# Section 1 Team 8 ROB 550 BotLab Report

Daniel Simmons, Yu-Ju Chiu, Manav Prabhakar, Ruiyang Wang

***Abstract*—The problem of motion planning for autonomous navigation is a core robotics challenge that has been and continues to derive new methodologies and algorithms for computationally efficient and real-time solutions. We present our methodologies and results for a motion planning task that focused primarily on reaching a fixed goal from a given start position and exploring unknown cluttered environments while following a collision free path.**

## I. Introduction

THE objectives and constraints associated with any motion planning task varies on its intended application. One of the major applications of these methodologies lies in the domain of non-holonomic robots in an industrial setup. The use of these robots is being widely considered in warehouses for selection, sorting or movement of items. Even self-driving cars would be an example of a non-holonomic setup. Researchers have been experimenting with different sets of sensors to ensure a fool-proof system including but not limited to cameras, 1-D LIDAR, 2-D LIDAR, Infrared, etc. Generally, vision-based approaches are augmented with multi – view cameras in order to compensate loss of the depth information, an essential quantity for perceiving the distance of objects in each scene. LIDAR act effectively in this aspect, with even one-dimensional LIDAR providing distance to the objects. This provides the robot with an idea of the position, size and shape of the object when perceiving its environment.
This paper largely focuses on our approach to targeting three problems

*a)* ***Waypoint Traversal****:* can be defined as controlling the MBot to move between any given waypoints with the help of its motion planner and odometric data.

*b)* ***Directed SLAM****:* is predicting current position with respect to random variation.

*c)* ***Exploratory SLAM****:* is mapping any unknown cluttered environment in the form of a 2-D occupancy grid. These problems if solved in the hierarchy of their mention can lead to an important ladder for building an autonomous system. An overview for the entire system has been given in Figure 1. The final outcome of the system is to build an occupancy grid. The nature of all these subsystems and the occupancy grid have been explained in detail in the subsequent sections.

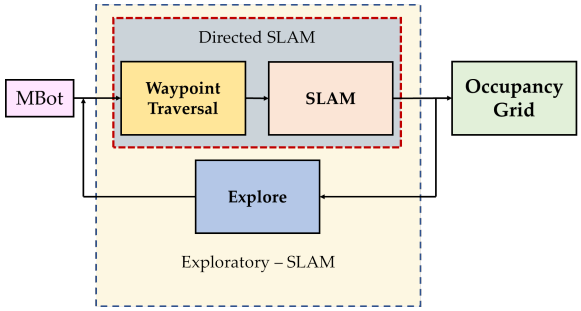


Fig. 1: Waypoint Traversal, SLAM and Explore modules working together on the MBot to produce an Occupancy Grid

We perform our experiments on a small non-holonomic robot, powered by a two-wheeled differential drive with a castor wheel at the rear and supported by a one-dimensional lidar for building its perception space. We call this robot as the MBot.

## II. Methodology

We will introduce our several models for achieving our goal, doing exploration in the unknown area.

### A. Controller

In this section, we will discuss the basic control methods of our motors, including motor calibration, motor speed controller, odometry model, PID for speed controller, and motion controller method.

*a) Wheel Speed Controllers:* The MBot works on the principle of differential drive. The highest level of abstraction for any autonomous vehicle to function in a cluttered environment would be the ability to move it based on its velocity vector. It is thus imperative that we define our wheel speed controls and encoders considering the translational and rotational velocities. Let $V_{trans}$ and $V_{rot}$ be the desired translational and rotational velocities. We then determine the left wheel speed ($V_{left}^{wheel}$) and the right wheel speed ($V_{right}^{wheel}$) using relative velocities taking in consideration the differential drive.

$$V_{left}^{wheel} = V_{trans} - (W_{base} \times V_{rot})/2 \quad (1)$$

$$V_{right}^{wheel} = V_{trans} + (W_{base} \times V_{rot})/2 \quad (2)$$

Once we have computed the left and right wheel velocities, we need to encode these velocities into motor commands For encoding these values, we first determine

the left ($V_{cmd}^{left}$) and right velocity commands ($V_{cmd}^{right}$) using a PID controller accompanied with a low pass filter. We further consider 6 calibration parameters

| Left Wheel | Value | Right Wheel | Value |
|---|---|---|---|
| FW_SLOPE ($m_f^l$) | 1.13 | FW_SLOPE ($m_f^r$) | 0.93 |
| FW_INTERCEPT ($c_f^l$) | -0.19 | FW_INTERCEPT ($c_f^r$) | -0.14 |
| BW_SLOPE ($m_b^l$) | 1.02 | BW_SLOPE ($m_b^r$) | 1.09 |
| BW_INTERCEPT ($c_b^l$) | 0.12 | BW_INTERCEPT ($c_b^r$) | 0.10 |

TABLE I: PID value for different controller

We obtain these values using a series of experiments computing speed and current values vs the duty. The details for the experiments and their respective plots can be found in Appendix. Using the calibration values from Table I, we compute and clamp the left ($D_{left}$) and right ($D_{right}$) motor duty to -1 and 1. In case we have a forward velocity, the values can be calculated using the equations 3 and 4.

$$D_{left} = Clamp\left[\frac{V_{cmd}^{left} - c_f^l}{m_f^l}\right] \tag{3}$$

$$D_{right} = Clamp\left[\frac{V_{cmd}^{right} - c_f^r}{m_f^r}\right] \tag{4}$$

We can compute the commands for backward velocities plugging the appropriate values from Table I in equations 3 and 4. We can then move on to compute the corresponding left and right motor commands.

*b) Odometry Model:* The speed of MBot is determined using magnetic encoders on the motors to measure their rotation. The odometry compares the rotation of one wheel to the rotation of the other. When the MBot is initialized the coordinate system is set to the origin of the map. The odometers on each motor returns the number of rotations each wheel underwent in a certain period of time. These values are averaged together and converted to meters to find the translational distance traveled by MBot, while the difference between them is utilized to calculate the change in angle. The global coordinate and rotational orientation of MBot are then updated with these values. A gyroscope can be used to supplement inaccuracy in the odometry by overriding turns with a large enough discrepancy between them. This allows for orientational errors caused by environmental factors, such as a groove or wheel slippage, to be detected. Therefore, gyroscopic data is beneficial for performing dead reckoning and enhancing pose estimate for the Mbot.

We use the encoder values and Tait Bryan values to compute perform pose estimation. We obtain the difference between the previous and the current encoder values (in ticks) for the left ($\Delta E_l(t)$)and right wheels $\Delta E_r(t)$.

We define $\Delta\theta$ as

$$\Delta\theta = \Delta E_r(t) - \Delta E_l(t) \tag{5}$$

The values from the gyroscope help us to determine if the wheels underwent slipping, resulting in faulty odometry.

Thus, we need to incorporate the gyroscopic values when there is a considerable disparity between the odometry and the gyroscopic values.

$$\Delta D = \frac{\Delta E_r(t) + \Delta E_l(t)}{2} \tag{6}$$

We set the gyro threshold ($G_{thresh}(t)$) to be 0.1 (determined experimentally) Let the orientation of the MBot be denoted by $G_{theta}(t)$. Now,

$$\Delta\theta_{cor} = \frac{(\Delta\theta + G_\theta) + \alpha(\Delta\theta - G_\theta)}{2} \tag{7}$$

where

$$\alpha = \frac{\left|\Delta\theta - G_\theta\right| - G_{thresh}}{\left|\left|\Delta\theta - G_\theta\right| - G_{thresh}\right|} \tag{8}$$

We now compute the updated pose ($X_{new}$,$Y_{new}$,$\theta_{new}$) of the Mbot using the values from equations 6, 7 and 8 and the current pose ($X_{curr}$,$Y_{curr}$,$\theta_{curr}$)

$$X_{new} = X_{curr} + \Delta D\left[1 + \cos\left(\theta_{curr} + \frac{\Delta\theta_{cor}}{2}\right)\right] \tag{9}$$

$$Y_{new} = Y_{curr} + \Delta D\left[1 + \sin\left(\theta_{curr} + \frac{\Delta\theta_{cor}}{2}\right)\right] \tag{10}$$

$$\theta_{new} = \theta_{curr} + \Delta\theta_{cor} \tag{11}$$

With $\theta_{new}$ being clamped between $-\pi$ and $\pi$.

*c) PID:* After we complete the calibration, we use PID control to adjust the speed. We have two layers of PIDs. The first layer of PID is in the $motioncontroller.cpp$, which calculates the target pose and current pose into forward velocity and turning velocity. The formula is as follows.

Then, we have

$$V = K_p M(t) + K_i \sum_{i=1}^{t} M_i(t) + K_d \Delta M(t) \tag{12}$$

where, $\Delta M(t) = M(t) - M(t-1)$

We can compute $V_{fwd}$ and $V_{turn}$ as the forward and turn velocities respectively using Eq 12. $M(t)$ denotes the relative distance or angle between the current pose and target pose at time t.

The second layer PID make MBot close to this speed and ensures that our robot can go straight ahead and turn correctly. We use a low-pass filter first to do denoise, then do PID control. The reference speed command is obtained by the motion controller, and we get the current

velocity from the encoder. Finally, we calculate the speed difference, the sum of the speed difference, and the error difference to multiply the PID gain value respectively to obtain the final input motor velocity. Moreover, we set a topic in $LCM$ channel for the PID value so that we can extract the PID gain value in real-time by re-publishing the gain valuet Table[II].

*d) Motion Controller:* We use the $RSR$ motion control method. This method first turns the MBot in the direction of the target and then moves straight forward to the target position. If necessary, the MBot would then turn again to the given target angle. The steering angle can be obtained by differentiating the current and goal coordinates and finding the difference from the current theta. The MBot rotates until it is within to an error of 0.005 rad, then proceeds forward until within 0.02m of the target coordinates. Finally, the MBot turns to the target theta to arrive at the goal position. Moreover, forward movement and steering are both done by the PID controller(for the first layer)Fig[2], Table[II]..

| PID | Kp | Ki | Kd |
|---|---|---|---|
| PID for first layer(fwd) | 1.0 | 0.5 | 0.02 |
| PID for first layer(turn) | 3.0 | 0.2 | 0.02 |
| PID for second layer(left speed) | 2.0 | 10.0 | 0.1 |
| PID for second layer(right speed) | 2.0 | 10.0 | 0.1 |

TABLE II: PID value for different controller

### B. Simultaneous Localization and Mapping (SLAM)

In our SLAM implementation, we do two parts, one is for mapping and the other one is for localization.

*1) Mapping:* For the mapping part the map is consistently updated using laser scans that have been interpolated to account for robot movement. we update the log odds value of each cell in the map in mapping.cpp. The method increases the value of the cell at which the laser terminates with a constant, $kHitOdds$, and decrease the value of the cell which the laser passes through with another constant, $kMissOdds$ Fig[3]. We use Bresenham's algorithm to plot the laser path.

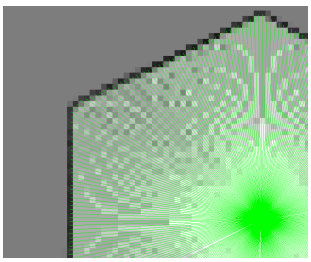

Fig. 3: The map is updating, the cell value is [-128, 127](white, black)

*2) Monte Carlo Localization:* Another important component of our SLAM system is localization. We achieved the Monte Carlo localization using three parts: action model, sensor model, and particle filter. In this section, we will first explain each part of our localization algorithm, and we will explain how the components of SLAM system interacts with each other in the next section.

*a) Action Model:* The first part of the localization algorithm is the action model. It predicts the pose of the MBot by representing a possible location as a particle. The particle is found by applying a standard deviation error to each movement the robot takes. The current odometry pose is compared to the previous recorded one to determine rotational and translational movement done. These values are then randomly adjusted based upon their magnitude to find a new pose: the pose of the particle. This results in the particles varying further with each movement.

| Parameter | Value |
|---|---|
| (Rotation constant) $k_1$ | 0.01 |
| (Translation constant)$k_2$ | 0.005 |
| MIN_DISTANCE | 0.0025 |
| MIN_THETA | 0.02 |

TABLE III: Uncertainty parameters

*b) Sensor Model:* The sensor model is used to update the weight of each particle based on the probability of their pose to be the true pose of MBot. We calculated this probability by mapping all laser scans to start at the pose of the particle, and increase the probability by the cell's log-odds if any laser scan that ended at an occupied cell on the map.

*c) Particle Filter:* The particle filter is the final part of the localization algorithm and outputs the estimated pose of MBot. It first initializes all particles at a single given pose, with the default $(0, 0, 0)$ for ($X_{curr}$,$Y_{curr}$,$\theta_{curr}$), with equal weights. After the action and sensor models update the particles we normalize their weight to sum to 1. Then, we take a weighted average of all particles' pose to estimate the robot position. We then re-sample the particles based on their current weight using low-variance re-sampling technique and feed them back to the action model for future loops when MBot moves.

*3) Combined Implementation:* The complete SLAM system is illustrated using in a block diagram Fig. 4.

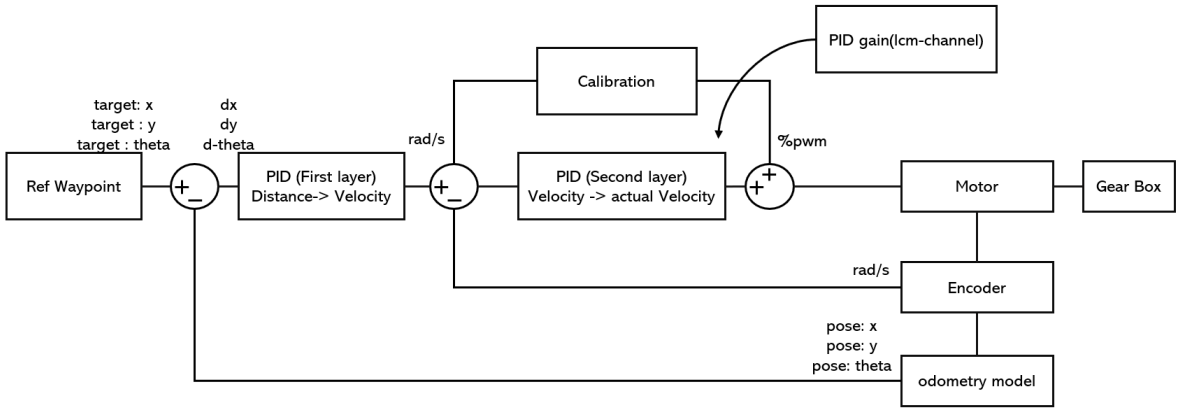


Fig. 2: The block diagram of velocity controller

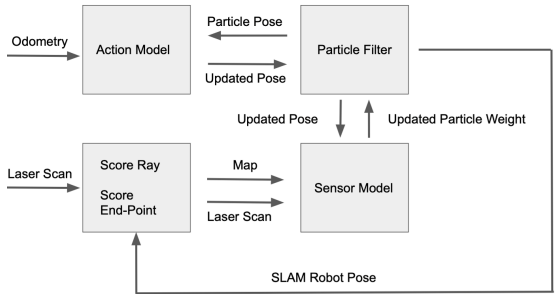


Fig. 4: Block diagram that illustrates how different components of the system interacts with each other

### C. Planning and Exploration

Planning and exploration is the process of the robot creating a map.

*1) Path Planning:* We use the A* graph search algorithm for producing an efficient goal-oriented path from a given start position Each cell in the map's occupancy grid is scored on how difficult it is to reach and its distance from the goal. We also apply an additional penalty based on the proximity of a cell to an obstacle in order to avoid walls as much as possible. The robot then generates a path by following the lowest scored cells until the goal is reached. Once found, we trim the path by removing redundant waypoints. A redundant waypoint would be one in which the heading of the path is the same as the next; an example would be moving 2m north instead of 1m north twice.

*2) Exploration:* Our exploration is driven by computing frontiers in the occupancy grid. A frontier can be defined as the collection of a contiguous cells in the occupancy grid that reside at the border of known free space and unknown space. We follow a greedy approach here trying to find the nearest frontier group. We then find out if that frontier is reachable. This is achieved by carrying out a Breadth-First Search around the cells from the center of the frontiers. If there are no obstacles near the frontier, it is set up as our goal position, otherwise is disregarded. The A* then calculates an optimal path to this new goal. The frontiers are updated in real-time until the MBot has explored the entire area.

## III. Experiments and Results

We carry out our experiments on the MBot, a two-wheeled differential - drive non-holonomic robot with a single castor wheel in the rear.

### A. Motion Controller

*a) Wheel and Motor Responses:* Figure 5 shows the left and the right wheel responses converging to the value of 0.5 m/s which is the reference signal. The signals seem to have a acceptable percent overshoot, settling time and steady state error showing the efficacy of the PID gains in controlling the MBot.

*b) Speed Limitations:* Any mobile bot can have only limited maximum permissible speed ($V_{max}$) because of the wheel encoders, motor ticks and feedback frequency. In Figure 6, we can see that beyond $0.65\ m/s$, the MBot is unable to speed up and thus, $V_{max} = 0.65\ m/s$. The minimum non-zero movement speed ($V_{min}$) of the MBot is 0.05 m/s.

*c) Test Drive:* Post tuning our gains, we run a test drive to verify the performance of our wheel encoders and motion controllers. Figure 7 shows the translation velocities and angular velocities. The test run was a straight 1m drive, followed by a 180 ° turn and then returning to the start position. It is noteworthy that angular velocity stays zero during the linear motion and then increases as the MBot proceeds to take a turn and then falls back when returning towards the start position in a linear motion. The MBot was able to successfully travel 1m, so the odometry was not adjusted.

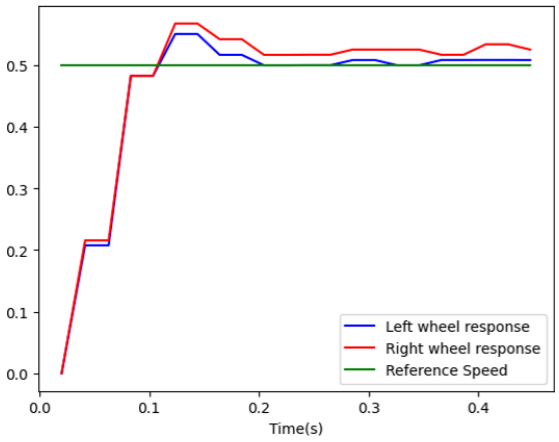


Fig. 5: Left and right wheel responses with respect to the reference speed (m/s vs s)

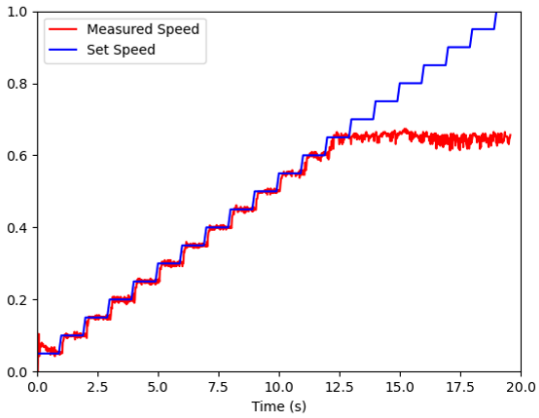


Fig. 6: The maximum and minimum possible velocities for the MBot (m/s vs s)

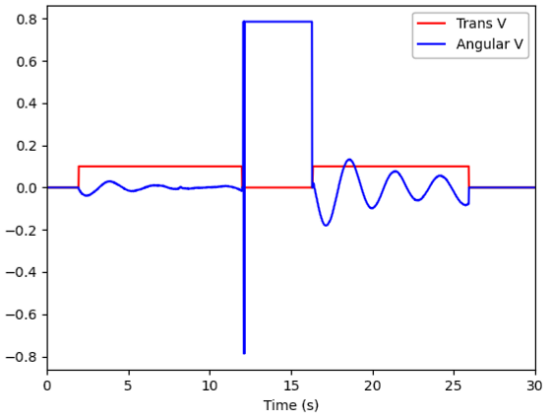


Fig. 7: The angular and translation velocities during a test run (m/s vs s)

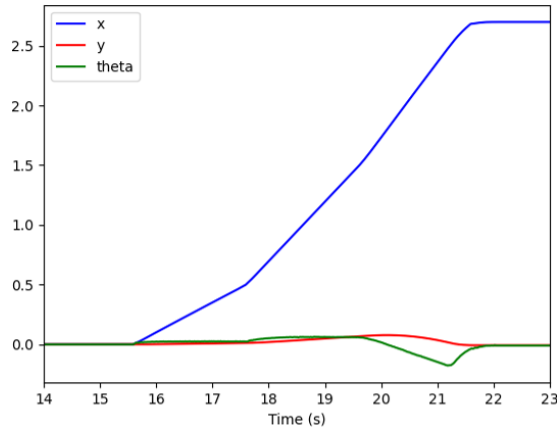


Fig. 8: MBot position and heading (m or radians vs s)

*d) Pose Estimation:* Recording robot position and heading (vs time) for the following step inputs: 0.25 m/s for 2s, 0.5 m/s for 2s, 1 m/s for 1s Figure 8, and the tableIV provides the specific values.

*e) A Simple Robotic Maneuver:* We run a simple robotic maneuver involving driving along a square four consecutive times. Figure 9 shows the pose estimate of the robot while using dead reckoning. We see that the robot drives adequately straight, but undershoots on turns. This could be the result of error accumulation of the $\Delta\theta$ values.

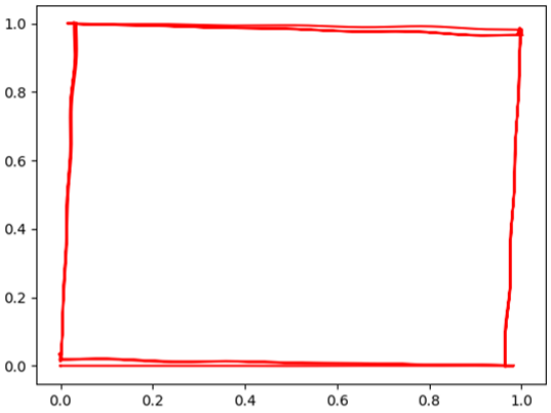

Fig. 9: Plot showing robot pose using dead reckoning while driving in a 1m square four times (m vs m)

| Time | x | y | theta | x-vel | y-vel | angular-vel |
|---|---|---|---|---|---|---|
| 2s | 0.486 | 0.024 | 0.024 | 0.243 | 0.012 | 0.012 |
| 4s | 1.504 | 0.058 | 0.049 | 0.509 | 0.017 | 0.0125 |
| 5s | 2.576 | 0.002 | -0.012 | 1.072 | -0.059 | -0.061 |

TABLE IV: Mbot position, heading and velocity value in 2t,4t and 5t (m/s, rad/s)

### *B. Simultaneous Localization and Mapping (SLAM)*

*1) Mapping:* Figure 10 shows how the MBot translates its surroundings onto an occupancy grid. The pink

colored cells denote frontiers, while the green rays are symbolic of the Lidar data. The cells in white and black denote the likelihood of them being empty or not respectively.

*2) Monte Carlo Localization:* We calculated the time taken for the particle filter to run with a varying cardinality of particle sets. Quite intuitively, Table V shows that the time increases with the number of particles. V It is noteworthy that our particle filter can update more than 10Hz even when we use 1000 particles. Based on a linear extrapolation, we believe the maximum number of particles we can run in 10Hz is 2200. In practice, we used 500 particles when finishing checkpoints and the competition because more number of particles does not improve our performance significantly and is computationally intensive. This is because of our optimal sensor model where we use end-points of each laser ray to update particle weight tested using simulations. It though does seem that it impacted our performance in the competition as the map wasn't as fine as was expected. For proof of concept, we took images at the midpoint of each 1m translation and at the corners after having turned 90° when using drive_square_10mx10m_5cm.log with 300 particles. We showed one image when MBot is at the first turning corner in Fig. 11, while the others have been attached in the appendix.

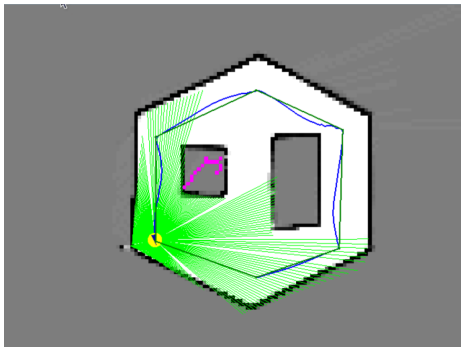

Fig. 10: MBot do mapping-only in $obstacle - slam - 10mx10m - 5cm.log$

| Num Particles | 100 | 300 | 500 | 1000 |
|---|---|---|---|---|
| Avg Time | 0.0051 | 0.0156 | 0.0245 | 0.0447 |

TABLE V: Time it takes for particle filter to update once with different number of particles

With the exact same setup, the trajectory of SLAM pose and odometry pose is compared in Fig. 12.

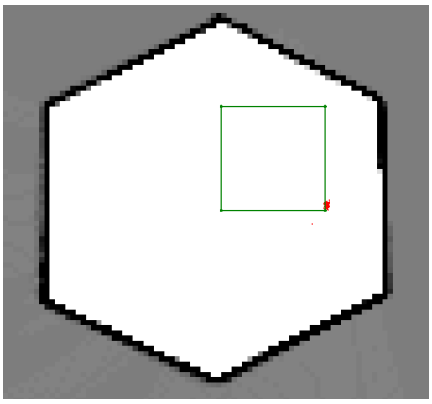

Fig. 11: Particles when MBot has finished turning 90° when using drive_square_10mx10m_5cm.log with 300 particles

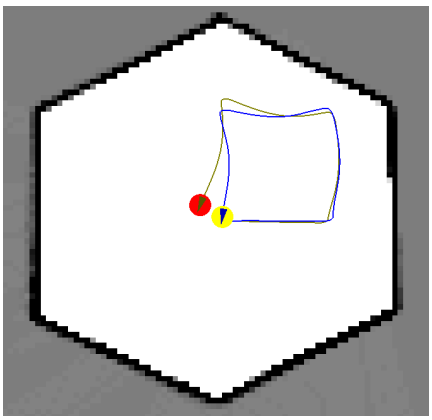

Fig. 12: The comparison between SLAM and odometry trajectory when using drive_square_10mx10m_5cm.log with 300 particles.

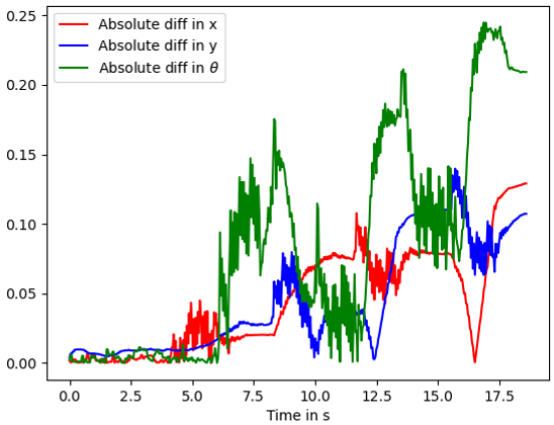


Fig. 13: Absolute error between SLAM and odometry pose when using drive_square_10mx10m_5cm.log with 300 particles.

The blue trajectory comes from SLAM while the green trajectory comes from odometry. As we can see from Figure 12, the pose from the odometry has not returned to its starting position while the SLAM pose has returned after driving around a square. This demonstrates that the slam is able to correct inaccuracies in the odometry. We also calculated the absolute difference between SLAM and odometry pose, which has been summarized in Figure 13. It is evident that with increasing time, the accumulated error and the corresponding difference between between SLAM and odometry pose increases.

Further, upon comparing the trajectories with the true pose (see Figure 14, we note our slam pose follows the true pose more accurately. The orange curve shows the ground-truth trajectory and the blue curve shows the SLAM trajectory. Owing to this, we used SLAM pose as our robot pose in motion planning. The absolute error between the SLAM pose and true pose is shown in Figure 15, and the RMS error is summarized in Table VI.

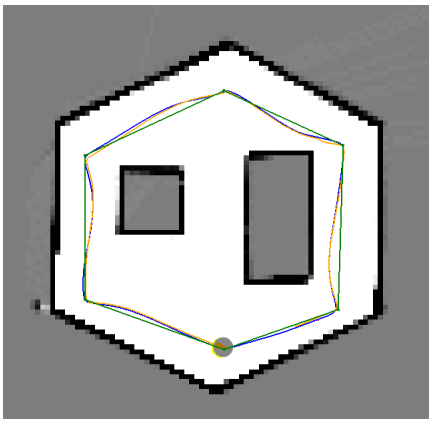

Fig. 14: The comparison between SLAM and ground-truth trajectory when using obstacle_slam_10mx10m_5cm.log with 300 particles. (m or radians vs s)

| Dimension | x | y | $\theta$ |
|---|---|---|---|
| RMS error | 0.055 | 0.049 | 0.677 |

TABLE VI: RMS error between SLAM pose and ground truth pose when using obstacle_slam_10mx10m_5cm.log with 300 particles. The unit for x any y is m, and for $\theta$ is rad

As we can observe from Fig. 15, the SLAM trajectory agrees well with the ground truth trajectory since the absolute error in dimension x and y are well bounded in 0.2m.

This can also be seen from Table VI as our RMS error in x and y are very small. Meanwhile, it seems that we have a very large error in $\theta$. We observed that the odometry and the SLAM pose lag behind the true pose noticeably when turning around corners. The SLAM pose does catch up with the true pose eventually.

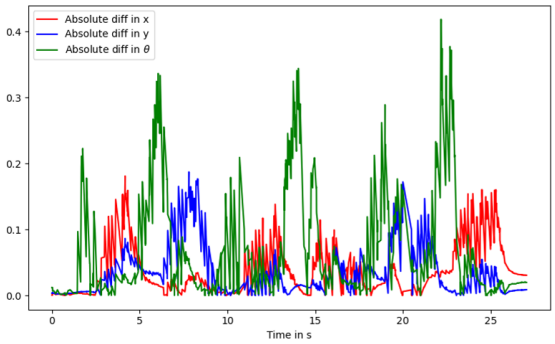


Fig. 15: Absolute error between SLAM and ground truth pose when using obstacle_slam_10mx10m_5cm.log with 300 particles. (m or radians vs s)

| Test | Min | Mean | Max | Median | Std Dev |
|---|---|---|---|---|---|
| Convex | 151 | 151 | 151 | 0 | 0 |
| Empty | 1122 | 3483 | 6055 | 6055 | 1985 |
| Maze | 4600 | 16939 | 39640 | 8396 | 13637 |
| Narrow | 1403 | 2318 | 3232 | 0 | 914.5 |
| Wide | 1117 | 3219 | 4569 | 4569 | 1506 |

TABLE VII: Successful planning statistics

| Test | Min | Mean | Max | Median | Std Dev |
|---|---|---|---|---|---|
| Convex | 69 | 100 | 117 | 69 | 22 |
| Empty | 42 | 73 | 104 | 0 | 31 |
| Filled | 36 | 97 | 137 | 103 | 33 |
| Narrow | 100 | 1.7E8 | 5.0E8 | 136 | 2.35E8 |
| Wide | 92 | 92 | 92 | 0 | 0 |

TABLE VIII: Failed planning statistics

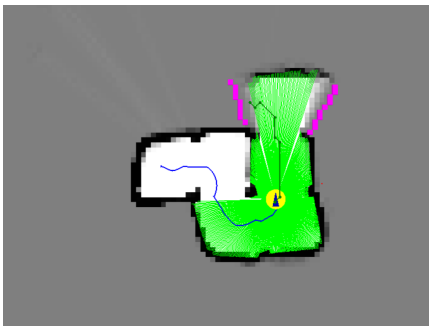

Fig. 16: Exploration in checkpoint2 maze

Initially, we thought the time mismatch to be a potential reason for this. However, the lag still exists after time alignment. This could particularly be the cause of a significant error in $\theta$. The error in $\theta$ arises during turning and would be corrected once the SLAM pose catches up with the true pose.

### C. Planning and Exploration

For planning and exploration, we observe that the MBot is able to find reachable frontiers and performs A* algorithm to find a valid path.

*1) Path Planning:* The A* is capable of passing all tests and is operable in botgui. Table VII and Table VIII presents the metrics for the time taken for the test cases. The algorithm performs well with the clear exception of one test: Narrow Constriction with valid goal but no valid path. This is the only relevant failed test. All of the others fail due to invalid goals, which does not result in A* being called. This test's time represents the period it takes for the algorithm to search every node in the lower half of the map, a process that would take prohibitively long if this situation were to occur physically. Despite this non-optimal performance, A* proves to be efficient when a valid path does exist. All of the passed tests take a short amount of time proportional to the complexity of the maze.

*2) Exploration:* In the exploration experiment, we make MBot perform exploration in the maze provided for Checkpoint 2. Post exploring, it returns to its starting position.

## IV. Discussion

### A. Performance in Competition

The day before the competition, our Mbot hit some problems with the motor and some hardware components due to an unprecedented fall. After, the competition, we re-calibrated our Mbot figuring out potential solutions.

*1) SLAM:* The task 1 for the competition aimed at testing the accuracy of our SLAM system explicitly by driving the Mbot around a 1 meter square 4 times while building the map. The map built in the competition is shown in Figure 17. From the figure, the map is consistently drifting when MBot is driving around the square. The map's quality worsens with every circle. As a result of this, MBot is unable to return to its starting position. $Solution$ : One quick solution we found to decrease the difference between robot's initial and final position is to turn off the map updating algorithm after MBot drives around the square by one circle. However, a more complete solution should be improving our mapping and localization algorithm to improve its accuracy. Based on our discussion with other teams, they have a more complex sensor model to evaluate the weight of their particles. We would love to give it a try if given more time.

*2) Exploration:* We encountered three main problems in exploration.

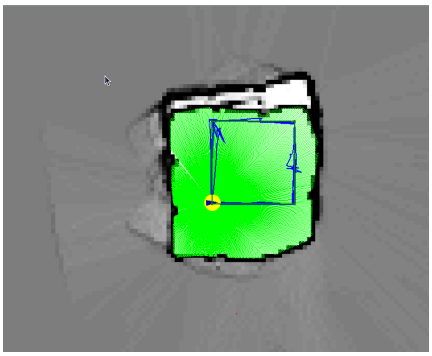

Fig. 17: The issue for task1, the mapping is shifting

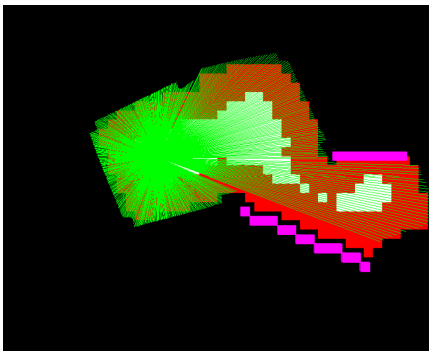

Fig. 18: No valid path to frontiers

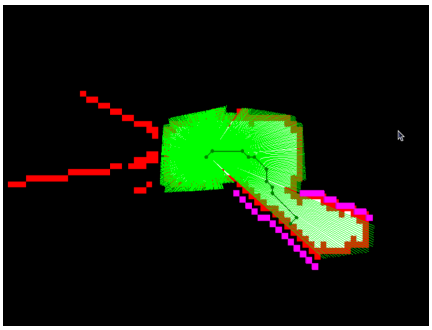

Fig. 19: A* finding valid path

- Our point will be in an unreachable place, because our obstacle distance is too large, and it will be blocked on the narrow road (Figure 18)
- Our BFS computation is quite slow resulting in a slower path update.
- The path generated by the A* algorithm is too close to the wall, and it is easy to bump into the wall or fail to generate the new target path.

$Solution$ :We may reduce the obstacle distance and increase $g$ value for a cell close to wall so A* algorithms would choose the path avoiding nodes near the wall (Figure 19)

## APPENDIX A
## REQUIRED IMAGES

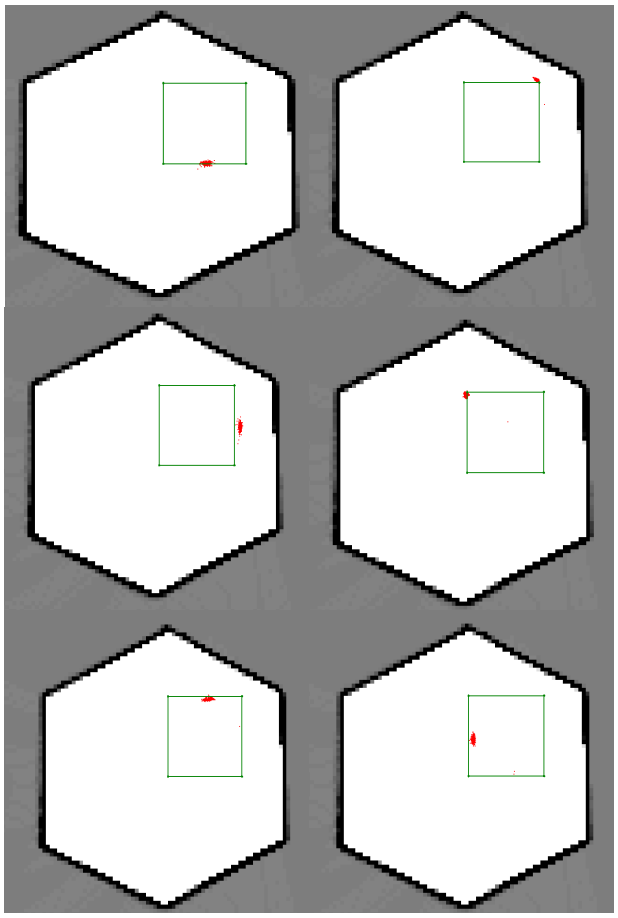

Fig. 20: Particles when MBot when using drive_square_10mx10m_5cm.log with 300 particles.

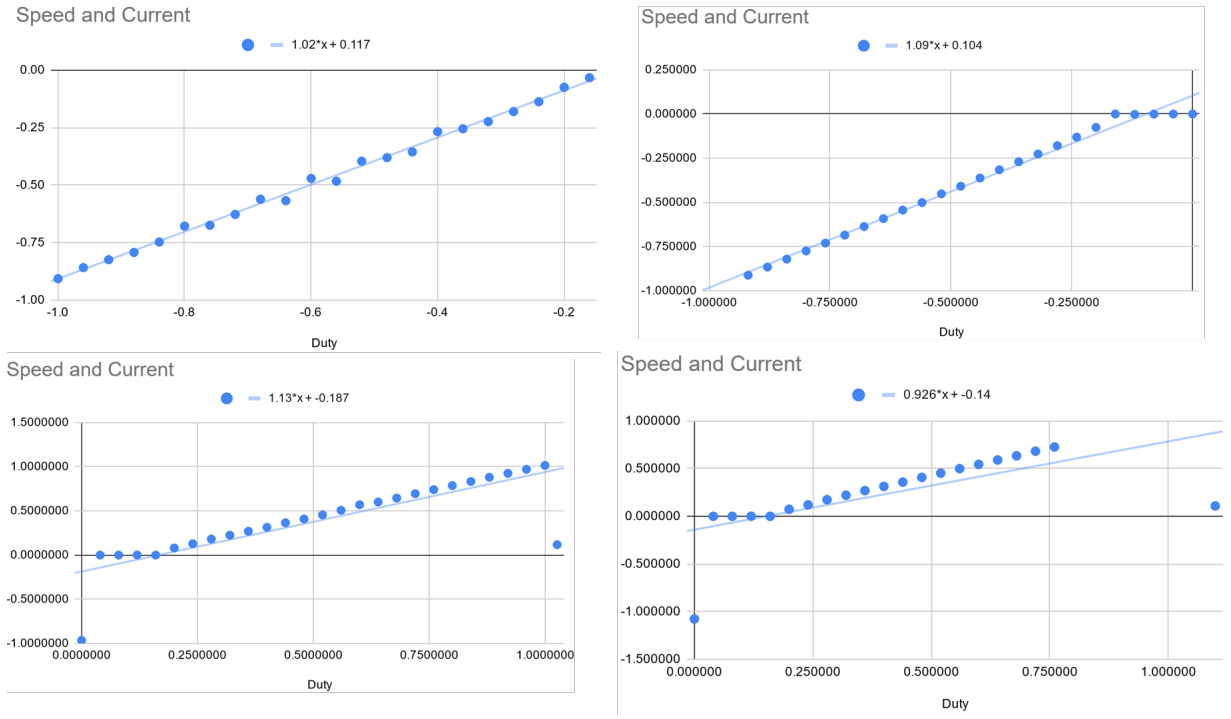


Fig. 21: Calibration values computed experimentally. Clockwise from top right forward-right, backward right, backward-left, forward-left