ROB 550: BotLab Report

Atharva S. Kashyap, Yu-I Chang, Rajatsurya M
Robotics Department, The University of Michigan,
Ann Arbor, MI 48109
{katharva, yuic, rajatsm}@umich.edu

***Abstract*—This report details the process of learning the fundamental techniques of robot control, mapping, and autonomous navigation through programming an MBot, a mobile robot platform. We developed a methodology that integrates Simultaneous Localization and Mapping (SLAM), A* path planning, and frontier-based exploration to allow the robot to autonomously navigate and map unknown environments. Through this experience, we gained valuable insights into autonomous systems, laying a strong foundation for future work in mobile robotics.**

I. Introduction

MOBILE robots working autonomously have been changing the face of many industries, including manufacturing, shipping, and driving [1]. For instance, mobile robots are deployed to work with employees to move heavy articles in Amazon warehouses to not only improve safety, but also increase efficiency [2]. Companies like Waymo, Cruise, and Uber are using mobile robotics technology to introduce autonomous driving taxis in several metro cities in the US [3].

In this project, the goal was to use a mobile robot, MBot as seen in Fig 1, to develop a warehouse solution where the robot would perform the following tasks. Firstly, it is developed to navigate around obstacles in a given maze at several speeds and also explore and navigate around unknown worlds. Secondly, it is designed to detect april-tag based mini crates and mini cones and move them to designated drop off zones, simulating a warehouse environment.

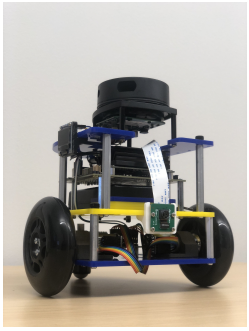


Fig. 1: MBot Classic

II. Methodology and Discussion

To accomplish the task of exploring and navigating a maze by the MBot, we needed to perform three steps: (A) Controller & Odometry, (B) Simultaneous Localization and Mapping (SLAM), and (C) Planning and Exploration.

As part of the Controller & Odometry, the goal is to control the robot's position and for this, we send commands to change the motor's velocity. SLAM helps with not only localizing the robot (the current position), but also map the environment within which the robot operates. Lastly, Planning and Exploration helps the robot navigate around a given map and explore unseen environments.

A. Controller and Odometry

a) Wheel Speed Calibration: To control the rotational speed of a brushed DC motor, we utilize an H-bridge circuit driven by PWM (Pulse Width Modulation) signals. PWM effectively regulates the average voltage applied to the motor by adjusting the duty cycle, thereby modulating the motor's speed. As an initial step in our motor control system, we aimed to characterize the relationship between the PWM input and the resulting motor speed.

We model this relationship using a piecewise linear equation:

$$\text{PWM} = \begin{cases} m_p \times \text{Speed} + b_p, & \text{if Speed} \geq 0 \\ m_n \times \text{Speed} + b_n, & \text{if Speed} < 0 \end{cases}$$

To determine the slope coefficients m_p and m_n, as well as the intercepts b_p and b_n, we varied the PWM duty cycle across a range of values and recorded the corresponding motor speeds using encoder feedback. The resulting linear relationship between PWM and speed is illustrated in Fig. 2.

However, the relationship between PWM and motor speed can be significantly influenced by both the motor characteristics and the friction between the wheels and the floor surface. Therefore, it is recommended to re-calibrate the system when operating with different motors or on different floor types. To evaluate the

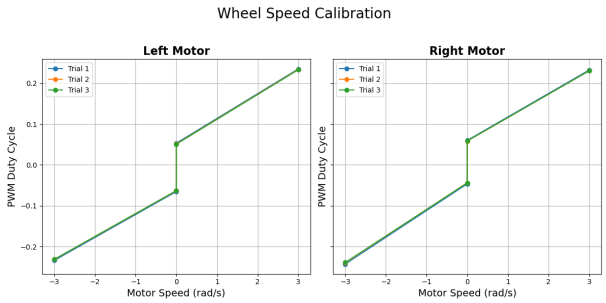


Fig. 2: Motor Speed Calibration Data

consistency and reliability of the calibration process, we repeated the same calibration procedure three times on the same type of floor but at different locations. The mean and variance of the resulting slope and intercept parameters are summarized in Table I, where the variances are observed to be very small.

Motor	Parameter	Mean	Variance
Left	m_p	0.061	9.941e-09
	b_p	0.052	1.326e-06
	m_n	0.056	6.267e-08
	b_n	-0.064	1.112e-06
Right	m_p	0.057	1.275e-08
	b_p	0.059	5.974e-07
	m_n	0.065	5.135e-08
	b_n	-0.045	1.557e-06

TABLE I: Motor Speed Calibration Data

b) Odometry: During feedback control, it's crucial for a robot to estimate its own position and orientation—also known as its **pose**. In an ideal scenario, a robot would perfectly follow the control commands it receives. However, in reality, discrepancies arise due to *sensor noise*, *environmental uncertainty*, *wheel slippage*, and *unexpected obstacles*. These challenges motivate the use of **odometry**, which estimates the robot's pose by integrating data from onboard motion sensors.

The simplest approach to odometry relies on **wheel encoder feedback** combined with a **differential-drive kinematic model**. The robot's planar linear and angular velocities can be estimated using:

$$V_x = \frac{r}{2}(v_{\text{left}} + v_{\text{right}}), \quad V_y = 0, \quad \omega_z = \frac{r}{2b}(v_{\text{right}} - v_{\text{left}})$$

where r is the wheel radius, b is the base radius, and v_{left} & v_{right} are the angular velocities of the left and right wheels.

However, wheel slippage may occur when the wheels rotate without generating actual motion. This results in wheel velocity measurements that do not correspond to real displacement, introducing significant errors into pose estimation. To mitigate this, we employ a lightweight sensor fusion technique known as **Gyrodometry**, which enhances heading estimation by integrating gyroscope data from the IMU with encoder-based odometry. In particular, when the discrepancy between the gyroscope angular velocity (ω_{gyro}) and the encoder-derived angular velocity (ω_{odometry}) exceeds a predefined threshold (e.g., 0.125 rad/s), the system replaces ω_{odometry} with ω_{gyro}:

$$\omega_z = \begin{cases} \omega_{\text{gyro}}, & \text{if } |\omega_{\text{gyro}} - \omega_{\text{odometry}}| \geq 0.125 \\ \omega_{\text{odometry}}, & \text{otherwise} \end{cases}$$

This simple yet effective thresholding strategy reduces the impact of encoder noise and wheel slippage, particularly during rapid turns or on low-friction surfaces. To evaluate different approaches, we conducted pose estimation experiments under two known conditions and compared the results of basic Odometry, Gyrodometry, and the IMU's internal fusion module. The outcome, shown in Table II, indicates that while gyrodometry improves upon raw encoder data, the IMU fusion consistently provides the most accurate heading estimates and overall better consistency across trials.

As a result, we ultimately adopted the IMU's built-in fusion method for pose estimation due to its superior orientation accuracy and reliability in real-world conditions.

	Expected	Odometry	Gyrodometry	IMU internal fusion
x (m)	1	1.0511	1.0561	1.1095
y (m)	0	0.0841	0.0258	-0.0173
θ (rad)	0	0.1320	0.0821	-0.0279
x (m)	0	0.0004	0.0004	0.0004
y (m)	0	0.0003	0.0006	0.0007
θ (rad)	3.14	3.1287	2.8580	2.8687

TABLE II: Comparison of expected and estimated poses using different sensor fusion methods.

c) Wheel Speed Control Loop: We implement a control loop that integrates feedback, feedforward, IMU fusion, and an acceleration-limiting low-pass filter to achieve robust wheel speed control, as illustrated in Fig. 3. The Raspberry Pi Pico receives velocity commands in the robot's body frame—comprising forward and angular velocities—from the upper-level computer. To prevent abrupt changes in acceleration, we apply a low-pass filter that limits the rate of change. Specifically, we constrain the maximum acceleration to $15\ \text{m/s}^2$ to mitigate slipping, and the minimum (deceleration) to $-1\ \text{m/s}^2$ to reduce the risk of the robot tipping over.

An angular velocity PID regulator is incorporated to reduce orientation error by comparing the commanded angular velocity with the value estimated from odometry. The corrected angular velocity is then combined with the forward velocity command and transformed into individual wheel velocities using differential-drive kinematics. These motor velocity commands are subsequently compared with the estimated wheel velocities obtained from encoder feedback. A second PID controller is then

used to compute the required PWM signals. Finally, a feedforward term derived from motor calibration is added to the PWM output before being sent to the motor driver.

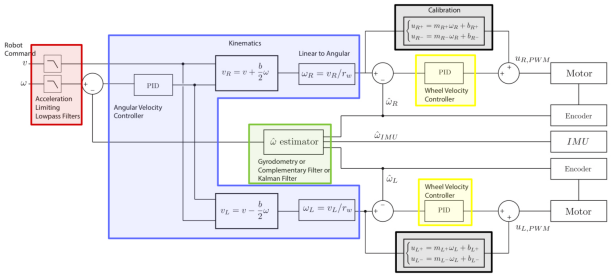


Fig. 3: Wheel Speed Control Loop

Red: Acceleration-limiting low-pass filter to smooth sudden velocity changes.
Blue: Transformation from body velocity to individual wheel velocities using differential-drive kinematics.
Green: Estimated Odometry using IMU Fusion
Yellow: PID controller for wheel velocity regulation.
Black: Feedforward compensation based on motor calibration.

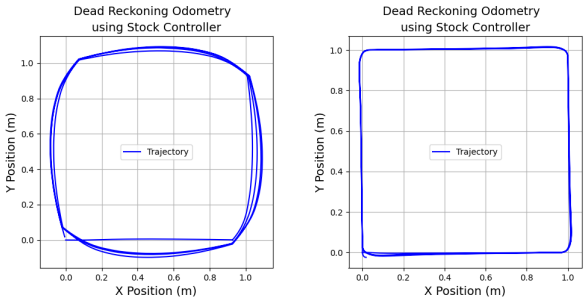
d) Wheel Speed PID Tuning and Evaluation: The PID parameters were manually tuned through a series of step input tests and open-loop velocity sweeps. The tuning process focused on several key performance metrics: tracking error (measured by the difference between commanded and actual wheel speeds), rise time, settling time, overshoot during acceleration transitions, and odometry consistency across repeated trajectories.

We began by increasing the proportional gain (K_p) to achieve an acceptable rise time and reduce steady-state error. A small derivative gain (K_d) was then introduced to dampen oscillations and minimize overshoot. The integral gain (K_i) was set to zero, as the relatively short duration of commands and the use of encoder feedback made integral correction unnecessary in our system. This tuning strategy resulted in a responsive and stable controller capable of maintaining accurate velocity tracking during motion.

Controller	K_p	K_i	K_d
Angular Velocity (ω_z)	0.15	0	0.001
Right Wheel	0.35	0	0.01
Left Wheel	0.35	0	0.01

TABLE III: Final PID controller parameters

The wheel speed controller demonstrated stable and responsive performance across various floor conditions. By combining feedforward and feedback control, the system achieved fast rise times and minimized steady-state error. The PID-based angular corrections effectively reduced orientation drift, particularly during repeated rotational maneuvers. As shown in Fig. 4, the trajectory results during repeated 1-meter square runs demonstrate the PID controller's ability to maintain accurate and consistent wheel speed control, effectively minimizing cumulative drift and enhancing motion precision.



PID Waypoint Following Pure Pursuit Controller

Fig. 4: Dead Reckoning Odometry while following a 1m square trajectory 4 times at high speed.

e) Motion Controller: To navigate a 1-meter square trajectory between waypoints, we implemented two control strategies: a stock PID waypoint-following controller and a Pure Pursuit controller. The robot was commanded to complete four loops of the square, and its dead reckoning pose was tracked throughout the execution to assess tracking performance and control stability.

The Pure Pursuit controller guides the robot by dynamically selecting a target point ahead along the desired path and steering toward it. The control laws for linear and angular velocity are defined as:

$$V_x = K_x \cdot \sqrt{dx^2 + dy^2}, \quad \omega_z = \frac{K_w}{r}$$

$$r = \frac{L^2}{2L\sin(\alpha)}, \quad \alpha = \arctan(\frac{dy}{dx}) - \theta_{pose}$$

Parameter	Value	Description
K_x	2.0	Proportional gain for linear velocity
L	1.0	Lookahead distance to the target point
K_w	4.0	Proportional gain for angular velocity
r	—	Turning radius
d_x, dy	—	Cartesian offset to waypoint
θ_{pose}	—	Orientation angle of the current pose

TABLE IV: Pure Pursuit controller parameters

As shown in Fig. 4, the MBot operated at relatively high speed limits ($V_x \approx 0.6$ m/s, $\omega_z \approx \pi$ rad/s). While both controllers completed the intended path, the Pure Pursuit controller produced significantly smoother and more accurate turns, particularly at corners, whereas the PID controller exhibited noticeable overshoot and cumulative drift.

In addition, Fig. 5 shows the robot's velocity profiles while executing a single square loop using Pure Pursuit Controller. The linear velocity, v_x, remains mostly steady during straight segments, while the rotational velocity, ω_z, shows sharp spikes during turns, corresponding to 90° rotations at each corner. These plots confirm that the robot's motion behavior is consistent with the planned square trajectory and reflect the controller's ability to manage both translation and rotation effectively.

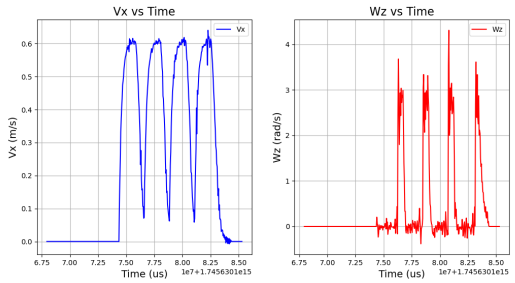


Fig. 5: Linear and rotational velocity during one 1m square loop.

B. Simultaneous Localization and Mapping (SLAM)

In mobile robotics, SLAM is a methodology used to construct a map of the environment, while simultaneously keeping track of the current location of the robot within the environment. In simple terms, SLAM helps the robot understand the following questions: "Where am I?" (Localization) and "What is around me?" (Mapping). Refer to Fig. 6 for an overview of the SLAM system.

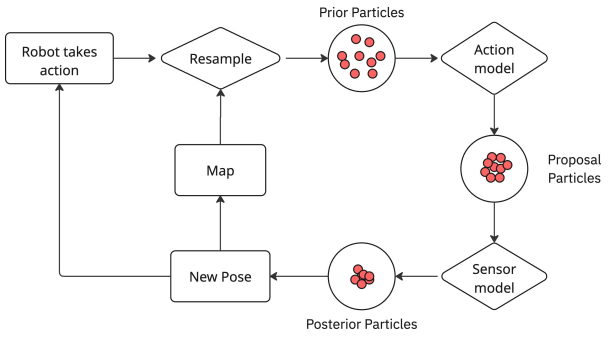


Fig. 6: SLAM System block diagram

1) Mapping: Mobile robots typically begin without knowledge of their location. Building a map of the environment enables them to localize and navigate effectively. In our system, we use LiDAR scans to construct an **occupancy grid map**, where the environment is discretized into fixed-size grid cells. Instead of binary values, each cell stores the **log-odds** of occupancy, defined as:

$$\text{logOdds}(m) = \log\left(\frac{\Pr(m)}{1-\Pr(m)}\right)$$

$\Pr(m)$ is the probability that a grid cell is occupied. The log-odds formulation converts this probability into an additive form for efficient Bayesian updates: positive values (when $\Pr(m) > 0.5$) indicate likely occupancy, while negative values ($\Pr(m) < 0.5$) suggest free space.

a) Updating the Grid Using LiDAR: When a LiDAR scan is received, the robot updates the occupancy grid by:

- Marking the **free space** along the laser beam
- Marking the **occupied cell** at the beam endpoint (i.e., obstacle)

To determine which grid cells a beam traverses, we implemented **Bresenham's Line Algorithm**, an efficient integer-only method that precisely identifies the set of discrete cells intersected by a straight line.

b) Comparison: Bresenham vs Divide-and-Step: Another approach to ray traversal is the *Divide-and-Step* method, which samples points along the ray at fixed intervals (e.g., every 0.5 grid length) and converts them into grid cells. While simpler, this method may produce duplicates or miss cells due to rounding. Table V compares the two.

Method	Description	Pros and Cons
Bresenham	Integer grid walk	+ Precise and efficient – More complex logic
Divide-and-Step	Ray sampling	+ Simple and intuitive – May skip or duplicate cells

TABLE V: Comparison of Ray Traversal Methods

c) Mapping Evaluation in Maze Driving: Fig. 7 illustrates the generated occupancy grid map during a maze driving scenario. The result demonstrates the effectiveness of our LiDAR-based mapping pipeline in capturing obstacle boundaries and free space in real-time.

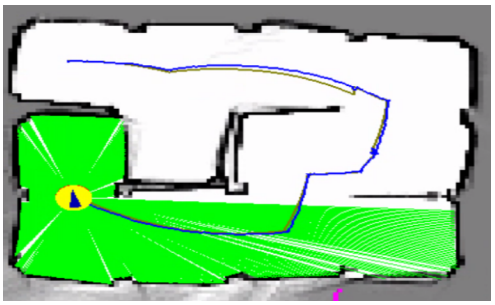


Fig. 7: Plot of the map from `drive_maze.log` file

2) Monte Carlo Localization: We use Monte Carlo Localization (MCL) to estimate the robot's pose relative to the map. A core aspect of the MCL makes use of the **Particle Filter**. Additionally, as part of MCL, we make use of the **Action Model** and a **Sensor Model** to estimate the pose of the robot.

a) Action Model: The action model predicts how the robot moves based on odometry, using a rotation–translation–rotation (RTR) sequence. We implement a probabilistic motion model with a Gaussian distribution for noise.

Let (x_k, y_k, θ_k) be the previous pose and $(x_{k+1}, y_{k+1}, \theta_{k+1})$ be the next pose from the odometry reading. Each particle is updated accordingly using sampled motion noise.

$$\begin{cases} \Delta x = x_{k+1} - x_k \\ \Delta y = y_{k+1} - y_k \\ \Delta\theta = \theta_{k+1} - \theta_k \end{cases}$$

Using this information, we can calculate the following for the initial rotation (δ_{rot_1}), translation (δ_{trans}), and final rotation (δ_{rot_2}):

$$\begin{cases} \delta_{rot_1} = \arctan(\Delta y, \Delta x) - \theta_k \\ \delta_{trans} = \sqrt{\Delta x^2 + \Delta y^2} \\ \delta_{rot_2} = \Delta\theta - \delta_{rot_1} \end{cases}$$

σ_n indicates the standard deviation:

$$\begin{cases} \sigma_{rot_1} = \sqrt{k_1 \cdot |\delta_{rot_1}|} \\ \sigma_{trans} = \sqrt{k_2 \cdot |\delta_{trans}|} \\ \sigma_{rot_2} = \sqrt{k_1 \cdot |\delta_{rot_2}|} \end{cases}$$

These equations are then used to calculate the noise which is estimated to be a Gaussian distribution and the equations are as follows

$$\begin{aligned} \hat{\delta}_{rot_1} &\sim \mathcal{N}(\delta_{rot_1}, \sigma^2_{rot_1}) \\ \hat{\delta}_{trans} &\sim \mathcal{N}(\delta_{trans}, \sigma^2_{trans}) \\ \hat{\delta}_{rot_2} &\sim \mathcal{N}(\delta_{rot_2}, \sigma^2_{rot_2}) \end{aligned}$$

The parameters k_1 and k_2 control the amount of noise added to each motion component. In our case, the default values worked well and provided reliable localization results, as listed in Table VI.

k_1	0.005
k_2	0.025

TABLE VI: Action Model Uncertainty Values

Using the above equations, we can update the particle motion as follows:

$$\begin{aligned} x_{k+1} &= x_k + \hat{\delta}_{trans} \cdot \cos(\hat{\delta}_{rot_1} + \theta_k) \\ y_{k+1} &= y_k + \hat{\delta}_{trans} \cdot \sin(\hat{\delta}_{rot_1} + \theta_k) \\ \theta_{k+1} &= \theta_k + \hat{\delta}_{rot_1} + \hat{\delta}_{rot_2} \end{aligned}$$

b) Sensor Model & Particle Filter: The sensor model helps to understand how well the proposed robot pose explains the actual sensor readings. In essence, given the map of the world in which the robot is navigating, a particle (representing a candidate pose), and measurements from the LiDAR scan, the sensor model outputs a likelihood score of "how likely is it that the robot was at a particular pose, given the current observation". Based on the actions taken by the robot and based on the readings from the sensor model, the Particle Filter estimates n number of particles (all the positions) that the robot could be present in.

To ensure real-time performance with the particle filter running at 10 Hz, the update time must remain below 0.1 seconds. We benchmarked the average update times for various particle counts, as summarized in Table VII and visualized in Fig. 8. Based on these results, we discovered that 4000 particles is the maximum usable count while still meeting the real-time requirement.

No. of Particles	100	500	1000	4000	5000	10000
Runtime (s)	0.0015	0.0076	0.0165	0.0947	0.1259	0.3656

TABLE VII: Average runtime of particle filter update for different particle counts.

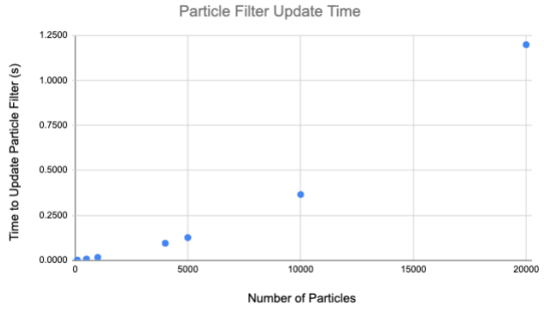


Fig. 8: Particle filter update time with increasing number of particles.

To evaluate pose uncertainty over time, we visualize particle filter outputs sampled at regular intervals, showing how the estimated poses align with the SLAM path. As illustrated in Fig. 9, each snapshot contains 300 particles, capturing the distribution of the robot's belief throughout its motion.

3) SLAM Evaluation: We evaluate the combined mapping and localization performance of our SLAM system, as illustrated in the block diagram shown in Fig. 6. The evaluation was conducted using the provided log file `drive_maze_full_rays.log`, where the robot was operated with SLAM enabled.

To assess accuracy, we interpolate the ground-truth poses and SLAM-estimated poses to align them at the same timestamps, then compute the root mean square

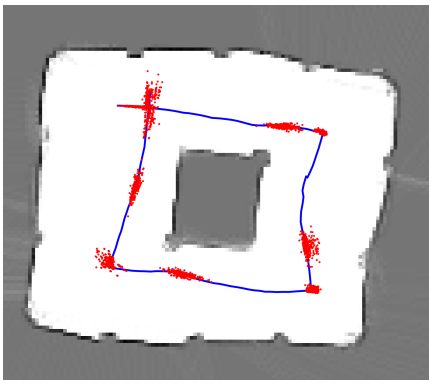


Fig. 9: Time-sampled particle visualization corresponding to the SLAM trajectory in `drive_square.log`.

(RMS) error between the two sets of poses recorded during the trajectory. The comparison is visualized in Fig. 10, and the quantitative results are summarized in Table VIII. The results indicate that the SLAM system achieves low estimation error.

	X (m)	Y (m)
RMS Error	0.0193	0.0364
Standard Deviation	0.0163	0.0346

TABLE VIII: RMS Error and Standard Deviation between SLAM poses and ground-truth poses.

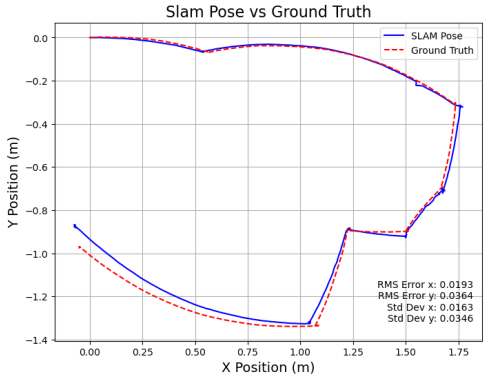


Fig. 10: Comparison of SLAM poses and ground-truth poses from `drive_maze_full_rays.log`.

C. Planning and Exploration

When the robot is provided with a map, a source, and a destination within the map, the robot makes use of a planning algorithm (e.g., A*) to create a path to go from the source to the destination. Exploration is the idea of expanding what is known about the map by planning and executing several paths to fully explore the robot's environment.

1) Obstacle Distance Grid: To support planning, we first compute an obstacle distance grid from the SLAM-generated occupancy map. This grid records the Euclidean distance from each free cell to the nearest obstacle, using an 8-connected Brushfire algorithm.

Occupied cells are initialized with zero distance, and distances are propagated outward using a priority queue. Cardinal neighbors add a cost of 1.0, while diagonal neighbors add $\sqrt{2}$. Cells are updated only if unvisited or if a shorter path is found.

2) A Path Planning:* With the obstacle distance grid available, we proceed to compute a safe and efficient path using the A* search algorithm. A* is a path-finding method that computes the shortest route between a given start node and a target node. It evaluates each node based on two components: the actual cost from the start node to the current node, referred to as `g_cost`, and a heuristic estimate of the remaining cost to the goal, known as `h_cost`. An overall score, `f_cost = g_cost + h_cost`, is computed for each node, and the algorithm prioritizes nodes with the lowest `f_cost` for exploration. This balance between actual cost and heuristic guidance enables A* to efficiently identify an optimal path.

a) `g_cost` calculation: The `g_cost` represents the actual cost accumulated from the start node to the current node. It is computed by summing the Euclidean distances between consecutive nodes, reflecting the true geometric travel cost.

To encourage safer navigation, a penalty is added when a node is close to obstacles. This penalty is based on the ratio between a predefined maximum safe distance and the node's distance to the nearest obstacle. If the distance lies between a minimum threshold (slightly above the robot's radius) and a maximum limit (10 times that radius), the penalty increases as the node nears an obstacle. This encourages the planner to avoid narrow or hazardous regions by inflating the cost of risky nodes.

$$g = g_{\text{parent}} + \sqrt{dx^2 + dy^2} + \frac{\texttt{max\_limit}}{\texttt{cell\_distance}}$$

where dx and dy are the x and y differences between the current node and its parent, `max_limit` is a predefined safe distance, and `cell_distance` is the distance to the nearest obstacle from the obstacle distance grid.

b) `h_cost` calculation: The heuristic cost is defined as the diagonal distance between the current node and the goal node, using the octile distance formula:

$$h = (dx + dy) + (\sqrt{2} - 2) \times \min(dx, dy)$$

where dx and dy are the absolute differences in the x and y coordinates between the current node and the

		Min	Mean	Max	Median	Std Dev
Successful	convex grid	333	1856	3379	0	1523
	empty grid	988	1031.33	1103	1103	51.05
	maze grid	4571	5955.75	7462	4571	1226.56
	narrow constriction grid	1161	1874.5	2588	0	713.5
	wide constriction grid	725	249374	745757	745757	350996
Failed	convex grid	31	31.5	32	0	0.5
	empty grid	16	16.5	17	0	0.5
	maze grid	14	63.6	218	30	77.48
	narrow constriction grid	23	2.11e7	6.34e7	26	2.98e7
	wide constriction grid	32	32	32	0	0

TABLE IX: Computation Times (in microseconds) for A* path planning for various grids

goal node. This formula calculates the minimum travel cost assuming movement in eight directions is allowed. It prioritizes diagonal moves wherever possible, and uses straight moves for the remaining distance.

c) A Planning Evaluation:* As shown in Fig. 11, the planner successfully generated a collision-free path from the goal position back to the starting point where mapping was originally initiated. The planned path fits well with the known free and occupied areas in the map, showing that the SLAM-generated map is reliable for tasks like navigation.

Furthermore, the robot was able to follow the planned path using its motion controller. The SLAM-estimated poses closely align with the A* path, indicating that the controller effectively tracked the trajectory and guided the robot to the desired location.

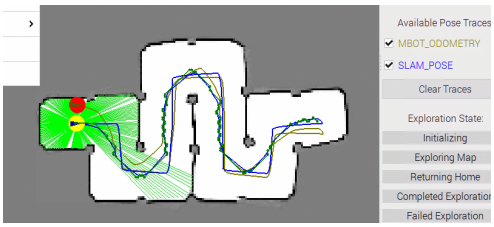


Fig. 11: A* Planned Path on the SLAM-Generated Maze Map
Green path: A* planned trajectory from right to left side of the maze.
Olive path: Odometry estimated from IMU fusion.
Blue path: SLAM-estimated pose.

d) A Efficiency and Performance Evaluation:* Refer to Table IX, which summarizes the times (in microseconds) taken by the A* algorithm to compute the shortest path. Our A* implementation is optimal and efficient in successful cases, with mean runtimes under 2ms on convex, empty, and narrow constriction grids. Maze grids are more complex, increasing computation time to around 6ms. The wide constriction grid,featuring a larger but still obstacle-dense bottleneck, shows the highest variance, with a mean of 249ms and a maximum of 745ms, due to the need to explore many alternative paths.

In failed cases, most environments complete quickly, with runtimes under 100us. The narrow constriction grid is an exception, where a single extreme case pushes the mean runtime beyond 21 seconds. This indicates that even simple-looking environments can cause A* to take significantly longer when tight spaces force exhaustive search before failure.

3) Map Exploration - Frontiers: We implement an autonomous exploration module for the MBot using a state machine approach. The robot explores an environment, identifies frontiers (boundaries between known and unknown regions), and plans paths to navigate through the environment efficiently. The primary objective is to complete exploration or return home in cases where all frontiers are explored or unreachable.

a) Initialization and Communication: The module initializes by subscribing to essential communication channels, including:

- `SLAM_MAP_CHANNEL`: Receives updates to the occupancy grid map.
- `SLAM_POSE_CHANNEL`: Receives updates to the MBot's pose.
- `MESSAGE_CONFIRMATION_CHANNEL`: Confirms receipt of messages, such as paths sent to the motion controller.

Upon receiving the first map and pose, the MBot's initial pose is stored as the home position. The planner is configured with the map and MBot's radius to facilitate motion planning.

b) Exploration Logic: The exploration process executes in a loop until the mbot either completes exploration or encounters failure. Data readiness is continually checked to ensure both the map and pose are available before proceeding. Once data is ready, the exploration state machine is invoked.

c) State Machine Execution: The exploration state machine transitions between five distinct states:

- **Initializing:** Sets up the system and transitions immediately to the exploration phase once the first bit of data is received.
- **Exploring the Map:** Identifies frontiers in the map and selects the best frontier to explore. The mbot

plans a path to the selected frontier and follows it until exploration is completed, fails, or transitions to another state.

- **Returning Home:** After completing exploration, the MBot navigates back to its initial home position using a planned path.
- **Exploration Completed:** The MBot remains in this state once the environment is fully explored and it has returned home.
- **Exploration Failed:** Triggered when no valid paths to frontiers or the home position are found.

d) Path Planning and Execution: During exploration, the MBot identifies frontiers using the occupancy grid map and selects an appropriate target frontier. Path planning is performed to navigate the robot towards the target while avoiding obstacles and unsafe regions. The path is validated, and updates are sent to the motion controller for execution. If the path changes or confirmation is not received, the path is resent.

e) Returning Home: When exploration is completed, the mbot navigates back to its home position. The robot calculates the distance to home:

$$d_{\text{home}} = \sqrt{(x_{\text{home}} - x_{\text{current}})^2 + (y_{\text{home}} - y_{\text{current}})^2}$$

and follows a planned path until it is within a defined threshold. If no valid path exists, the exploration process fails.

f) Status Updates: Throughout exploration, the module continuously publishes status updates via the `EXPLORATION_STATUS_CHANNEL`. These updates reflect the current state, such as initializing, exploring, returning home, or completed exploration, along with the progress status.

g) Completion and Failure Handling: The exploration terminates in one of two states:

- **Completed Exploration:** All frontiers are explored, and the MBot has returned home successfully.
- **Failed Exploration:** No valid paths to the frontiers or home position can be found.

In either case, the module sends appropriate final status messages and remains in its terminal state.

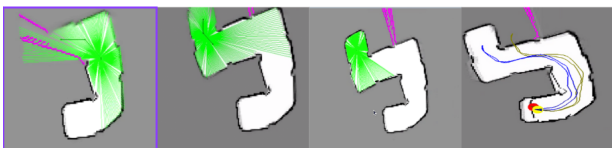


Fig. 12: The figure illustrates frontier-based autonomous exploration, followed by the robot returning to its home position.

4) Localization Strategy: The filter supports two initialization modes:

- **Pose-based Initialization:** When an initial pose is provided(e.g., via a localization guess or global stat), all particles are initialized uniformly around that pose with equal weight:

$$T_0 = F_{\text{init}} : \quad \forall i \in \{1, \ldots, N\}$$

- **Random Initialization:** When no prior pose is known, particles are distributed randomly across the valid free space in the occupancy grid. A short "burn-in" sequence of 5 LiDAR scans is used to refine the initial belief of updating particle weights and re-centering them through multiple sensor updates without motion.

III. Conclusion

Through BotLab, we successfully integrated low-level control and high-level autonomy to enable a mobile robot to navigate a maze environment. By leveraging odometry, wheel speed controllers, and a motion controller, we ensured accurate and responsive movement of the robot. These control systems formed the foundation for localization and path planning, allowing the robot to interpret and execute motion commands.

At the autonomy layer, we implemented a full SLAM pipeline that enabled the robot to build a map of its surroundings while localizing itself within it. With this map, we applied A* search to compute efficient paths to specific goals, and used frontier-based exploration to autonomously identify and explore unknown regions. Together, these modules demonstrated a complete perception-planning-control loop, equipping the robot with the ability to intelligently explore and navigate in real-time. This lab not only deepened our understanding of each individual module but also highlighted the importance of their seamless integration in autonomous robotics.

References

[1] D. Goodwin, "The evolution of autonomous mobile robots," September 2020, accessed: 2025-04-17. [Online]. Available: https://control.com/technical-articles/the-evolution-of-autonomous-mobile-robots/

[2] S. Dresser, "Amazon announces new fulfillment center robots, sequoia and digit," October 2023, accessed: 2025-04-17. [Online]. Available: https://www.aboutamazon.com/news/operations/amazon-introduces-new-robotics-solutions

[3] S. Nolan, "Waymo and cruise reimagine public autonomous transport," January 2025, accessed: 2025-04-17. [Online]. Available: https://evmagazine.com/self-drive/waymo-cruise-reimagine-public-autonomous-transport

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

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