Team 10 ROB 550 BotLab Report

Dhiraj Maji, Sarvesh Mayilvahanan, Cesar Ramos

***Abstract*—In this project, we designed a pipeline for the Mbot that enabled it to autonomously navigate and map an unknown environment. The motion, body frame, and wheel speed controllers generate commands for the Mbot to move from one point to another in a smooth fashion. The Simultaneous Localization and Mapping (SLAM) system helps the robot explore its surroundings and build a map of the environment as well as localizing the Mbot within that map. The path planning and exploration systems help in generating waypoints between a start and a goal position and in exploring the environment.**

I. Introduction

THE Mbot is a mobile robot that moves with differential drive, 2 parallel wheels and a rear caster; thus, it has two degrees of freedom. With respect to its sensing capabilities, it has a scanning 2D Lidar and a MEMS 3-axis IMU which allows us to estimate the pose of the robot and make a map of the environment. Finally, for compute we have two boards: the Raspberry Pi 4B and a Raspberry Pi Pico W. They serve different purposes: the Pico handles all the low level motor control and the Pi4B handles the high level mapping, localization and filters. A robot with these capabilities is important because it replicates the real problem of autonomous navigation in a pedagogical where students can explore the boundary between theory and practice, and based on that, make real life decisions to handle uncertainty.

Moreover, we can relate the functionality of the Mbot into acting, perception and reasoning. The Controller and Odometry block of our Mbot handles the acting part. The odometry is important to determine our current position. Given a goal point and the current position, the controller is necessary to generate velocity commands for our motors to move the Mbot to the goal position. The perception functionality of our Mbot is handled by the Simultaneous Localization and Mapping (SLAM) system. It plays the role of building a map of the environment around the Mbot and localizing the Mbot within that map. It helps in determining the position of the obstacles and the regions in the map where the robot can go. The reasoning of our Mbot is handled by the Path Planning and Exploration system. To autonomously navigate an environment, we need to autonomously generate waypoints from one point in the map to another. This is handled by the path planning system. Moreover, we usually don't have a map of the environment all the time, hence the robot will have to explore its environment to build the map. The exploration system handles the reasoning of how to explore an unknown environment.

II. Methods

In this section, we discuss the main parts of our Mbot which can be divided into the Control system, the Simultaneous Localization and Mapping (SLAM), and the Path Planning and Exploration. The Control system decides the setpoints for our Mbot such that it can navigate from one point to another smoothly. The SLAM system handles the perception of our environment and creates a map and localizes the Mbot in that map. The Path planning and Exploration part helps our Mbot to move autonomously through the environment and create a map of it, and decide the waypoints to reach from one point in the map to another efficiently.

A. Controller and Odometry

1) Controller: At the highest level, our controller can be divided into 3 main parts - a) the motion controller, b) body frame controller and c) wheel speed controller.

The motion controller can be again divided into two parts - a PID controller for point to point movement and another PID controller for in place rotation. The PID controllers take in waypoints and give velocity control setpoints for the body frame controller. In our case, the motion from one point to another is carried out as Rotation, Translation and Rotation (RTR). The Mbot initially rotates to orient itself along the path in which it is supposed to move. Then the robot moves from one waypoint to the next. Then finally the robot orients itself again to the desired orientation at that waypoint.

The body frame controller is used to control the forward speed and the turning speed of the bot. We use a PID controller for our body frame controller with PID values for forward velocity as 0.4, 0.001, 0.0 and the PID values for turn velocity as 0.4, 0.01 and 0.0001, which are the K_p, K_i, K_d values respectively. The body frame controller computes the velocity errors using the encoder data and generates the left and right wheel speed controls. For each wheel, we compute the setpoints for forward motion and turn motion separately and then add them to get the final setpoint for that wheel.

The wheel speed controllers are used to control the individual speeds of the left and right wheel. We use a PID controller with values of 0.4, 0.01 and 0.0001 for our left wheel speed controller. we again use a PID controller with values of 0.4, 0.01 and 0.0001 for our right wheel speed controller (K_p, K_i, K_d respectively). Our controller takes in the encoder output and computes the error between individual wheel speeds and the current setpoints. Then it computes the next setpoint for each wheel, which is converted to a duty value for each motor based on the calibration parameters. The calibration process completes a linear regression of the duty and wheel speeds, which allows us to easily convert between the two domains.

TABLE I: PID values for the controllers

Controller	K_p	K_i	K_d
Left wheel control	0.4	0.01	0.0001
Right wheel control	0.4	0.01	0.0001
Forward vel. control	0.4	0.001	0.0
Turn vel. control	0.4	0.01	0.0001

2) Odometry: The odometry system in the robot uses the encoder measurements from the motors to update the robot's position estimate. This estimate is often incorrect as it is purely based on internal sensors on the robot, which causes errors to accumulate over time. This is accounted for by utilizing another sensor with external information to refine the odometry position. This correction method is Simultaneous Localization and Mapping (SLAM) and is discussed in the following subsection.

The odometry for the MBot was calculated using the motion model. Initially, the distance travelled by each wheel Δs_w is calculated using the encoder measurements as shown in Equation (2) below.

$$\Delta\phi_w = 2\pi \cdot \frac{\Delta e_w}{\text{ER} \cdot \text{GR}} \tag{1}$$

$$\Delta s_w = r_w \Delta\phi_w \tag{2}$$

Where Δe_w is the change in encoder value within odometry updates, and ER and GR are the encoder resolution and gear ratio respectively. r_w is the wheel radius. The encoder resolution was found to be 20.0 and the gear ratio was found to be 78.0, which were determined based on provided specifications for the motor and encoder. The wheel radius was measured as 41.85 mm, which was an average of multiple measurements.

Using the motion model, the change in heading $\Delta\theta$ and change in position $(\Delta x, \Delta y)$ are calculated.

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{b} \tag{3}$$

$$\Delta d = \frac{\Delta s_R + \Delta s_L}{2} \tag{4}$$

$$\Delta x = \Delta d \cos(\theta + \Delta\theta/2) \tag{5}$$

$$\Delta y = \Delta d \sin(\theta + \Delta\theta/2) \tag{6}$$

Where $\Delta s_R, \Delta s_L$ are the distance travelled by the right and left wheels and b is the wheelbase (distance between wheels). The wheelbase was measured to be 0.163 meters. Δd in Equation (4) represents the linear distance travelled between odometry measurements and is an approximation of the true arc length travelled by the robot. However, at small distances, they are approximately the same. The change in x, y position is simply an application of trigonometry given the change in orientation and distance.

The new odometry position estimate $p_{t+\Delta t}$ can then be calculated as a sum of the previous odometry position $p_t = [x, y, \theta]_t$ and the change in orientation as follows:

$$p_{t+\Delta t} = p_t + [\Delta x, \Delta y, \Delta\theta] \tag{7}$$

3) Gyro-odometry fusion: Gyrodometry is a method to fuse the odometry estimate with gyro data to gain a more accurate heading estimate. The basic premise is to trust the motor encoders to estimate the robot's heading in the majority of situations, but in the event that the odometry and gyro estimate very different headings, it is likely that the robot experienced some physical misalignment and the gyro is more accurate.

Using the change in heading from the odometry $\Delta\theta_{\text{odom}}$ and change in heading from the gyro $\Delta\theta_{\text{gyro}}$, Gyrodometry works as follows

Algorithm 1 Gyrodometry

```
Data: $\Delta\theta_{\text{odom}}$, $\Delta\theta_{\text{gyro}}$, $\theta_t$
Result: $\theta_{t+\Delta t}$
$\Delta\theta_{G-O} = \Delta\theta_{\text{gyro}} - \Delta\theta_{\text{odom}}$
if $|\Delta\theta_{G-O}| > \Delta\theta_{thresh}$ then
    $\theta_{t+\Delta t} = \theta_t + \Delta\theta_{\text{gyro}}$
else
    $\theta_{t+\Delta t} = \theta_t + \Delta\theta_{\text{odom}}$
end
```

The threshold $\Delta\theta_{\text{thresh}}$ was tuned to be 0.001 radians, but we found that using just the odometry heading was very accurate for our purposes and chose to not use gyrodometry when estimating the pose.

B. Simultaneous Localization and Mapping (SLAM)

The SLAM systems consists of two parts - mapping and localization. The Mapping algorithm builds the map of the Mbot environment and continuously updates the map as the Mbot moves in its environment. The Monte Carlo Localization is used to localize the Mbot in the map. The Monte Carlo localization uses a Sensor model, Action model and the Particle filter to find its position and orientation with respect to the map.

1) Mapping: The map used for SLAM is an occupancy grid with the log of object probabilities. This map is updated using the lidar data by first updating the cells containing the ray endpoints. If the ray had the maximum range, this means no object was hit and we decrement the log odds in the endpoint cell. If the ray range was less than the maximum, we increment the log odds because an object was hit. Then for all the intermediate cells, we decrement the log odds as there was no object in those locations. The intermediate cells are found using Breshenham's algorithm, which finds all cells that are touched by a line in the map.

2) Monte Carlo Localization: The Monte Carlo Localization has 3 major parts:

- Action Model which applies an action to a particle to propagate it forward in time to get the new pose for the particle.
- Sensor Model which gives the probability that a lidar scan matches the pose of the particle given the map and generates a likelihood for each particle.
- Particle filter which resamples the particles using low-variance sampling and computes an estimated SLAM pose using the posterior distribution of particles.

a) Action Model: The action model propagates each particle using the odometry model by decomposing the movement between the current and previous odometry poses into three actions: rotation α, translation Δs, and rotation $\Delta\theta - \alpha$ as seen in Figure 1. We use these actions to update our distribution of actions (only if our robot moved), which we randomly sample from and use to propagate each particle. Each of the actions are calculated as follows using the current and previous odometry poses:

$$\alpha = \tan^{-1}(\Delta y/\Delta x) - \theta_{t-1} \tag{8}$$

$$\Delta s = \sqrt{\Delta x^2 + \Delta y^2} \tag{9}$$

$$\Delta\theta - \alpha = \theta_t - \theta_{t-1} - \alpha \tag{10}$$

The action model serves to predict the next state solely based on a control input (without any measurement or data), and as such it can be expressed as the problem of determining $P(X_t|u_t, X_{t-1})$, where X_t is the state at time t and u_t is the control input (action taken by the robot). In Equation 11 we give an mathematical model of the action model where x_t, y_t, θ_t are random variables representing the state. The variance of the distribution of these random variables is dependent on $\varepsilon_1, \varepsilon_2, \varepsilon_3$, which are random variables representing the three actions, whose distributions are updated based on the change in odometry pose using Equations (8) - (10).

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} (\Delta s + \epsilon_2)\cos(\theta_{t-1} + \alpha + \epsilon_1) \\ (\Delta s + \epsilon_2)\sin(\theta_{t-1} + \alpha + \epsilon_1) \\ \Delta\theta + \epsilon_1 + \epsilon_3 \end{bmatrix} \tag{11}$$

where

$$\epsilon_1 \sim \mathcal{N}(0, k_1|\alpha|)$$
$$\epsilon_2 \sim \mathcal{N}(0, k_2|\Delta s|)$$
$$\epsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|)$$

In table II, the scaling constants affecting the variance of the action distributions are listed. Since the parameters k_1, k_2 control the variance of the distribution of our particles we determined them by hand tuning. Our objective was to obtain a dispersion of particles that was diverse enough so as to give a good estimated pose. However, we also did not want too much dispersion such that there weren't enough particles with good weight (high likelihood from the sensor model).

TABLE II: Action model uncertainty parameters.

Parameter	
k_1	0.1
k_2	0.1

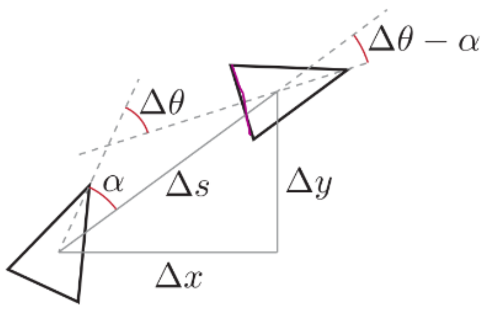


Fig. 1: Geometrical setup for the action model, with three-step action.

b) Sensor Model: The sensor model provides a likelihood for a particle given the lidar and map data, which can be expressed as the problem of determining $P(z_t|X_t, m_t)$, where z_t is the lidar scan and m_t is the map at time t. This is accomplished by assigning a score to each ray in the lidar scan, the product of which represents the likelihood of the particle as shown in Equation

$$P(z_t|X_t, m_t) = \prod_{i=0}^{N} P(\text{ray}_i = d_i|X_t) \tag{12}$$

In order to take advantage of the current occupancy grid configuration, which stores log probabilities, we can reformulate the likelihood of the particle to be

$$\log P(z_t|X_t, m_t) = \sum_{i=0}^{N} \log P(\text{ray}_i = d_i|X_t) \tag{13}$$

where $\log P(\text{ray}_i = d_i|X_t)$ is defined in our likelihood model as the log odds of the ray endpoint raised to a power of 1.5 if the probability of the endpoint being an object is greater than 0.5 (log odd > 0 for this particular implementation). Otherwise, the score for the ray is an arbitrarily small value 0.001. This is reflected in Equation (14), where $m_t(\text{ep}_i)$ is the occupancy grid log odd of the endpoint of ray i.

$$\log P(\text{ray}_i = d_i|X_t) = \begin{cases} m_t(\text{ep}_i)^{1.5} & m_t(\text{ep}_i) > 0 \\ 0.001 & \text{otherwise} \end{cases} \tag{14}$$

c) Particle Filter: The particle filter's main function `updateFilter` is a combination of the mapping, action model, and sensor model described above. It first updates the action model using the odometry data. It then resamples the particles using a low variance resampler, which is the new prior distribution. The low variance resampler works as follows:

Algorithm 2 Low Variance Resampler

```
Data: Particle distribution $Z_t$, Particle weights $W_t$, Number of particles $M$
Result: $\overline{Z}_t$
$\overline{Z}_t = \emptyset$
  $r = \text{rand}(0, M^{-1})$
  $c = W_t^{(1)}$
  $i = 1$
for $m = 1 \ldots M$ do
    $U = r + (m-1) \cdot M^{-1}$ while $U > c$ do
        $i = i + 1$ $c = c + W_t^{(i)}$
    end
    $\overline{Z}_t \leftarrow Z_t^{(i)}$
end
return $\overline{Z}_t$
```

After resampling the particles to obtain a prior distribution, the proposal distribution is generated by propagating the prior particles using the action model. Then the posterior distribution is found by finding the likelihood of each particle using the lidar and map data through the sensor model, which becomes the new weight for that particle (after normalizing so that all weights sum to 1).

Once the posterior distribution of particles is found, the SLAM pose can be estimated through a weighted average of the particles. For our implementation, we chose to only consider the top 10% of particles sorted by weight.

$$x_{SLAM} = \frac{1}{\sum_{i=1}^{N} W_t^{(i)}} \sum_{i=1}^{N} W_t^{(i)} x_t^{(i)} \tag{15}$$

$$y_{SLAM} = \frac{1}{\sum_{i=1}^{N} W_t^{(i)}} \sum_{i=1}^{N} W_t^{(i)} y_t^{(i)} \tag{16}$$

$$\theta_{SLAM} = \tan^{-1}\left(\frac{\frac{1}{\sum_{i=1}^{N} W_t^{(i)}} \sum_{i=1}^{N} W_t^{(i)} \sin(\theta_t^{(i)})}{\frac{1}{\sum_{i=1}^{N} W_t^{(i)}} \sum_{i=1}^{N} W_t^{(i)} \cos(\theta_t^{(i)})} \right) \tag{17}$$

where $N = 0.1M$, where M is the current number of particles. The threshold of 10% was tuned based on observation of the distribution of weights during SLAM.

This process within the particle filter is shown in Figure 2. There are two main paths involving the mapping and particles. The mapping uses the SLAM pose to set the ray origins, which ultimately affects the sensor model's likelihood. The action model is used in conjuction with the resampler and previous particle posterior distribution to create a proposal distribution. This distribution is evaluated by the sensor model and used to update the weights and produce a posterior distribution, which is used to estimate the SLAM pose.

C. Planning and Exploration

This system has two main parts:

- Path planning, which helps in finding waypoints to move from one point in the map to another efficiently and avoiding obstacles.
- Exploration, which helps the robot to autonomously navigate the environment and create a map of it. These two sub-sections will be discussed in detail below.

1) Path Planning: We use the A* algorithm to plan our path from one point to another on the map. It computes two types of cost - the g-cost and the h-cost. The g-cost of a node is the distance from the start node to that node. The h-cost of a node is defined as the 4-way Manhattan distance from the node to the goal node. By summing these two costs, we can find the f-cost of a node. The A* algorithm expands in the direction of the node with the lowest f-cost until it finds the goal. Since the path usually has too many waypoints, we prune the path to reduce the number of waypoints in order to make the movement of the Mbot much smoother. This is accomplished by checking the difference in pose between waypoints. If the previous waypoint is within a distance Δs and within an orientation $\Delta theta$, then the waypoint is not added to the resultant path. These thresholds were manually tuned to be $\Delta s = 0.2$ m and $\Delta\theta = 0.5$ radians.

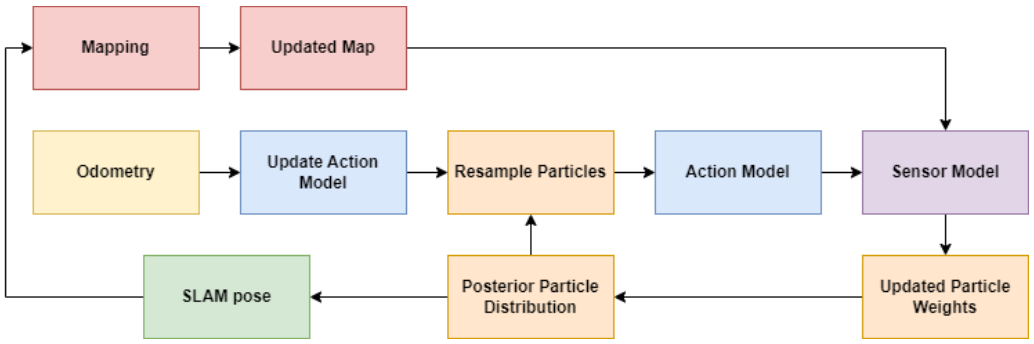


Fig. 2: Pipeline of SLAM system implemented for Mbot.

2) Exploration: The Exploration section helps in navigating an unknown environment and building a map of that environment. It resembles closely to that of a real world scenario where we do not have a map of our environment, and will need a exploration system to explore our environment and build that map. The Mbot initially starts at an unknown home location and builds the map around it. It looks for frontiers in the map which indicate that the region beyond it is not explored. Then it plans a path to the nearest frontier and moves a step towards it. This path is found by check a grid of points next to the frontier with a minimum distance of the robot's wheelbase b. If a valid point that the robot can plan a path to through A* is found, the robot will move towards the frontier. In the case that no path can be found, the frontier is considered unreachable. It keeps repeating the process again and again until all the reachable frontiers have vanished from the map. When all the reachable frontiers have been explored, exploration has finished and the Mbot returns to its home position where it started.

To find the frontiers, we use a connected components search in the map. Each connected component consists only of cells that are frontiers. Then, we scan the grid until an un-visited frontier cell is encountered, which we then grow until all connected cells are found. We then continue scanning through the grid.

III. Results

A. Controller and Odometry

1) Controller: As explained before the MBot uses a PID controller to regulate the speed of each wheel independently. As part of benchmarking these controllers we measured the response to a step input of 0.5 m/s as can be seen in Figure 3. In practice, the rise time and settling time were good enough for the MBot to follow a path. Moreover, the steady state error was not detrimental to later tasks and thus the gains were deemed correct for the application.

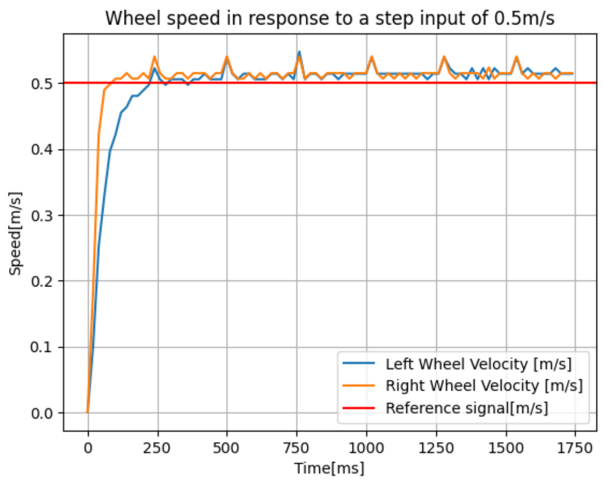


Fig. 3: Plot of the wheel speed response response to a step input of 0.5 m/s when running it in closed loop using a PID controller.

In our cascaded controller design the second stage is a linear velocity and rotational velocity controller, this quantities are with respect to the robot itself and not the wheels. In order to measure the performance of this stage we make the robot drive for 1m straight, turn 180 degrees and then return to the starting position. In Figure 4 we see this benchmark and see a good performance in both linear and angular velocity considering their scales. We see that as the robot is turning 180 degrees, the linear velocity stays at 0, indicating the Mbot is turning in place. As the robot is accelerating, decelerating, and moving linearly, there is slight angular velocity to correct the robot's heading, which is a product of the motion controller.

We make the robot drive straight under a sequence of different step inputs (0.25 m/s for 2 s, 0.5 m/s for 2 s, 1 m/s for 1 s) and plot the position and heading angle as can be seen in Figure 5. We see that the rotational velocity controller works really well for low

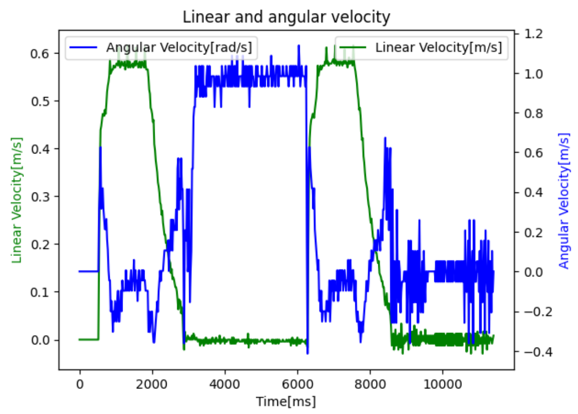


Fig. 4: Plot of linear and rotational speed as the robot drives 1m, turns 180 degrees, then returns to the start position.

speeds($\leq$ 0.25m/s) and has a larger settling time for larger speeds($>$ 0.5m/s).

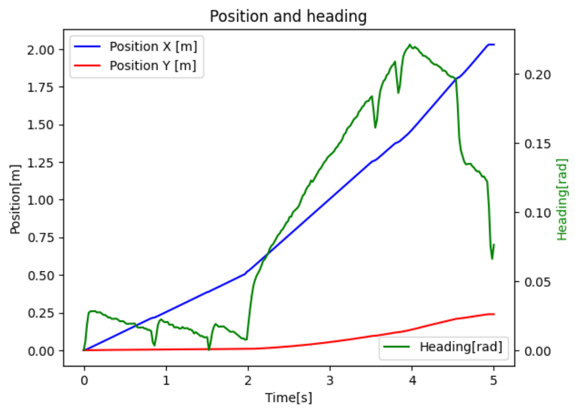


Fig. 5: Plot of robot position and heading (robot frame) vs time for the following step inputs: 0.25 m/s for 2 s, 0.5 m/s for 2 s, 1 m/s for 1 s.

Finally, we tested the wheel controller to find slowest and fastest speed the robot can travel and report the values in table III. These values were found by testing target velocities much higher and lower than typical linear velocities, and observing the steady state velocities using those targets.

Maximum Speed [m/s]	Minimum Speed [m/s]
0.662	0.065

TABLE III: Table showing the slowest and fastest speed the robot can travel using our wheel controller.

2) Odometry: To benchmark the position estimate based solely on encoders we make the robot traverse a 1 meter square 4 times and compare the pose estimate obtained using odometry (only encoders) to that obtained using SLAM (assuming this as the closest truth we have for this comparison). The comparison can be seen in Figure 6.

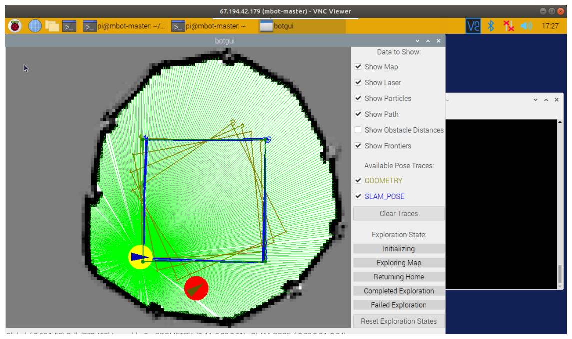


Fig. 6: Pose estimate using dead reckoning and moving through a 1 meter square 4 times. The green line represents the pose estimate produced using solely encoders and the blue line to the pose estimate using SLAM (our assumed truth).

To validate the odometry model we made the MBot traverse a square path and based on error with respect to the true square we made corrections. We detected errors in turning but not in straight line motion which indicated a problem in the baseline. Thus, we realized that the default value in the code base was wrong and updated with our measured one. Finally, we first tuned the PID controllers so as to minimize the error while during the square circuit. The updated wheelbase we used was 0.163m, the wheel radius for the left wheel was 0.04185m and the wheel radius for the right wheel was 0.041925m.

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: Figure 7 shows the plot of our map from the log file obstacle_slam_10mx10m_5cm.log. We can see that it clearly captures the objects in the middle of the arena and is able to capture the walls cleanly with little to no smearing.

2) Monte Carlo Localization:

a) Action Model: Testing on the provided log files in `--action-only --localization-only` mode (which allows for testing of only the action model) found that the action model was able to sufficiently propagate particles with a reasonable distribution.

b) Sensor Model: Testing on the provided log files in `--localization-only` mode (which allows for testing of only the sensor model) found that the likelihood function correctly assigned higher weights to ‘good’ particles and low weight to ‘bad’ particles, with enough distinction to cause the SLAM pose to follow the true pose.

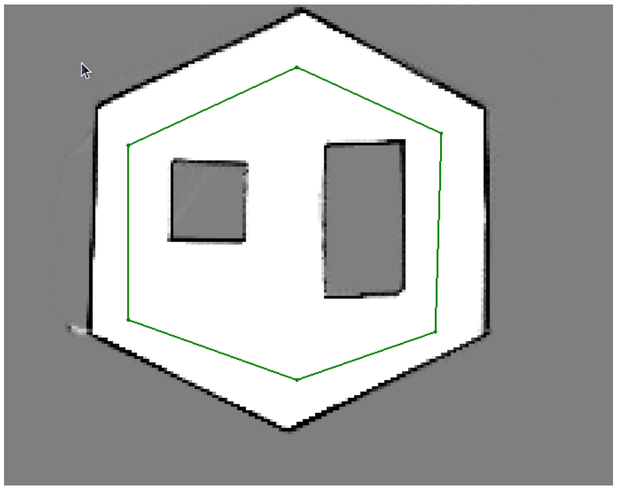


Fig. 7: Plot of our map from the log file obstacle_slam

c) Particle Filter: Putting all the components together, we tested the log files with full SLAM and found that we were able to get an accurate SLAM pose estimate, although the addition of small errors in each of the components (mapping, action model, sensor model) meant that it would sometimes cause smearing of the map.

Through manual testing, we found that our filter can support a maximum of 1300 particles running at 10 Hz on the RPi.

TABLE IV: Time to update the particle filter.

Number of Particles	Time(ms)
100	6.69
300	19.72
500	31.83
1000	59.21

3) Combined Implementation: We first investigate the performance of our pose estimate using SLAM versus the true pose. In table V we see the RMSE metric of our estimates. The values for states x, y look within bounds that express a good estimate, however, this is not the case for θ. As can be seen in figure 10 this is due to an misalignment between the data points. We presume this is caused by our data acquisition procedure. Thus, we manually align the time series and compute an ”Aligned RMSE” shown in table V where the errors for the three states are more accurate to what happened in reality.

TABLE V: Benchmark of our Slam estimates.

Error Metric	x	y	θ
RMSE	0.052	0.055	0.99
Aligned RMSE	0.032	0.004	0.023

We can additionally compare the pose estimates from SLAM and odometry by observing each of the poses as the robot explores a labyrinth. In Figure 9, we see that as the robot progresses through the labyrinth, the SLAM pose remains in a good state, accurately modeling the robot's pose while the odometry pose gradually gets worse as the error propagates.

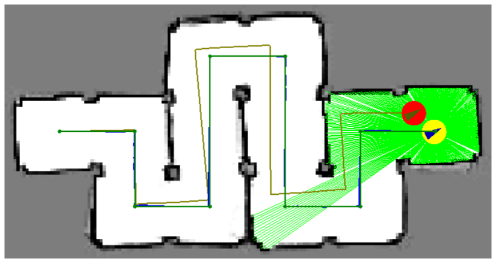


Fig. 9: Comparison of pose estimate for x, y, θ from SLAM (blue) and odometry (green).

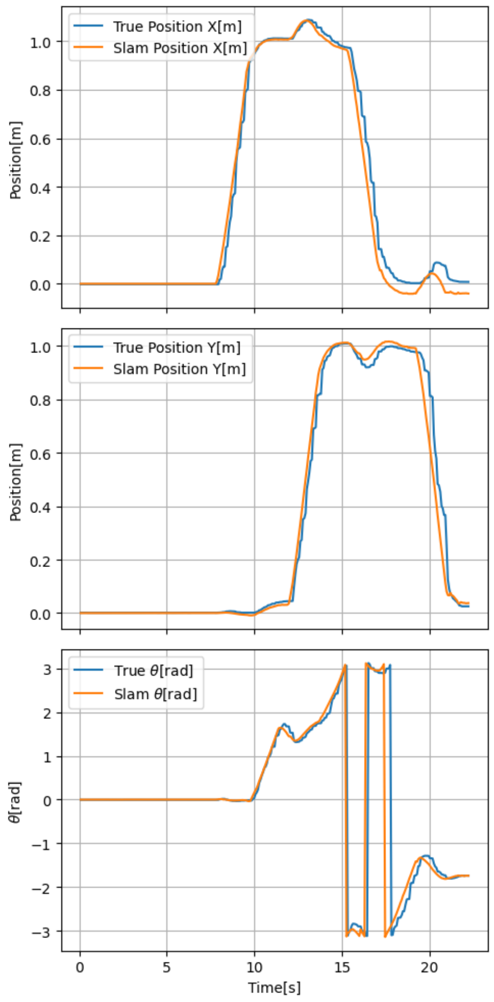
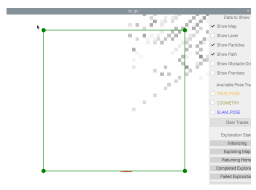
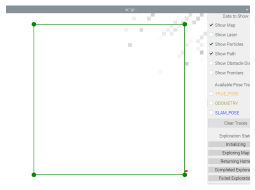
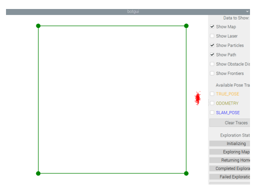
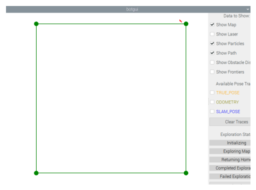


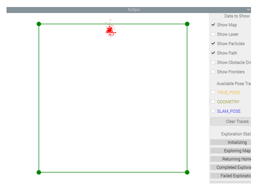
Fig. 10: Comparison of estimate for x, y, θ versus the true values.

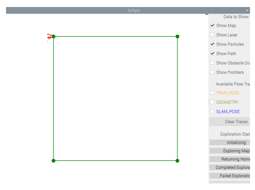

(a) Midpoint 1

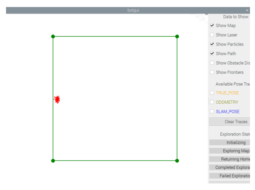

(b) Corner 1


(c) Midpoint 2


(d) Corner 2


(e) Midpoint 3


(f) Corner 3


(g) Midpoint 4

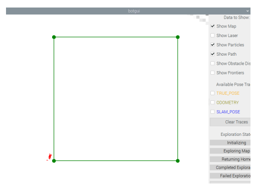

(h) Corner 4

Fig. 8: Plot of 300 particles at each mid-point and corner of drive square

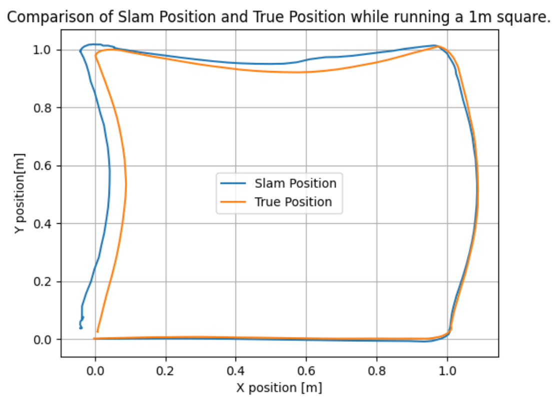


Fig. 11: Comparison of slam pose vs true pose when running a square of 1m.

C. Planning and Exploration

1) Path Planning: To evaluate the speed of our path planning algorithm, we perform test runs on various test cases. Table VI shows the statistical analysis of our algorithm's speed on test cases where our path planner was successful. Similarly, Table VII shows the statistical analysis of our algorithm's speed on test cases where the path planner failed.

TABLE VI: Timing information for successful planning attempts.

Grid	Min	Mean	Max	Median	Std.dev
convex grid	93	140	187	0	47
maze grid	394	4798.5	8952	3204	3263.4
narrow constriction	2592	3853.7	4667	4667	904.5
wide constriction	1638	3476.7	4576	4576	1308.4

TABLE VII: Timing information for failed planning attempts.

Grid	Min	Mean	Max	Median	Std.dev
convex grid	13	31	49	0	18
empty grid	33	46.6	51	50	6.83
filled grid	35	46.2	50	49	5.64
narrow constriction	12	31	50	0	19
wide constriction	47	47	47	0	0

IV. Discussion and Conclusion

The controllers implemented in the system performed very well in the competition. Our Mbot was able to perform the drive square task very accurately and finished the task with negligible error in orientation and around 3 cm error in position.

The mapping system of our Mbot performed well on the log files as seen in Figure 7 but experienced drift on the competition. The map was smeared on the competition tasks which indicated that the position of the robot was accumulating errors over time.

The path planning system was working well on the competition, though there is room for improvement. The Mbot was moving close to the walls as we didn't account for the obstacle distance in the g-cost. If given more time, we would account for the obstacle distance in our g-cost so that the Mbot can maintain safe distance from the obstacles.

The Exploration block was working well but had certain flaws. In certain cases, our exploration system wasn't able to find a valid waypoint near the frontiers, and hence it would skip that frontier. Additionally, in some instances, the map would slightly smear and cause the robot to think the frontier was unreachable when in actuality it was. As a result, even though we were able to explore the small dungeon on the competition day, we failed to explore the large dungeon. To remedy this, we could implement a method to wait until the bot has finished the path to the targeted frontier before planning to a new frontier. Our current method gets a new path at every iteration.

Overall, we were able to successfully implement a SLAM pipeline for the Mbot and had all of the base functionality working.

References

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

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