ROB 550 Botlab Report - Team 2 (PM)

Yung-Ching Sun, Tung Do, Liangkun Sun
{ycs, tungsdo, liangkun}@umich.edu

***Abstract*—This report presents our implementation and evaluation of controllers, a SLAM system, and planning algorithms on the MBot platform as part of the ROB 550 BotLab project. Acting, perception, and planning are three critical elements for mobile robots to operate accurately in the real world. We began by implementing and evaluating wheel-speed and motion controllers for MBot, enabling effective acting capabilities. Next, we developed an occupancy grid map implementation to build an environmental map based on laser scans and robot poses. We then implemented a particle filter with action and sensor models for robot localization, forming a complete SLAM system for MBot's perception. Finally, we implemented A* path planning and frontier-based exploration algorithms, providing reasoning capabilities for autonomous decision-making. Our controllers, SLAM, and planning system demonstrated excellent accuracy in comprehensive evaluations. Overall, this project provided hands-on experience with core mobile robotics capabilities, including speed and motion control, localization, mapping, planning, and autonomous exploration, all implemented into a real-world robotic system.**

I. Introduction

Mobile robots are autonomous systems designed to navigate and interact with their environment without human intervention. For effective operation in real-world applications, they require capabilities such as precise movement, mapping, localization, path planning, obstacle avoidance, and environmental exploration. This project focuses on implementing the core components of a mobile robot — acting (controllers), perception (SLAM), and planning — using the MBot platform.

MBot Classic platform integrates a differential drive mechanism powered by brushed DC motors with magnetic encoders. The robot's computation is handled by a dual-processor architecture: a Raspberry Pi 5 for remote development and high-level processing (SLAM and planning) and a Raspberry Pi Pico that manages low-level sensor data acquisition and motor control. A 2D LiDAR and a camera are connected with Raspberry Pi 5 for sensing, and the robot is powered by a 12V lithium-ion battery. The development was done in C and C++ with communication and message logging handled using Lightweight Communications and Marshalling (LCM).

This report details our implementation of controllers, SLAM, and planning system for MBot Classis. Section II describes the development of wheel speed calibration, odometry estimation, a wheel speed PID controller, and a motion controller. Section III introduces our SLAM implementation, covering mapping, localization, and particle filters. Section IV presents the planning and exploration modules, including obstacle distance, A* path planning, frontier-based exploration, and localization from unknown starting positions. Each section includes our implementation methods, evaluation results, and discussion on potential improvements.

II. Controllers

A. Wheel Speed Calibration

1) **Methodology:** Before starting to run our robot, we have to perform the calibration. During the calibration process, varied PWM commands are sent to each wheel motor, and the corresponding wheel speeds are recorded. The calibration data provides: encoder polarity, motor polarity, and the slopes m and intercepts b that define the relationship between the PWM duty cycles and the actual speeds of the wheel:

$$\text{PWM} = m \times \text{speed} + b \tag{1}$$

This relationship allows us to know the motor punch that could help with handling motor frictions and it could also serve as future reference in the feed-forward controller for our wheel speed control.

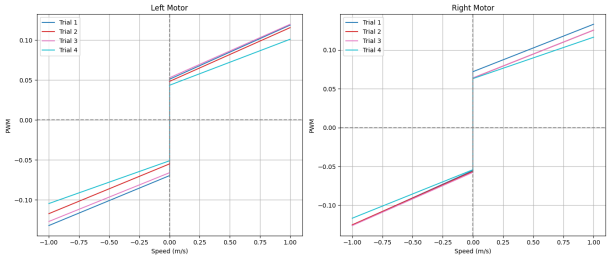

Fig. 1. Motor calibration results. Plots of PWM versus speed for left wheel motor (left) and right wheel motor (right).

2) **Results and Discussion:** We performed the calibration 4 times using the same MBot on a concrete floor in the lab. The data are shown in Table IX, the mean and variance of slopes and intercepts for each wheel motor are summarized in Table I, and the plots of PWM versus speed are shown in Fig. 1. We can see that the variance of positive/negative slopes and intercepts for both wheel motors are less than 0.0001, which is very

small. The potential sources of variation could be caused by: different areas in the lab might have slightly different friction, the mechanical mounting missalignment and condition of the wheels, battery voltage level, sensor noise. Therefore, performing calibration before running the MBot is critical for ensuring accurate motor control.

TABLE I
MOTOR CALIBRATION RESULTS WITH VARIANCE

	Positive Slope		Negative Slope		Positive Intercept		Negative Intercept	
Motor	**Mean**	**Var.**	**Mean**	**Var.**	**Mean**	**Var.**	**Mean**	**Var.**
Left	0.065	2.39E-05	0.059	1.82E-05	0.049	1.67E-05	-0.061	7.76E-05
Right	0.059	1.65E-05	0.068	1.20E-05	0.066	1.77E-05	-0.056	2.06E-06

B. Odometry

1) **Methodology:** For a mobile robot, accurate odometry estimation is crucial to allow the robot to determine its position and orientation while moving. In the original MBot code base, odometry is estimated using encoder readings to calculate wheel velocity and update the MBot's pose using dead reckoning equations. However, in a real robot, various systematic and non-systematic factors, such as wheel slipping and travel over unexpected objects, or uneven floor, can introduce errors in odometry estimation. To mitigate these errors, incorporating an IMU to improve odometry estimates is a common approach. We implemented Gyrodometry [1] to achieve this. When the difference between the angular velocity measured by the gyroscope (in the IMU) and the angular velocity estimated by odometry exceeds a threshold **0.005**, we trust the gyro measurement instead; thus, updating the robot angular velocity estimation based on gyro readings.

TABLE II
ODOMETRY ERROR WITH AND WITHOUT GYRODOMETRY

Pose	**Abs. Mean Odometry Pose Error**
x (w/o Gyrodometry) [m]	0.0040
y (w/o Gyrodometry) [m]	0.0091
θ (w/o Gyrodometry) [deg]	6.6513
θ (w/ Gyrodometry) [deg]	2.7478

2) **Results and Discussion:** To evaluate the performance of our odometry system, we commanded our robot to move forward or rotate at specific linear or rotational speeds, measured the actual pose difference, and compared it with the odometry readings. Table II shows the the results of the evaluation. Without Gyrodometry, the absolute mean odometry pose errors in x and y are less than 0.001m (1cm), which is sufficiently accurate, so no further improvement was necessary. However, the error in θ without using Gyrodometry reached 6.65 degrees, which is not ideal. After implementing Gyrodometry for improvement, the error in θ decreased to 2.75 degrees, indicating our improvement was effective.

C. Wheel Speed PID Controller

1) **Methodology:** Our final wheel speed controller is implemented based on the MBot_ControlLoop handout. In the main control loop in `mbot_classic.c`, we first use `mbot_ctlr()` to control MBot's body velocities, then compensated both commanded and measured wheel velocities using the motor polarities. Next, use `mbot_motor_vel_ctlr()` to perform PID control on the motor velocities and use PWM from calibration (we obtained the pwm-speed relationship from calibration) as feedforward to enhance the motor control. In addition to tuning PID parameters of PID controllers, we noticed that when changing linear or rotational velocity, large accelerations or decelerations caused abrupt and unstable robot movements and deviations from the expected path. Therefore, we limited the acceleration and deceleration by applying low-pass filters to both linear velocity command (v_x cmd) and angular velocity command (ω_z cmd) in the body velocity controller.

TABLE III
CONTROLLER PARAMETERS

PID Param.	**Kp**	**Ki**	**Kd**	**Kp**	**Ki**	**Kd**	**Filter Param.**	**Time Const.**
Right Wheel	0.35	0.015	0.005	0.3	0.00	0.002	V_x cmd	0.2
Left Wheel	0.35	0.015	0.005	0.3	0.00	0.002	ω_z cmd	0.55
V_x	0.35	0.015	0.005	0.3	0.00	0.002	-	-
ω_z	0.35	0.002	0.002	0.3	0.00	0.002	-	-

Table III shows the parameters in our wheel speed controller. The 2nd to 4th columns are the PID parameters we submitted for checkpoint 1. While these settings provided good performance for the speed control, we observed that whenever the robot reached the goal, the wheel motors continued to move and oscillate. This issue is mainly due to the integral gain (Ki) in the PID controller, which accumulates past errors. When the robot reaches the goal, any small residual error can cause the accumulated integral to overshoot, and the controller will continue to apply non-zero command to the motors, resulting in unintended movement and drift. Therefore, we retuned the PID parameters and set the integral gain (Ki) to 0, as shown in the 5th to 7th columns, which are our final settings. For the proportional gain (Kp), we found that a too-high Kp could cause the system to oscillate. If the robot started oscillating, we slightly reduced the P gain to stabilize its behavior. For the derivative gain (Kd), we use it to further stabilize the system; however, if Kd is too large, it might amplify noise. Thus, we set Kd to 0.002, which provided good performance. For the

low-pass filters on velocity commands, we tuned the time constant. A smaller low-pass filter time constant makes the system follow commands more quickly but less stably, while a larger time constant results in smoother and more stable behavior but slower response. During our experiments, when driving the MBot around a 1m square, we observed that large angular accelerations caused greater path deviation, so we set the time constant for ω_z as 0.55 and v_x as 0.2.

2) **Results and Discussion:** To evaluate the performance of our controllers, we drove the MBot in the maze with both slow speed ($v_x = 0.2$ [m/s], $\omega_z = \pi/4$ [rad/s]) and high speed ($v_x = 0.8$ [m/s], $\omega_z = \pi$ [rad/s]). As shown in Fig. 12 and Fig. 14, the robot was able to move, turn, follow paths accurately at different speeds. In comparison to the trials before our improvement (see Fig. 11 and Fig. 13), where the robot deviated from the path each time it turned due to the lack of acceleration limits. These results evident that the improved controllers showed better stability and precision.

D. Motion Controller

1) **Methodology:** Initially, we applied the RTR (Rotate α, Translate d, Rotate β) controller for motion control. It can be shown that the forward velocity v and rotational velocity ω can be controlled as: $v = K_d d$ and $\omega = K_\alpha \alpha + K_\beta \beta$. The parameters shown in Table IV allow our robot to have great performance.

We also implemented the Pure Pursuit motion control algorithm to ensure that the robot follows a smooth curve within the waypoints. Instead of aiming directly at the waypoint, the pure pursuit connects the robot (x_r, y_r) and the goal (x_g, y_g) with an arc. The pure pursuit controller controls the velocity as: $v_x = K_x L$ and $\omega_z = K_\omega / r$, where $L = \sqrt{x_L^2 + y_L^2}$ is the distance between the robot current point and the target point and $r = L^2/2y_L$ is the radius of curvature (r) required to steer the robot toward the lookahead point. An additional improvement we made is that we set $L = 0.4$ instead of dynamically and frequently adjusting the velocities, so our mbot can move more smoothly. Also, we found that x_L and y_L cannot be calculated directly as $(x_g - x_r)$ and $(y_g - y_r)$, this is because when the mbot turns 90 degrees, x and y will be opposite, making the movement of the MBot go wrong after turning. Thus, we used $x_L = L\cos\alpha$ and $y_L = L\sin\alpha$ to calculate x_L and y_L, where α is the lookahead heading error. The parameters K_x, K_ω for the pure pursuit controller are shown in Table IV.

2) **Results and Discussion:** Fig. 2 shows the odometry of the MBot driving around a 1m square 4 times before and after applying improvements to our wheel speed and motion controllers. As shown, before our improvements (Fig. 2ac), the robot deviated from the path over time, with a noticeable increase in path deviation after each turn. After the improvements (Fig. 2bd), the robot can follow the path almost perfectly. In our real test, our MBot successfully returned to its initial position after completing 4 squares. Fig. 3 shows the linear and rotational velocity as the robot drives 1 loop around the square. It is evident that our controllers provide stable control for both linear and rotational velocities.

TABLE IV
MOTION CONTROLLER PARAMETERS

Controller	Parameters
RTR Controller	$K_d = 1.0, K_\alpha = 3.5, K_\beta = -0.2$
Pure Pursuit Controller	$K_x = 1.0, K_\omega = 2.0$

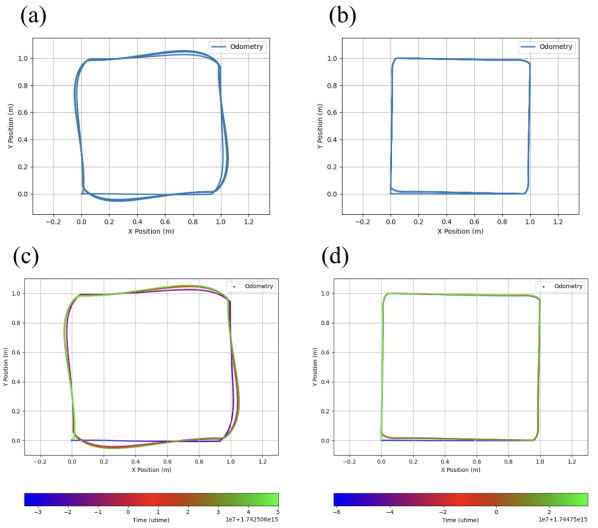

Fig. 2. Estimated odometry pose of our MBot driving a 1m square four times. (a) and (c) show results before applying low-pass filtering and pure pursuit control, while (b) and (d) show results after. (a) and (b) use a single color; (c) and (d) use a time-based color gradient.

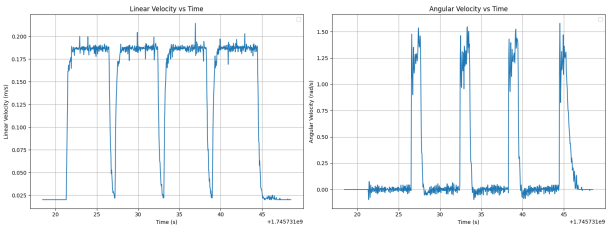

Fig. 3. Plots of the robot's linear (left) and rotational (right) velocity as it drives one loop around the 1m square.

III. SIMULTANEOUS LOCALIZATION AND MAPPING

A. Mapping

1) **Methodology:** The first step in our SLAM system was to create an occupancy grid map of the robot's

environment. Each cell represents the possibility it is being occupied or free. If a laser beam from the robot hits an obstacle, that cell becomes more likely to be marked as "occupied." If a beam passes through a cell, it becomes more likely to be marked as "free."

Log-odds were used to represent the occupied probability in each cell, ranging from -128 to 127, with 0 meaning completely unknown, negative values meaning more likely to be free, and positive values meaning more likely to be occupied. This format also facilitates efficient computation and reduces the likelihood of rounding errors. For each laser scan, we updated the map in two parts: Hit cells (where the laser ended) were updated by increasing the cell's log-odds value, and Passed-through cells (those between the robot and the hit cell) had their log-odds decreased. To figure out which cells were passed through by a beam, we implemented Bresenham's line algorithm, a classic method from computer graphics for drawing straight lines on grids. This ensures that every cell along the beam's path gets updated properly.

We also accounted that our LIDAR sensor is relatively slow. When the robot moves, the origin of the laser rays is also going to move. Therefore, we used previous and current robot poses and time stamps to find the actual origin for each beam, which is implemented in `MovingLaserScan()`. This is especially important for accurate mapping when the robot is in motion, and would also be used in sensor model.

2) **Results and Discussion:** Fig. 4 shows the result of our system mapping the environment from `drive_maze.log`. White regions represent free space through which the laser beams passed. Black regions represent occupied space where the beams hit walls or other obstacles. Gray regions are unknown areas the robot hasn't observed yet. As shown, some walls in the map appear blurry (e.g., bottom right) due to the robot not staying in that place long enough for full map convergence, or unstable robot movement causing blurry boundaries (e.g., bottom left). However, the overall map has clear walls and well-defined boundaries of the maze, indicating that our occupancy grid mapping algorithm accurately reconstructed the environment. (The white shadow in unobserved areas is due to issues with the provided log file.)

One challenge in this part was tuning the parameters for how much to increase or decrease the log-odds when a beam hit or missed. If increasing too much, the map became noisy with overconfident walls. If decreasing too much, the free space was underestimated. After testing with several log files, we found that setting `kHitOddsArg` parameters to 3 usually provides stable maps with minimal noise, but when the robot moves fast, setting `kHitOddsArg` parameters to 5 could lead to better mapping performance. In the future, we could improve this part of the SLAM system by integrating obstacle distance penalties or adjusting update rates dynamically based on how quickly the robot is moving.

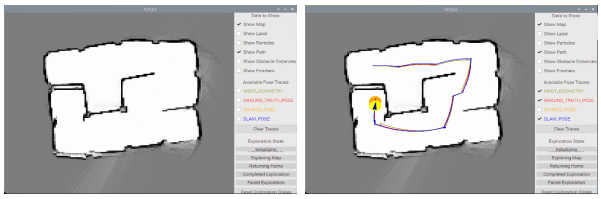

Fig. 4. Occupancy grid map generated from drive_maze.log. The robot's estimated path (blue) aligns closely with the ground-truth path (red), and the map represents the maze structure.

B. Localization

For localization, we implemented a Monte Carlo Localization (MCL) system, which uses a particle filter to estimate the robot's position within a known map. This method maintains a set of particles, as the robot moves and receives sensor data, these particles are updated using the action model and the sensor model to reflect more accurate beliefs about the robot's position.

1) **Action Model:** The action model estimates how the robot has moved between two time steps. It uses data from the odometry poses to calculate the change in position and orientation. Since odometry estimation and wheel movements aren't perfect, we introduce randomness into the motion estimate to simulate noise and uncertainty. For each particle, we add some variation to its movement using a Gaussian distribution, which helps the filter remain robust even when the robot's movement is not perfectly predictable. This randomness is controlled by tuning parameters based on the robot's translational and rotational movement. The action model predicts how the robot moves over time using odometry data. It calculates the change in the robot's position $(\Delta x, \Delta y)$ and orientation $(\Delta\theta)$ between two timestamps. Using the rotate-translate-rotate motion rule, where δ_{rot1} is the initial rotational change to face the direction of motion, δ_{trans} translational change along that direction, and δ_{rot2} is the final rotational change to reach the new orientation. The action model we utilized is as follows:

$$\begin{aligned} \delta_{rot1} &= \text{atan2}(\Delta y, \Delta x) - \theta_0 \\ \delta_{trans} &= \sqrt{(\Delta x)^2 + (\Delta y)^2} \\ \delta_{rot2} &= \Delta\theta - \delta_{rot1} \end{aligned} \tag{2}$$

$$\begin{aligned} \hat{\delta}_{rot1} &= \delta_{rot1} - \textbf{sample}(k_1\delta_{rot1}^2) \\ \hat{\delta}_{trans} &= \delta_{trans} - \textbf{sample}(k_2\delta_{trans}^2) \\ \hat{\delta}_{rot2} &= \delta_{rot2} - \textbf{sample}(k_1\delta_{rot2}^2) \end{aligned} \tag{3}$$

where **sample**() is Gaussian distributions that introduce uncertainties to the action. The new particle pose is

updated as:

$$\begin{aligned} x' &= x + \hat{\delta}_{trans} \cdot \cos(\theta + \hat{\delta}_{rot1}) \\ y' &= y + \hat{\delta}_{trans} \cdot \sin(\theta + \hat{\delta}_{rot1}) \\ \theta' &= \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \end{aligned} \tag{4}$$

2) **Parameters in Action Model:** For the parameters in our action model, k_1 reflects the uncertainty in rotation and k_2 reflects the uncertainty in translation. Therefore, we tested our action model with different log files to see whether the localization when turning is unstable (tune k_1) or moving forward is unstable (tune k_2) to balance the localization responsiveness and stability. We found that smaller values lead to less noisy predictions but risk convergence failure, while larger values spread the particles too much. Additionally, the action model should not update the particles if the robot is not moving. We set parameters for minimum translation and rotation threshold to prevent the action model too sensitive to some slightly unstable robot movement. After some experiments, the parameters we choose to use are shown in Table V. These values ensure our particle filter could operate effectively without overly aggressive spreading or collapse, resulting in a good localization performance.

TABLE V
PARAMETERS FOR THE ACTION MODEL

Parameter	k_1	k_2	Min. Translation [m]	Min. Rotation [rad]
Values	0.01	0.03	0.0025	0.02

3) **Sensor Model:** The sensor model in our Monte Carlo Localization (MCL) implementation plays a crucial role in evaluating how well each particle's predicted pose matches the actual environment. It assigns a likelihood (or weight) to each particle based on how well a simulated laser scan from that pose aligns with the map. To achieve, we simulate each ray in a laser scan using the particle's pose and trace the expected endpoint in the occupancy grid. If the simulated endpoint lands in an occupied cell (i.e., a wall), it is considered a match. A higher number of matched endpoints results in a higher weight for the particle. This process is repeated for each particle during every update cycle.

4) **Sensor Model Optimizations:** To further speed up computation without significantly sacrificing accuracy, we also tested the following strategies: (1) **Ray Stride:** Instead of processing every single laser ray, we used a stride of 7 (i.e., every 7th ray) to reduce the number of calculations. This reduces computation time while still capturing the overall scan shape. (2) **Maximum Ray Range:** We limited ray evaluations to a maximum range of 1000 (in grid units), beyond which returns are ignored. This avoids wasting time on long-distance rays which often do not provide meaningful localization data. These optimizations are crucial in making the sensor model scalable to a higher number of particles while keeping the system real-time capable.

5) **Particle Filter:** The workflow of our particle filter is shown in Fig. 7 updateFilter(). Using the odometry estimates, the action model updates action and compute new distribution. If the robot moves, particles will be resample the high weighted particles from posteriors using `importanceSample` function to get priors, then the action model disperses (resample) the priors with its action for proposals. Next, sensor model use the likelihood to weight each particles using laser and map knowledge and get posteriors. Finally, we average the best 30% particles to estimate the posterior pose.

6) **Particle Filter Performance within 10Hz:** To evaluate the efficiency of our particle filter, we measured the time required to update the particles for different numbers of particles. For a system running at 10Hz, we need the total update time per cycle to be less than 0.1 seconds. Our initial results are shown in Table X and Fig. 15. After improving our action and sensor models. The final results are summarized in Table VI. We plot the update time versus the number of particles (using 100 to 3000 particles since 5000 exceeds 0.1 sec) to observe their relationship (see Fig. 5). The linear fitting results indicate that the update time is linearly correlated (with $R^2 = 0.9890$) with the number of particles. Using the fitting result $y = 0.0000166x - 0.00231$, we estimate the maximum number of particles our system can support while running comfortably at 10Hz is **6163 particles**.

TABLE VI
TIME TO UPDATE THE PARTICLE FILTER

Number of Particles	100	500	1000	2000	3000
Update Time [sec]	0.0017	0.0056	0.0123	0.0289	0.0495

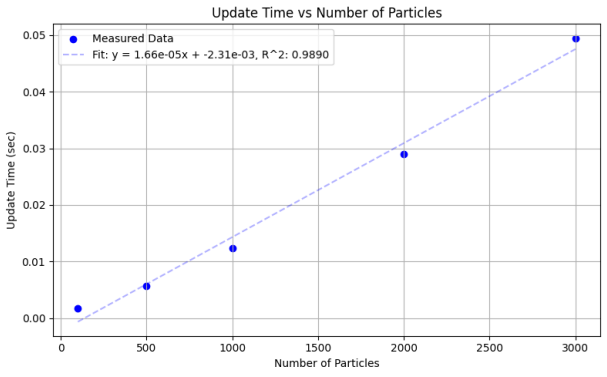

Fig. 5. Plot of particle update time versus number of particles.

Fig. 6 shows the particle distribution (300 particles) at regular intervals using the `drive_square.log` file.

This visualization shows that as the robot moves, the particles will be updated (spread out) by action model, then using sensor model, particles converged around the robot's actual path and orientation. The cloud of particles was typically dense and converged when the robot had good sensor visibility, followed consistent motion, and did not move too fast.

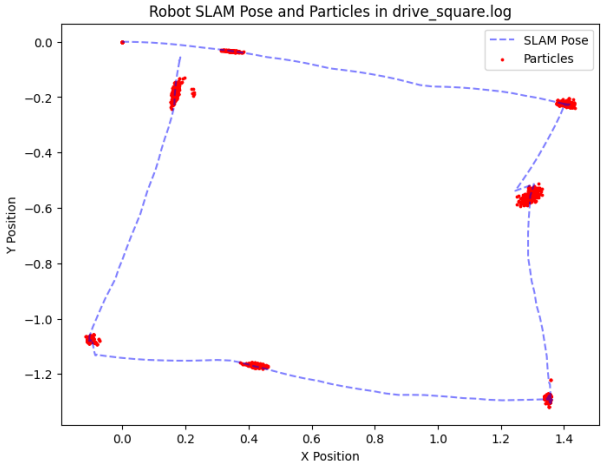

Fig. 6. Robot SLAM pose and 300 particles along the path using our particle filter on drive_square.log.

C. SLAM

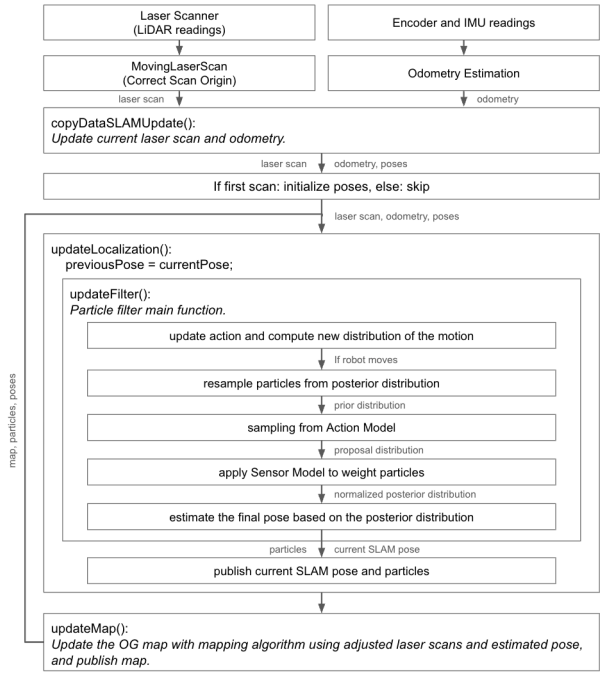

Fig. 7. SLAM block diagram.

1) **Methodology:** Fig. 7 shows our SLAM system overview. The SLAM algorithm combines both Monte Carlo Localization (particle filter) and occupancy grid mapping. For every incoming laser scan, the robot iteratively estimates its position using the localization module and then updates the occupancy grid map with new pose.

2) **Results and Discussion:** We ran our system with `drive_maze_full_rays.log`, and compared the SLAM-estimated pose against the ground-truth pose, odometry pose, and the source SLAM pose provided in the log file at each corresponding timestamp. A visualization of the comparison is shown in Figure 8. We can see that the SLAM pose starts deviate from the ground-truth pose at around (1.625, -0.7), where the odometry pose seems to have some errors. Yet, our SLAM pose is closely align with source SLAM pose and odometry pose. To evaluate pose accuracy quantitatively, we computed the mean errors, maximum errors, and Root Mean Square (RMS) errors in 2D position (x-y plane), x direction, and y direction as shown in Table VII.

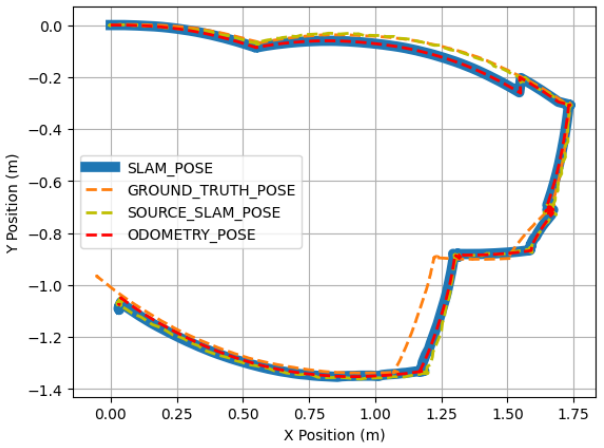

Fig. 8. Comparison between estimated SLAM pose (blue), ground-truth pose (orange), source SLAM pose (yellow), and odometry pose (red) in `drive_maze_full_rays.log`.

TABLE VII
STATISTICAL RESULTS OF THE ACCURACY EVALUATION FOR OUR SLAM SYSTEM

SLAM Pose Compared with:	Ground Truth	Odometry	Source SLAM
Mean Error [m]	1.1572	1.1326	0.0287
Max. Error [m]	1.6311	1.8405	0.0788
RMS Error [m]	1.2317	1.2140	0.0369
RMS Error In X [m]	0.9558	0.9725	0.0303
RMS Error In Y [m]	0.7769	0.7267	0.0209

These results indicate that our SLAM implementation is reasonably accurate, though some deviation is present likely due to accumulated error in pose estimation over time and the odometry estimation errors in the log file.

Improving the performance further may require fine-tuning the parameters in action and sensor models, applying beam model to enhance sensor model, and increasing the number of particles during localization.

Additionally, to assess the effectiveness of our full SLAM pipeline in real-world scenarios, we ran our implementation in the maze in the lab. Figure 9 shows a snapshot of the constructed map with the estimated path using SLAM. The trajectory appears smooth and well-aligned with the odometry and the defined paths, and the walls in the constructed map is clear and precise, showing consistent and accurate mapping and localization performance of our SLAM system.

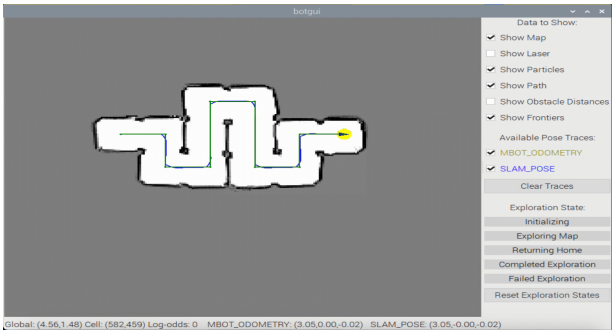

Fig. 9. SLAM path (blue) compared with odometry (yellow) and ground truth (green) in maze in the lab.

IV. Planning and Exploration

A. A* Path Planning

To enable our MBot to plan a path from one pose to another within the environment, we implemented the A* path planning algorithm [2], as shown in Algorithm 1. A* is a graph search algorithm that computes the optimal path by minimizing a cost function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cumulative cost from the start node to current node n, and $h(n)$ is a heuristic estimate of the cost from the current node n to the goal node. A* is derived from Dijkstra's algorithm (where $h(n) = 0$) and Greedy Best-First Search (where $h(n) \gg g(n)$), and is guaranteed to find an optimal solution in a short time when an admissible heuristic is used [3].

1) **Heuristic Function:** The heuristic function $h(n)$ estimates the heuristic cost (h_cost) from the current node to the goal. Defining an appropriate heuristic function is critical for the A* algorithm. If $h(n)$ is **admissible**, meaning $h(n)$ never overestimates the true cost to the goal, it can be proven that A* will find an optimal path [3]. However, if $h(n)$ is underestimated, it may result in longer search times to find the optimal path. On the other hand, if $h(n)$ overestimates the true cost, A* may find a solution faster, but it will not guarantee optimality. Since our system can move in diagonal directions, we adopted the 8-way Diagonal Distance as our heuristic function:

$$h(n) = \Delta x + \Delta y + (\sqrt{2} - 2)min(\Delta x, \Delta y) \quad (5)$$

where $\Delta x = |n_x - g_x|$, $\Delta y = |n_y - g_y|$, $n = (n_x, n_y)$ is current node, and $g = (g_x, g_y)$ is the goal node.

2) **Cost Functions and Obstacle Avoidance:** A* algorithm explores the node with the minimum f_cost, where f_cost is computed as $f(n) = g(n) + h(n)$. Here, the g_cost $g(n)$ represents the accumulated cost to reach the current node n, we calculated the $movement\_cost$ using 8-connected movement cost (1.0 for straight travel cost and 1.4 for diagonal travel cost). Yet, a challenge in A* arises in environments with obstacles, where the optimal path often tends to follow the edges of the obstacles. In real-world scenarios, this can increase the risk of collisions due to uncertainties in the robot pose and map. To mitigate this issue, we incorporate obstacle distance using the Brushfire algorithm [4] within the g_cost, which penalizes the paths that are too close to the walls. Thus, our g_cost is calculated as:

$$g(n) = movement\_cost + obs\_dist\_penalty \quad (6)$$

3) **Path Planning:**

a) Initialization: We maintain an open list to store unexplored nodes and a closed list to store explored nodes. First of all, A* initializes the cost of the start node as $g\_cost = 0$ and $h\_cost = h(start)$ and adds the start node into the open list.

b) Exploring a New Node: When the open list is not empty, A* selects the node with the lowest f_cost as the current node n to explore. In our implementation, we used a priority queue as the data structure for the open list to efficiently store and retrieve unexplored nodes with the lowest f_cost. If the current node n is the $goal$, the search terminates and a path is constructed using the `extract_node_path()`. Otherwise, A* adds n to the closed list and begins exploring its neighbors.

c) Expanding Neighbor Nodes: A* explores the neighbors (denoted as $kids$) of the current node n by expanding n using an 8-connected rule. For each unexplored neighbor kid (i.e., not in the closed list), if the kid is not in the open list or has a lower g_cost (indicating a better path) than the one recorded in the open list, this kid is either added to the open list or has its costs and parent information updated in the open list.

d) Extracting Path from Nodes: By iterating previous steps, if the goal node is reached, the `extract_node_path` reconstructs the solution path by iteratively following the parent pointers from the goal node back to the start node. The sequence of nodes is collected into a vector, reversed to obtain the correct start-to-goal order, and returned as the final planned path. If the open list becomes empty but the goal is not reached, it indicates that a valid path was not found.

4) **Pruning:** When our A* algorithm generates a path on the occupancy grid map, the path consists of discretized grid cells, leading to an unsmooth trajectory. In real-world scenarios, this can cause frequent or abrupt changes in direction, resulting in unstable movement for the MBot. To address this issue, we implemented the `prune_node_path()` function to smooth the path. We applied the cross product to check the collinearity of three consecutive nodes (previous, current, and next) and remove the unnecessary nodes that lie on the same line, thereby optimizing the path.

5) **Results and Discussion:** Figure 10 shows the path (green nodes and lines) planned by our A* implementation and the actual path (blue curve) executed by our MBot. It demonstrates that our robot can effectively avoid obstacles and find an optimal path to the designated goal. Table VIII summarizes the timing information for our A* path planning across various test cases. Overall, our A* algorithm is able to find an optimal path and perform efficiently in most scenarios. During successful planning attempts, the mean pathfinding time for each case is less than 1.0E+06 μs (1 second). For failed attempts, most test cases complete within 100 μs (0.0001 seconds). Note that in successful attempts, the `test_wide_constriction_grid` case takes significantly longer time (up to 2.55E+06 seconds). Possible reasons are that in a wide constriction environment, A* performs extensive node expansion to find a valid path. Also, the 8-way diagonal distance heuristic may not be aggressive enough. Similarly, in failed attempts, `test_narrow_constriction_grid` shows long termination times for the similar reasons. A potential strategy for improvement is to use a more aggressive admissible heuristic or apply weighting to the heuristic, which can help guide the search more effectively in constricted environments. Yet, we should also consider that a too aggressive heuristic might make A* not find an optimal path. A demo video showing our MBot planning path and avoiding obstacles to defined goal: A* Demo.

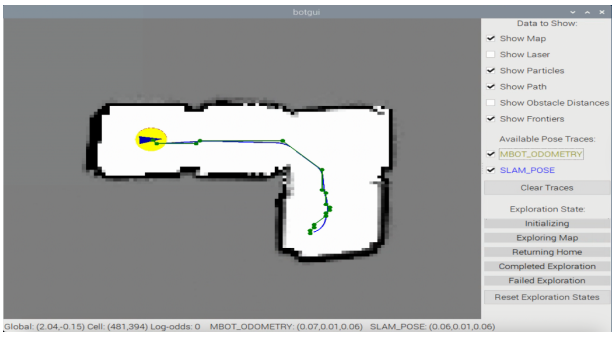

Fig. 10. Path planned by A* (green) and the actual path driven by our MBot (blue).

TABLE VIII
A* PATH PLANNING EXECUTION TIMES (IN μS)

Successful Planning Attempts					
Test Case	**Min**	**Mean**	**Max**	**Median**	**Std dev**
test_convex_grid	253	1435	2617	0	1182
test_empty_grid	1061	1729.67	2415	1713	552.894
test_maze_grid	4617	7012.75	7906	4617	1384.46
test_narrow_constriction_grid	1019	1086	1153	0	67
test_wide_constriction_grid	1025	8.51E+05	2.55E+06	2.55E+06	1.20E+06
Failed Planning Attempts					
Test Case	**Min**	**Mean**	**Max**	**Median**	**Std dev**
test_convex_grid	20	25.5	31	0	5.5
test_empty_grid	22	43	64	0	21
test_filled_grid	15	26.2	42	24	8.95321
test_narrow_constriction_grid	14	9.24E+06	2.77E+07	23	1.30E+07
test_wide_constriction_grid	18	18	18	0	0

B. Map Exploration

While our Mbot can operate and perform SLAM under manual control or following predefined waypoints and paths, the ability to autonomously navigate, explore, and map an unknown environment could significantly enhance the robot's autonomy and offer benefits for various real-world applications. To achieve this, we employed a frontier-based exploration that enables the Mbot to autonomously plan its exploration path and perform SLAM in unknown environments.

1) **Frontiers Detection:** Frontiers are defined as regions that lie between known free space (grid's log-odds < 0) and unknown areas (grid's log-odds $= 0$) of the map. To find all the frontiers within the current map, the search begins from the robot's current position. A connected components search, implemented using a Breadth-First Search (BFS) algorithm, is then employed to identify and connect all reachable frontier cells. These connected frontiers facilitate the selection of the robot's next exploration target during the exploration process.

2) **Next Goal Selection and Navigation:** To select and plan the next exploration path, the centroids of all detected frontiers are first computed. These centroids are then sorted in ascending order based on their distance to the robot's current pose. For each centroid, a local BFS is performed to find a candidate navigation goal that is within the map and not an occupied grid. Using the previously implemented A* path planner, we evaluate whether the candidate navigation goal is a valid goal that the robot can reach. If a valid path is found, it is selected as the next exploration path. This strategy allows the robot to efficiently and effectively explore the closest reachable frontier while avoiding unreachable regions.

3) **Exploring the Map:** During exploration, the exploration algorithm detects frontiers in the current map using `find_map_frontiers()`. If the robot is within a target reach threshold of 0.1 from the current target and unexplored frontiers still exist, it plans a new current path using `plan_path_to_frontier()` and

assigns the final position of this current path as the new exploration target. A path length > 1 indicates that exploration is in progress. If no frontiers remain, meaning all frontiers have been explored, the exploration is complete. Otherwise, if unexplored frontiers still exist but no valid path can be found, the exploration is failed. Note that when the exploration is nearly completed, although no frontiers are left, the map may not yet converged. To address this, we insert a `usleep`(3000000) delay that allows the robot to pause for 3 seconds to ensure that the final portion of the map is properly constructed. Once the exploration is complete, MBot will set the initial starting pose as the home pose and navigate back to return home. The result of our exploration is shown in Fig. 16 and a demo video: Exploration Demo.

C. Map Localization with Unknown Starting Position

If the robot's initial position is unknown, we cannot rely on the localization method described in Section III. Instead, we enhance our particle filter by initializing the particles randomly across the map. As the robot moves and gathers more observations, the particles are updated and gradually converge toward the true robot pose, utilizing action model and sensor model. Specifically, particles that are inconsistent with the sensor observations are removed, while those with higher weights (consistent with the observations) are retained. To improve the performance in this task, a further enhancement of sensor model is needed by applying the beam model to score rays in different cases, allowing the particle weights to be properly adjusted based on the sensor observations. With these improvements, the robot will be able to estimate its pose in the provided map and perform initial localization without prior knowledge of its starting position.

V. Conclusions

The BotLab project provided a comprehensive hands-on experience with the MBot Classic. Our experimental results and evaluations demonstrated the high performance and accuracy of our wheel speed and motion controllers, SLAM system, A* path planning, and exploration algorithms. Although we did not have time to complete tasks in the competition, we can confidently conclude that our implementation is effective and holds great potential for success in the competition. Future work will be improving the robustness of our SLAM and computational efficiency of A*, and completing competitions to validate our implementations.

References

[1] J. Borenstein and L. Feng, "Gyrodometry: A new method for combining data from gyros and odometry in mobile robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1. IEEE, 1996, pp. 423–428.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968.

[3] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.

[4] J. Barraquand and J.-C. Latombe, "Robot motion planning: A distributed representation approach," *The International journal of robotics research*, vol. 10, no. 6, pp. 628–649, 1991.

VI. Appendices

A. Additional Results for Controllers

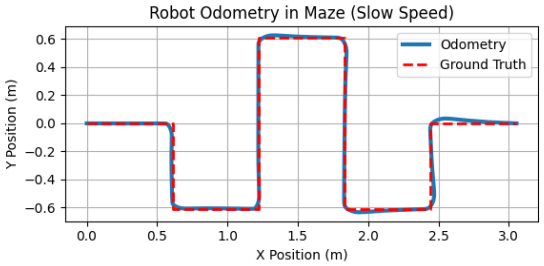

Fig. 12. Drive maze result in slow speed (0.2 [m/s] and $\pi/4$ [rad/s]) after implementing low-pass filters on commands and an advanced motion controller.

TABLE IX
Motor Speed Calibration Data

Parameter	Trial 1	Trial 2	Trial 3	Trial 4
Motor Polarity	-1	-1	-1	-1
	-1	-1	-1	-1
Encoder Polarity	-1	-1	-1	-1
	-1	-1	-1	-1
Positive Slope	0.067815	0.067319	0.066945	0.057611
	0.061130	0.061680	0.061609	0.053372
Positive Intercept	0.050854	0.048141	0.052721	0.043328
	0.072003	0.063855	0.063942	0.063096
Negative Slope	0.062449	0.062237	0.061016	0.053471
	0.070303	0.068879	0.068549	0.062470
Negative Intercept	-0.069765	-0.055105	-0.066102	-0.051197
	-0.055411	-0.056390	-0.057694	-0.054322

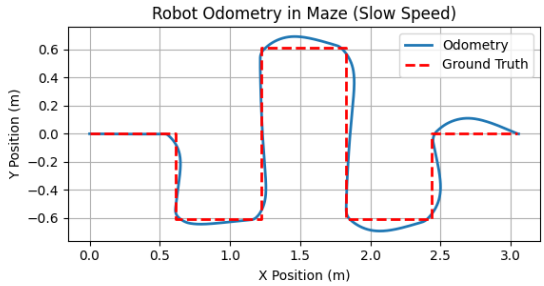

Fig. 11. Drive maze result in slow speed (0.2 [m/s] and $\pi/4$ [rad/s]) before implementing low-pass filters on commands and an advanced motion controller.

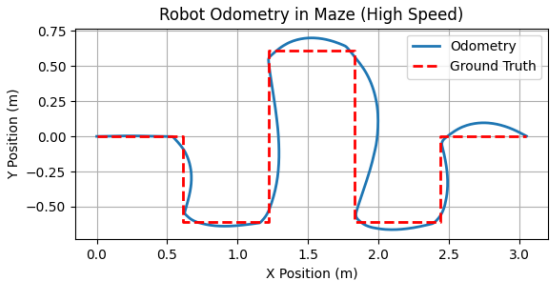

Fig. 13. Drive maze result in high speed (0.8 [m/s] and π [rad/s]) before implementing low-pass filters on commands and an advanced motion controller.

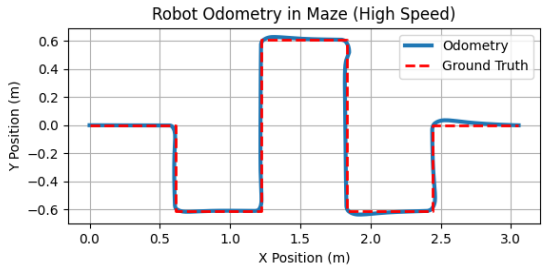

Fig. 14. Drive maze result in high speed (0.8 [m/s] and π [rad/s]) after implementing low-pass filters on commands and an advanced motion controller.

B. Additional Results for SLAM

TABLE X
TIME TO UPDATE THE PARTICLE FILTER

Number of Particles	100	500	1000	2000	2289	3000
Update Time [sec]	0.0041	0.0198	0.0406	0.0833	0.0967	0.1323

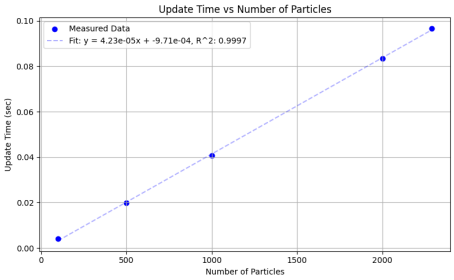

Fig. 15. Plot of particle update time versus number of particles.

C. Additional Results for Planning and Exploration

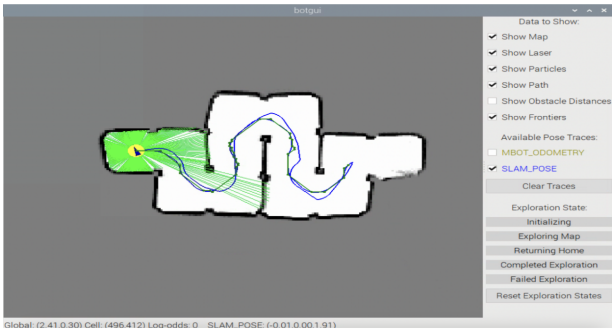

Fig. 16. Exploration result in the maze.

Algorithm 1 A* Algorithm

$start.g \leftarrow 0.0$
$start.h \leftarrow h\_cost(start, goal)$
$start.parent \leftarrow nullptr$
$open \leftarrow \{start\}$
$closed \leftarrow \emptyset$
while $open \neq \emptyset$ **do**
  $n \leftarrow \arg\min_{n \in open} f(n)$
  $open \leftarrow open \setminus \{n\}$
  if $isGoal(kid)$ **then return** $extractPath(kid)$
  $closed \leftarrow closed \cup \{n\}$
  $kids \leftarrow expandNode(n)$
  for each $kid \in kids$ **do**
    if $k \in closed$ **then** skip this kid
    $kid\_g \leftarrow g\_cost(n, kid)$
    if $kid\_g < kid.g$ or $kid \notin open$ **then**
      $kid.g \leftarrow kid\_g$
      $kid.h \leftarrow h\_cost(kid, goal)$
      $kid.parent \leftarrow n$
      if $kid \notin open$ **then** $open \leftarrow open \cup \{kid\}$
    end if
  end for
end while
return failed to find a path