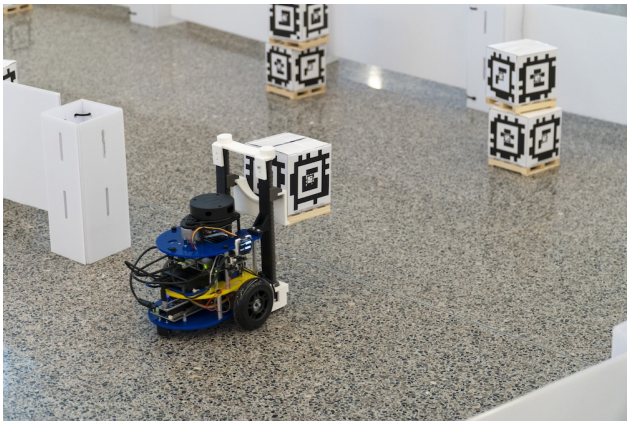
Autonomous Maze Navigation with SLAM and A-Star Path-Planning

Justin M. Lu, Changhe Chen, Nilay Roy Choudhury



***Abstract*—Autonomous navigation is a cornerstone of modern robotics, enabling robots to operate independently in dynamic environments. This study focuses on the fundamental aspects of autonomous navigation, including localization, mapping, path planning, and motion control, through the deployment of a wheeled robot tasked with exploring and escaping a structured environment. A real-time visual interface is utilized to illustrate the robot's evolving perception of its surroundings. Theoretical foundations are combined with practical implementation challenges to build robust low- and high-level controllers, process sensor feedback for precise localization, and integrate simultaneous localization and mapping (SLAM) techniques. The proposed framework demonstrates reliable planning and exploration in real-world conditions, showcasing the effectiveness of the developed algorithms in achieving autonomy.**

I. Introduction

AUTONOMOUS navigation is a critical capability for robotic systems across various applications, from industrial automation to exploration in inaccessible environments. This project focuses on designing and implementing a mobile robot capable of autonomous exploration, mapping, and navigation within structured environments such as mazes. By integrating key components—motion control, simultaneous localization and mapping (SLAM), and path planning—the robot adapts to its surroundings to perform complex tasks independently.

The robot uses LIDAR for environmental sensing, generating detailed 2D maps to localize itself and navigate through its surroundings. SLAM techniques enable real-time map construction and pose estimation, while algorithms like A* are employed for path planning to determine efficient routes to targets. A closed-loop control system, combining odometry and feedback from sensors, ensures precision in motion and responsiveness to environmental changes. This report provides a detailed account of the system's development, addressing the theoretical foundations, practical implementation challenges, and performance evaluation in navigating and escaping structured environments.

II. Methodology

A. Odometry

The MBot platform, available in a variety of configurations, consists of a differential-drive robot in its default state. As such, the state of the MBot can be represented as a position (using Cartesian coordinates (x, y)) as well as an angle θ depicting rotation. As such, we utilize the encoders and IMU that the MBot is equipped with in order to estimate the change in position, orientation, and velocity of the robot over time. This process is called **odometry**, and is utilized in order to ensure robustness in future tasks involving localization, navigation, and pathfinding.

1) Calibration: Calibration is a crucial step to account for hardware-specific variations in the MBot, such as differences in motor response and encoder behavior. It measures parameters like slopes and intercepts for both positive and negative motor directions, which correct for asymmetries in motor performance. These calibrated parameters ensure accurate velocity estimation and consistent odometry. The results of the calibration, including an evaluation of the variability in these parameters, are further analyzed in the Results section to assess their impact on the robot's overall performance.

2) Wheel Speed Calculation: To compute the rotational velocities of the robot's wheels in radians per second, the raw encoder tick data is converted using the following equation:

$$\text{velocity}[i] = \frac{\text{encoder\_polarity}[i] \cdot 10^6 \cdot 2\pi}{\text{GEAR\_RATIO} \cdot \text{ENCODER\_RES}} \cdot \frac{\Delta\text{ticks}[i]}{\Delta\text{time}}$$

Here, encoder_polarity$[i]$ accounts for the direction of rotation, GEAR_RATIO and ENCODER_RES account for the gear reduction and encoder resolution, and $\frac{\Delta\text{ticks}[i]}{\Delta\text{time}}$ represents the rate of encoder ticks over time. Using this formula, the left (v_L) and right (v_R) wheel velocities are determined.

These wheel velocities are then used to calculate the robot's differential body velocity in terms of forward velocity (v_x), lateral velocity (v_y), and angular velocity (w_z):

$$v_x = R \cdot \frac{v_L - v_R}{2}, \quad v_y = 0, \quad w_z = R \cdot \frac{-v_L - v_R}{2B}$$

where R is the wheel radius and B is the robot's base radius. This computation provides the robot's translational and rotational velocities, enabling differential drive control.

3) Gyrodometry: Gyrodometry enhances the robot's pose estimation by combining traditional wheel encoder-based odometry with gyroscope data for angular velocity correction [1]. The angular velocity (w_z) is calculated using both odometry ($wz\_odometry$) and the gyroscope (wz_gyro), and the difference (Δw_z) between these values is compared to a threshold. If the difference exceeds the threshold, the gyroscope reading is used to update w_z; otherwise, the odometry value is retained. Additionally, gyrodometry integrates changes in the gyroscope's angular position ($\Delta\theta_{\text{gyro}}$) to refine the robot's heading (θ), correcting for drift in odometry-based estimates. This hybrid approach significantly reduces errors caused by wheel slip and improves motion tracking accuracy.

B. Motion Control

1) Controller Description and Parameter Table: The final controller was designed using a PID control loop to accurately track the setpoint velocities for both wheels and angular velocity (w_z). The parameters were tuned to balance stability and responsiveness, accounting for motor-specific differences and ensuring smooth operation during motion control. The proportional (K_p), integral (K_i), and derivative (K_d) gains were configured individually for the left and right wheels, as well as for angular velocity control. The parameter values obtained during tuning are further discussed in the Results section.

2) Tuning Process and Metrics: The tuning process began with adjusting the proportional gain (K_p) for the left and right wheels to ensure initial stability and responsiveness to the setpoint velocities. Oscillations were introduced deliberately to gauge the system's response, after which the derivative gain (K_d) was set to reduce these oscillations. Since the integral gain (K_i) was left at 0.0, steady-state error correction relied solely on the proportional and derivative terms.

For angular velocity control, a higher K_p value of 0.75 was selected to achieve precise rotational control, while K_d was set to 0.0 due to the absence of significant oscillatory behavior in angular velocity tracking. Metrics such as error minimization and system stability within a permissible range were used to evaluate the tuning process.

3) Additional Features and Observations: Although the design intended to incorporate low-pass filters to reduce noise in encoder readings and PID outputs, these were not implemented in the final setup [2]. As a result, the controller exhibited performance degradation at higher speeds, such as increased noise and tracking errors. These issues are discussed in detail in the Results section.

The absence of filters particularly affected the derivative term, amplifying noise at higher velocities and reducing the system's robustness. This limitation highlights the importance of including noise filtering in future iterations to improve performance and extend the operational range of the controller.

4) Motion Control Between Waypoints: The motion control algorithm is designed to guide the MBot through a series of predefined waypoints by sending velocity and path commands via the Lightweight Communications and Marshaling (LCM) framework. The algorithm initializes by resetting the robot's odometry to $(0, 0, 0)$, ensuring a consistent starting reference frame. Velocity limits are set to cap the linear velocity ($v_x = 0.2$ m/s) and angular velocity ($w_z = \pi/4$ rad/s), preventing excessive motion speeds.

A global path is constructed from a list of waypoints, each defined by (x, y, θ) coordinates in the global frame. These waypoints include straight-line movements and orientation changes to create precise trajectories. The path is converted into a sequence of `pose2D_t` messages, which represent each waypoint. The completed path is published as a `path2D_t` message to the controller via the `CONTROLLER_PATH` channel, ensuring the robot can sequentially follow the trajectory. This approach leverages modular commands to effectively manage both position and orientation adjustments at each waypoint.

C. Simultaneous Localization and Mapping (SLAM)

Our SLAM system, consisting of an action model, sensor model, particle filter, and an occupancy grid, is implemented as a particle-filter based framework for real-time SLAM [3]. Motion updates from the action model and sensor data updates via the sensor model are combined in a probabilistic manner to build not only an occupancy grid map of the robot's environment, but also to estimate the robot's pose as it moves throughout said environment [4].

1) Action Model: The action model is essentially responsible for predicting the robot's pose based on odometry data, primarily through statistical (probabilistic) methods. The robot's motion is decomposed into a rotation-translation-rotation (RTR) model (since it is differential drive), and Gaussian noise is introduced to account for the inherent uncertainties and inaccuracies in the robot. Furthermore, direction handling and thresholding is introduced to prevent unnecessary movements. This Gaussian noise is parameterized for each of the three actions, as well as tuning constants k_1 and k_2 as follows:

TABLE I: Action Model Uncertainty Parameters

Action	Relevant Gain $(k1, k2)$	Stdev. of Noise
Rotation 1	k_1	$k_1 * \|\alpha\|$
Translation	k_2	$k_2 * \|ds\|$
Rotation 2	k_1	$k_1 * \|d\theta - \alpha\|$

where α, ds, and $|d\theta - \alpha|$ represent the magnitude of the first rotation, translation, and secondary aligning rotation respectively. As such, we note that tuning constant k_1 is solely responsible for tuning the variance associated with turning, while tuning constant k_2 is utilized only for tuning the variance (or uncertainty) associated with straight-line movement.

2) Sensor Model: The sensor model evaluates how well each particle aligns with the robot's environment, and processes laser scan data from the MBot's LiDAR to compute the likelihood of a particle's 'correctness'. Each ray is then scored based on the proximity of the ray endpoint to occupied cells in the map (via Breadth-First-Search).

3) Particle Filter: The particle filter essentially distributes a series of particles representing the robot's belief of its pose. Low-variance resampling is used to ensure higher-weight particles are retained, as a particles 'weight' is its log-likelihood of being the true robot pose. These particles are sampled from the aforementioned action model to predict the next likely state.

4) Combined Implementation: Our combined implementation combines the action model, sensor model, and particle filter into a SLAM pipeline that operates in real-time. At each time step, the action model predicts the motion of the robot by sampling from the RTR model (with Gaussian noise), generating a new set of predicted poses. These poses are then evaluated by the sensor model which assigns likelihood weights.

The particle filter then normalizes these weights and applies low-variance resampling to focus computational resources on the most likely poses. This process continuously updates the robot's estimated pose and the environment map, enabling robust adaptation to dynamic changes while improving accuracy over time [5].

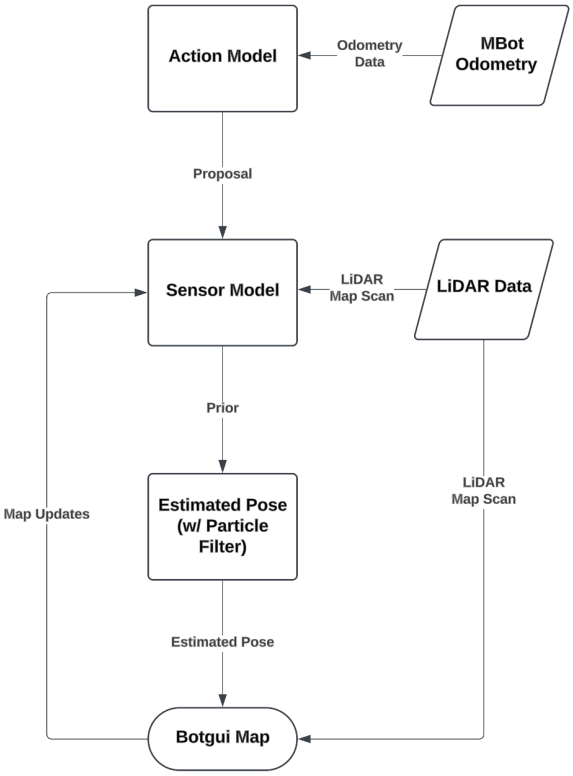


Fig. 1: SLAM System Block Diagram

Figure 1 illustrates our SLAM system pipeline using a block diagram. The system integrates LiDAR data and odometry from the MBot with probabilistic methods to account for uncertainties in sensor measurements. Leveraging these probabilistic approaches in our implementation ensures robust performance - minimizing overreliance on potentially noisy sensor data while maintaining accurate localization and mapping. This balance between sensor fusion and probabilistic modeling is central to the effectiveness of our SLAM implementation.

D. Path Planning

The A* search algorithm is implemented to find a path from a start position to a goal position in a grid-based environment. It combines heuristic and cost-based strategies to determine the optimal path while avoiding obstacles. Below is a detailed explanation of the workflow, including relevant formulas.

1) Initialization: The algorithm begins by converting the start and goal positions from global coordinates to grid cell coordinates using:

$$\text{cell}_x = \left\lfloor \frac{\text{global}_x}{\text{cell width}} \right\rfloor, \tag{1}$$

$$\text{cell}_y = \left\lfloor \frac{\text{global}_y}{\text{cell height}} \right\rfloor. \tag{2}$$

A priority queue (`open`) is initialized to track nodes for exploration, with the start node pushed into it. Additional queues (`closed`, `visited`) track explored nodes to avoid revisiting.

2) Node Expansion and Cost Calculation: The A* search algorithm involves several key steps to find an optimal path from the start to the goal node. Below is a detailed breakdown:

First, the algorithm checks if the current node matches the goal node. If they match, the path is found, and the algorithm terminates successfully.

Next, the algorithm explores 8 neighbor nodes (in cardinal and diagonal directions). Each neighbor is validated based on two criteria: grid boundaries and obstacle distance. The node must lie within the grid, and its distance from obstacles must satisfy:

$$d_{\text{obstacle}} = \text{distances}(x, y),$$

where nodes with $d_{\text{obstacle}} < \text{minDistanceToObstacle}$ are discarded.

The cost calculations follow, where the g-cost (the cumulative cost from the start node to the current node) is determined using:

$$g(n) = g(n_{\text{parent}}) + \text{moveCost},$$

with:

$$\text{moveCost} = \begin{cases} 1.414 & \text{(Diagonal move)} \\ 1 & \text{(Straight move).} \end{cases}$$

Nodes too close to obstacles, specifically those with $d_{\text{obstacle}} < \text{minDistanceToObstacle} \times 0.8$, are assigned:

$$g(n) = \infty.$$

Additionally, a penalty for proximity to obstacles is computed as:

$$g(n) += \text{distanceCost}, \quad \text{where}$$

$$\text{distanceCost} = (\text{maxDistanceWithCost} - d_{\text{obstacle}})^{\text{CostExponent}}$$

The heuristic cost, h-cost, is calculated as:

$$h(n) = \text{ManhattanDis}(n, \text{goal}) + (1.414 - 2) \cdot \min(\Delta x, \Delta y),$$

where:

$$\text{ManhattanDistance} = |\Delta x| + |\Delta y|.$$

Finally, nodes are added to the `open` and `visited` queues with their cumulative cost:

$$f(n) = g(n) + h(n).$$

If a better path to an already visited node is discovered, the node's cost and parent pointer are updated accordingly.

3) Path Extraction and path pruning: Once the goal node is reached, the algorithm proceeds to reconstruct the path leading from the start node to the goal node. Beginning at the goal node, the function iteratively follows each node's parent pointer until the start node is encountered, effectively tracing the path in reverse order.

After the traversal is complete, the collected sequence of nodes represents the path from the goal to the start. To provide a final path that progresses naturally from the start to the goal, the sequence is reversed. This ensures the resulting path is in the correct order for subsequent use. By leveraging this parent-pointer mechanism, the algorithm efficiently reconstructs the optimal path found during the search process.

The extracted path is converted to global coordinates and orientations using:

$$\theta = \text{atan2}(\Delta y, \Delta x). \tag{3}$$

The result is a sequence of 2D poses, including x, y, and θ.

E. Exploration

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.

1) 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.

2) 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.

3) 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:** Transitioned to when no valid paths to remaining frontiers or the home position can be planned.

4) 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.

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

$$d_{\text{home}} = \sqrt{(x_{\text{home}} - x_{\text{current}})^2 + (y_{\text{home}} - y_{\text{current}})^2}, \quad (4)$$

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

6) 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.

7) 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.

III. Results

A. Odometry

TABLE II: Table of Calibration Parameters

Parameter	Mean	Standard Deviation
Positive Slope Right	0.05237	0.00065
Positive Intercept Right	0.05178	0.00643
Negative Slope Right	0.05910	0.00164
Negative Intercept Right	-0.04229	0.00349
Positive Slope Left	0.05937	0.00052
Positive Intercept Left	0.04823	0.00451
Negative Slope Left	0.05435	0.00052
Negative Intercept Left	-0.05865	0.00396

1) Calibration: The calibration process yielded key parameters for both positive and negative motor directions, as summarized in Table II. These parameters, including slopes and intercepts, correct for asymmetries in motor performance and ensure accurate velocity estimation. The low standard deviations observed across all parameters indicate consistency in the calibration process and minimal variability in motor response.

Positive and negative slopes represent the relationship between input commands and motor velocities, while intercepts account for inherent offsets in motor behavior. For example, the slightly higher negative slope values compared to positive slopes suggest a marginal difference in motor response during reverse motion. These calibrated parameters directly enhance odometry accuracy, enabling the MBot to track its trajectory more reliably. The observed results demonstrate that the calibration effectively minimizes discrepancies, laying the foundation for precise control during navigation tasks.

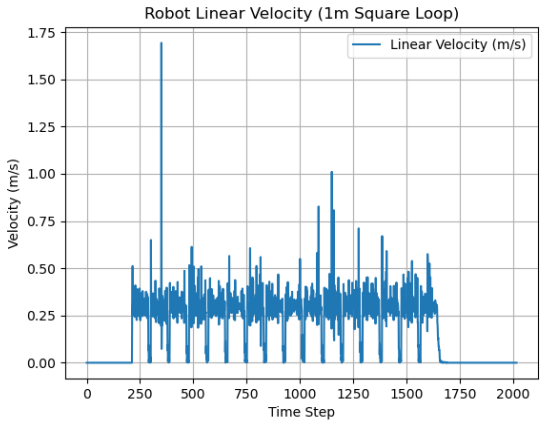


Fig. 2: Robot Linear Velocity for a 1m Square Path

2) Wheel Speed Calculation: Figure 2 shows the robot's linear velocity while executing a 1-meter square loop. The plot highlights fluctuations in velocity, with noticeable spikes occurring during turns. These spikes

indicate that the PID controller for angular velocity (w_z) was not optimally tuned, causing overshoot and instability during directional changes.

Additionally, the absence of filtering in the velocity computation exacerbated the noise, as seen in the inconsistent variations throughout the path. Encoder readings and motor response irregularities contributed to the noisy measurements, particularly at higher speeds. Implementing low-pass filters in future iterations could smooth out the velocity profile and improve overall control performance.

The results underscore the need for refined PID tuning for angular velocity control and filtering mechanisms to achieve smoother and more reliable motion.

B. Motion Control

TABLE III: PID Parameters for the Final Controller

Controller	K_p	K_i	K_d
Left Wheel	0.26	0.0	0.025
Right Wheel	0.26	0.0	0.025
Angular Velocity (w_z)	0.75	0.0	0.0

1) Controller Parameters and Analysis: The parameters obtained from tuning the PID controller are presented in Table III. The left and right wheel controllers showed similar gains, reflecting balanced motor behavior, while the angular velocity controller had a higher proportional gain to handle rotational dynamics. These parameters were critical for reducing tracking error and improving stability during operation.

These parameters were evaluated based on the robot's ability to track velocities accurately and maintain stability under various motion conditions. The results demonstrate that the tuned PID controller effectively minimized steady-state error and oscillations, ensuring reliable performance during navigation tasks.

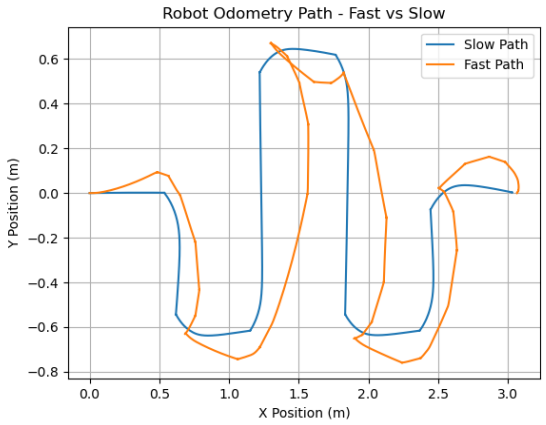


Fig. 3: Odometry Path Comparison at Low and High Speed

2) Motion Control Path Comparison: Figure 3 shows a comparison of the robot's odometry path at low and high speeds, highlighting the performance of the motion control algorithm. The "Slow Path" corresponds to a lower velocity setting ($v_x = 0.2\,\text{m/s}$), while the "Fast Path" was executed at higher velocities.

At low speeds, the MBot follows the intended trajectory more accurately, as seen in the smooth and consistent path in blue. The lower velocity minimizes wheel slip, encoder noise, and deviations caused by motor response delays, resulting in better tracking of waypoints.

In contrast, the "Fast Path," shown in orange, demonstrates greater deviations from the desired trajectory. At higher speeds, the robot experiences increased inertia, reduced control accuracy, and pronounced overshoot during turns. These effects cause noticeable path distortion, particularly around sharp corners and direction changes, where the robot's angular control (w_z) struggles to keep up with the setpoints.

Overall, the results confirm that slower velocities improve waypoint tracking accuracy, while higher velocities introduce errors due to mechanical and control limitations. This analysis emphasizes the need for velocity-dependent tuning or advanced control strategies to enhance performance under varying speed conditions.

C. Simultaneous Localization and Mapping (SLAM)

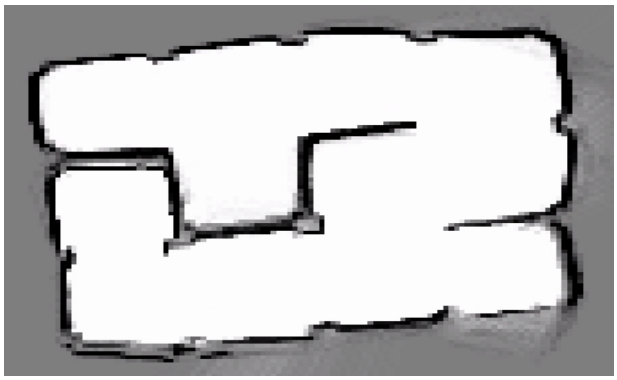


Fig. 4: Map from log file 'drive_maze'

Figure 4 depicts the map from our log file 'drive_maze.log', and will be utilized primarily to evaluate the performance of our action model, sensor model, particle filter, and combined implementation. This map was selected due to its relative complexity in comparison to the more basic log file maps available.

In our empirical testing of the RTR action model, we found that the values depicted in Table IV most accurately reflected the true uncertainty in MBot movement. These values resulted in the most realistic particle

TABLE IV: Action Model Tuning Constants

Gain (k_1, k_2)	Value
k_1	0.05
k_2	0.025

distributions emanating from the true robot pose, without understating the true noise and movement error present in real-world testing.

To illustrate how processor performance impacts the upper bound on the number of generated pose estimations (i.e the number of particles), we constructed an experiment measuring update time as more particles were introduced. Our results were as follows:

TABLE V: Avg. Particle Filter Update Times

Number of Particles	Update Time (μs)
100	32140
500	87934
1000	165716

Table V shows the update rate in μs associated with the particle filter for 100, 500, and 1000 particles respectively. As a broad estimate, the maximum number of particles our implementation of the particle filter could feasibly handle (sustainably, during nominal MBot operation) would be approximately 500 particles. These quantities of particles represent the realistic upper bound on particle generation while taking into account the response time of the onboard Raspberry Pi 5.

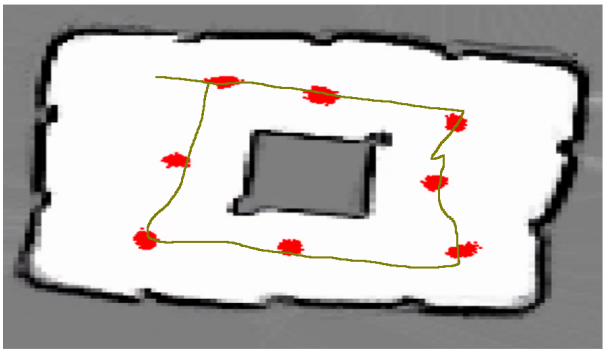


Fig. 5: Particle Plot Distribution

Figure 5 illustrates our particle filter in action, plotting 300 particles at regular intervals along the square path taken by the MBot during a test run. We conclude that our chosen constants k_1 and k_2 accurately depict the uncertainty of the MBot during its path, and accurately represent real-world error conditions for the MBot differential-drive platform.

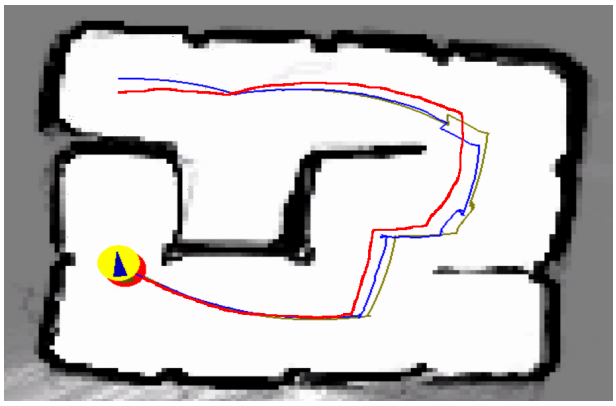


Fig. 6: Estimated vs. Ground-Truth Comparison

Figure 6 compares the SLAM pose generated by our implementation with the ground-truth pose. The paths align fairly well, obtaining an Root Mean Squared Error (RMSE) of approximately 0.114, indicating that our system performs admirably given its hardware constraints. Our results are further reinforced by the fact that the MBot, in real-world performance, consistently ends up at a distance less than 4cm from its target pose for any path with more than four vertices. Our empirical results further reinforce why the probabilistic methods employed in *Probabilistic Robotics* proved so influential and successful.

D. Planning and Exploration

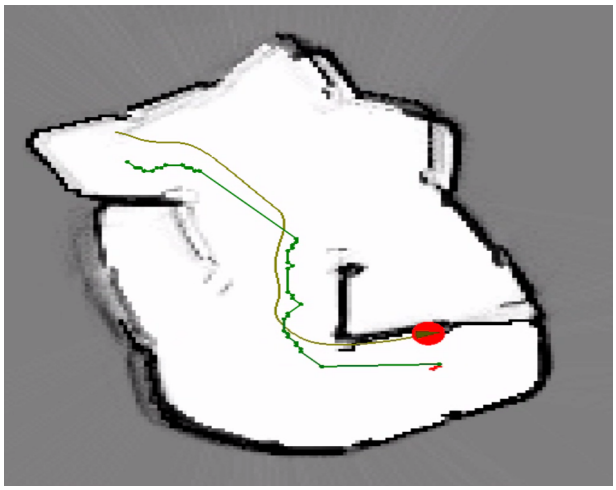


Fig. 7: Planned vs. Actual Path in Custom Environment

Figure 7 depicts the planned path (dotted green) overlayed with with the actual driven path (olive green) in a custom environment.

Runtimes for the A-Star algorithm are depicted in Table VI. We can observe significant variation across different test configurations. For sparse environments

TABLE VI: Successful Cases Timing (μs)

Cases (μs)	Min	Mean	Max	Median	Stdev
convex_grid	3371	3953	10700	3471	1271
empty_grid	8127	14747	60657	13456	8785
filled_grid	NA	NA	NA	NA	NA
maze_grid	3508	4193	10470	3626	1304
narrow_cons_grid	8146	9640	19411	9447	1861
wide_cons_grid	8074	9401	20861	9060	1958

like the Empty Grid, execution times showed high variability due to increased memory usage and expanding open nodes. The narrow grid displayed a wide spread and an extremely large maximum time, highlighting the difficulty in our algorithm for navigating constrained spaces. The wide grid exhibited consistent performance overall, while the convex and maze grid were more stable and had acceptable runtimes. We can conclude that overall, A-Star performance is highly dependent on grid complexity, with sparse grids increasing computation times due to memory overhead while blocked or simple grids enable faster results.

IV. Discussion

The results from the calibration, odometry, and motion control experiments demonstrate the effectiveness of our implemented methods. The calibration parameters, shown in Table II, exhibit low variability, ensuring consistent motor performance. The differences in slopes and intercepts between positive and negative motor directions indicate slight asymmetries in motor behavior that are effectively compensated for through calibration. The calibration step significantly enhances odometry accuracy, forming the basis for reliable navigation.

Analysis of robot velocity in Figure 2 shows spikes during turns, which can be attributed to PID tuning for angular velocity (w_z). These 'spikes' introduce instability and overshoot, particularly in sharp turns, where the robot struggles to maintain a smooth trajectory. Additionally, the lack of filtering amplifies noise in the encoder readings, resulting in a fluctuating velocity profile even on straight segments. Implementing low-pass filtering and refining PID parameters, particularly for angular control, would reduce these inconsistencies and improve overall stability.

Figure 3 shows the impact of speed on path tracking accuracy. At lower speeds, the MBot adheres closely to the intended trajectory, benefiting from reduced wheel slip and better motor response. However, higher velocities exacerbate deviations due to increased inertia and control limitations, leading to path distortion. These observations suggest the need for speed-dependent control strategies or dynamic parameter tuning to optimize performance across a range of speeds.

Finally, the SLAM tests and particle filter experiments demonstrated robust results in mapping and robot pose estimation, as shown in Figures 4 and 5. The chosen tuning constants k_1 and k_2 accurately reflect real-world uncertainty, producing realistic particle distributions. However, the update time analysis (Table V) highlights computational constraints that limit the maximum number of particles to approximately 500 for real-time operation. Future optimizations, such as parallel processing or improved sampling methods, could further enhance SLAM performance without sacrificing responsiveness.

In conclusion, while the current implementation provides reliable navigation and mapping capabilities, further refinements in PID tuning, noise filtering, and computational efficiency are necessary to improve performance under more challenging conditions.

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] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*. MIT Press, 2011.

[3] J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

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

[5] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artificial Intelligence*, vol. 128, no. 1-2, pp. 99–141, 2001.

[6] J. Borenstein and L. Feng, "Umbmark: A benchmark test for measuring odometry errors in mobile robots," in *Mobile Robots X*, vol. 2591. SPIE, 1995, pp. 113–124.

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