a significant impact on the computation resources used. Our robot, which runs at 10 Hz for SLAM can support around 300 particles to run in real-time. Any more particles will not allow the SLAM to run efficiently at 10 Hz.

3) Combined Implementation: The trajectories and RMSE of the pose from Odometry and SLAM from the implementation of the full SLAM system are shown in the fig 14. As it can be seen in fig 14, the trajectories of a pure odometry is very different then that of SLAM. Though the odometry performs adequately in straight sections, once rotation is introduced, the pose deviates from the actual ground truth. The RMSE error shows error in pose that accumulates over time and in the final pose, have a error around 0.1 meters in both x and y direction as well as an orientation error of 0.2 radians.

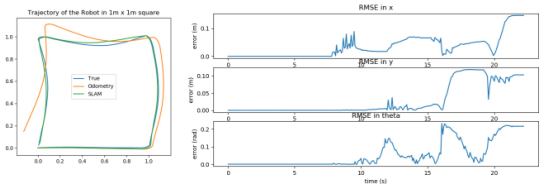

Fig. 14: Comparison of Trajectories and RMSE between Odometry and SLAM pose

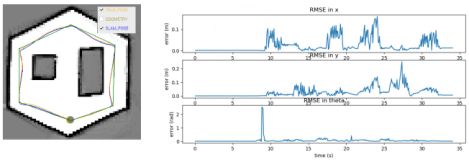

Fig. 15: Comparison of Trajectories and RMSE between Ground Truth and SLAM pose

The error that accumulates from odometry pose is mainly due to the slip of the wheel and the low precision in the encoder. The SLAM system incorporates the external and internal noise by incorporating lidar for the sensor model as well as modeling noise through the use of particles and propagation in the action model.

To evaluate the performance of our SLAM system, the figure 15 also shows a comparison between the groundtruth and SLAM pose in an environment with obstacles. Fig 15 shows the precision of our SLAM system as we can see the trajectory from SLAM fits nicely on top of the ground truth. The RMSE plot shows that though there is some error for x, y, and θ, the error in the final pose is around 0 for all states. The mean error for the state of the pose is shown in the table V.

TABLE V: Mean RMSE for different pose states

Pose state	mean RMSE
x	0.039
y	0.043
θ	0.157

The errors that are apparent in the x and y states can be attributed to the latency of the SLAM system when outputting the SLAM pose. Because our particle filter has 300 particles and uses a stride of 2 for the sensor model, the output of the slam pose is not in real time. Additionally, the error in the θ of 2 radians is due to the start of the localization as the robot begins to move. The sensor model does not run while the robot is not moving, as a result when the robot starts to move, the orientation error is expected to be large as it takes the average orientation of all particles.

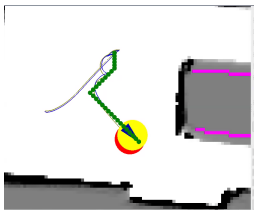

Fig. 16: Planned A^* path (green) versus actual travel path(blue).

TABLE VI: . Timing Statistics for A^*. In $1e^3$ us.

Test Name	Min	Mean	Max	Med.	Std.
$convex\_grid$	10.7	10.9	11.1	11.1	0.2
$empty\_grid$	21.0	33.2	50.5	28.0	12.6
$maze\_grid$	9.5	11.1	13.9	13.9	1.8
$narrow$	22.2	37.1	58.6	58.6	15.6
$wide$	23.3	29.4	34.4	34.4	4.6

C. Planning and Exploration

1) Path Planning:

a) Planned A^* Path Versus Actual Traveled Path: Figure 16 shows the planned A^* path (green dotted path) with the actual SLAM and Odometry path overlayed on top. In figure 16, the mbot tried to follow the planned path from top to bottom. Since the distances to obstacles play an important role when calculating the g_score in A^*, it prefers a path that goes through the center of the free space. Moreover, in cases of large change in angle, our robot chose to move backward while turning, as indicated in the spiral shape of the actual path.

b) Timing Statistics for A^* Tests: The time statistics (in $1e^3$ us) of our robot processing with the succeeded tests in $astar\_test\_files$ is shown in table VI.

Our A^* algorithm passed 4 out of 6 tests in the $astar\_test\_files$. It failed some of the test cases in the $test\_narrow\_constriction\_grid$ and $test\_convex\_grid$. Our A^* algorithm performs relatively efficient, as it usually takes e^5 us to run, which is around $1e^4$ Hz, which is negligible comparing to the $10Hz$ slam update frequency.

We observed that in the scenario with narrow gird width, the Astar was able to plan a path to navigate though the area that was marked "dangerous". However, as we have tested out in the real world cases, the $A*$ never plan any dangerous or invalid path, we have reason to question that there might be some unresolved issues with code that draws out the boundaries between the safe-zones and danger-zones. This issues also affects us in the exploration process.

2) Exploration: Using the strategy described in the method section, our robot can successfully explore a large map and return home. During the testing, we realized that a high grid resolution (around 0.1 or 0.2) can output a very clean map and will not cost too much computational power. Figure 17 shows a map plotted during the task 2 of the competition with grid resolution 0.2.

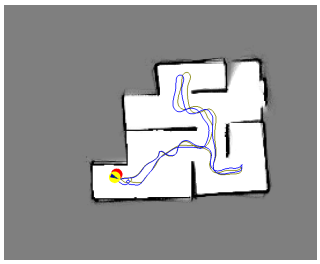

Fig. 17: Maze map obtained by our exploration algorithm during task 2 of the competition.

3) Localization: Using the strategy described in the method section, the robot was able to successfully localize itself when positioned at an unknown location provided knowledge of the map. A threshold for scan-matching percentage to 90% was enforced to guarantee a correct localization result. If this threshold was not met, the robot would then re-initialize the particle using the uniform distribution mentioned earlier.

IV. Discussion

Our mbots were able to fulfill every required tasks efficiently and precisely. Our A^* algorithm is robust, as it always outputs the shortest path to the target location. Our exploration algorithm is efficient, as we chose an eager strategy to always move to the closest frontier and updates the path frequently. Our SLAM algorithm is reliable and output accurate pose estimates. Last but not least, our localization algorithm is very accurate.

However, we did encounter some challenges while implementing the algorithms, and there are some potential future works that could be done. For example, the controller sometimes encounters some random oscillations when the target orientation is π, which may be resulted by our angle-wrapping strategy. We could also improve on the way that we are incorporating the obstacle distances, so we will have a better trade-off between path efficiency and safety. Moreover, we could improve the particle filters by filtering out the outliers before calculating the posterior distribution. This will better handle the multimodal distribution and alleviates shifts of the map.

References

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

[2] I. S. D. Siegwart, R. Nourbakhsh, *Introdution to Autonomous Mobile Robots*. The MIT Press, 2011.