Team 5 ROB 550 BotLab Report

Hao-Yu Chan, Chenen Jin, Benard Adewole
{hychan, jchenen, adbenard}@umich.edu

***Abstract*—The MBot Classic is a differential-drive robot designed for autonomous navigation and object manipulation. We integrated a PID controller, gyrodometry-enhanced odometry, Pure Pursuit motion control, and a particle filter-based SLAM system. A* path planning and frontier exploration enabled efficient mapping of unknown environments. Our system achieved a SLAM RMS error of 3.25 cm and supported real-time localization with around 6000 particles, successfully completing two industry-inspired challenges.**

I. Introduction

Nowadays, autonomous ground vehicles play a part in our everyday lives. Our project seeks to demonstrate a lot of the same capabilities that many modern autonomous systems need to operate successfully. MBot Classic is a three-level differential-drive ground robot designed for autonomous navigation and interaction in structured environments. At the base level, the robot is equipped with DC motors that drive a two-wheeled system, complemented by wheel encoders and an inertial measurement unit (IMU) to track motion and orientation. The middle level houses the power system (battery), a Raspberry Pi 5 embedded processor for onboard computation, a Pico microcontroller for the lower level control, and a forward-facing camera to detect and identify objects marked with AprilTags. Mounted on the top level is a LIDAR scanner, which provides high-resolution range data for navigation, mapping, and localization.

In this project, we integrate the M-Bot hardware with a robust software library and a custom-built forklift mechanism to complete four core challenges. These include: The implementation of a hierarchical motion control system, a simultaneous localization and mapping (SLAM) framework, a path planning algorithm for safe and efficient exploration, and the design of a forklift mechanism to interact with and manipulate objects in the environment. These components demonstrate a comprehensive mobile robotic system capable of perception, decision-making, and physical interaction in a dynamic environment.

II. Methodology

A. Wheel speed controller, odometry, and motion controller

1) Wheel Speed Calibration: Before each deployment of the MBot on a new surface, the wheel motors must be recalibrated. A calibration program is used to measure the robot's motion to determine the polarity of the encoders and motors, and determine the relationship between the PWM cycles and the actual speed of the wheels by sending PWM commands to the wheels. On a concrete floor, such as the one found in the lab, calibration was performed using this tool. Multiple trials were run across a range of PWM values for both left and right wheels.

2) Odometry: Odometry with information from the encoders of the wheels only brought about large errors, and the errors would propagate due to both systematic and non-systematic factors. Gyrodometry[1], which involves fusing odometry with gyro-based heading estimate, improved the accuracy of MBot rotation angle ω. The original odometry equations without gyro information are given below:

$$\hat{v}_x = \frac{(\hat{w}_L - \hat{w}_R)R}{2}$$
$$\hat{w}_z = \frac{(-\hat{w}_L - \hat{w}_R)R}{2b}$$

Where $\hat{v}_x$ represents forward velocity, $\hat{w}_R$ and $\hat{w}_L$ represent the wheel speed, $\hat{w}_L$ is the angular velocity,R is the wheel radius, and b is the base radius of the MBot.

The implementation of the gyrodometry is presented in Algorithm 1. We could choose the threshold value $\Delta\theta_{\text{thres}}$ based on the actual performance. Here, we selected 0.125 as the initial value and tuned it to improve the accuracy further.

Algorithm 1 Gyrodometry

```
1: Initialize $\theta_0$                                   ▷ Initial orientation
2: for each time step $i$ do
3:     Compute $\Delta_{G-O,i} = \Delta\theta_{\text{gyro},i} - \Delta\theta_{\text{odo},i}$   ▷ Gyro-odometry difference
4:     if $|\Delta_{G-O,i}| > \Delta\theta_{\text{thres}}$ then
5:         $\theta_i = \theta_{i-1} + \Delta\theta_{\text{gyro},i} \cdot T$   ▷ Update using gyro
6:     else
7:         $\theta_i = \theta_{i-1} + \Delta\theta_{\text{odo},i} \cdot T$   ▷ Update using odometry
```

Algorithm 2 Feedback & Feedforward Control

```
1:  **Input:** Robot command $v$ (linear velocity), $\omega$ (angular velocity)
2:  **Parameters:** Wheel radius $r_w$, wheelbase $b$
3:  **Initialize:** $\hat{\omega} \leftarrow 0$ (estimated angular velocity)
4:  **while** control loop running **do**
5:                                ▷ Kinematics: Convert to wheel velocities
6:      $v_R \leftarrow v + \frac{b}{2}\omega$          ▷ Right wheel linear velocity
7:      $v_L \leftarrow v - \frac{b}{2}\omega$          ▷ Left wheel linear velocity
8:      $\omega_R \leftarrow \frac{v_R}{r_w}$           ▷ Right wheel angular velocity
9:      $\omega_L \leftarrow \frac{v_L}{r_w}$           ▷ Left wheel angular velocity
10:                           ▷ Feedback: Measure actual wheel velocities
11:     $\omega_{R,\text{enc}} \leftarrow$ ReadEncoderRight()   ▷ Right encoder
12:     $\omega_{L,\text{enc}} \leftarrow$ ReadEncoderLeft()    ▷ Left encoder
13:                                   ▷ PID control for wheel velocities
14:     $u_{R,\text{PWM}} \leftarrow \text{PID}(\omega_R, \omega_{R,\text{enc}})$   ▷ Right wheel
15:     $u_{L,\text{PWM}} \leftarrow \text{PID}(\omega_L, \omega_{L,\text{enc}})$   ▷ Left wheel
16:                                   ▷ Apply control signals to motors
17:     SetMotorRight($u_{R,\text{PWM}}$)
18:     SetMotorLeft($u_{L,\text{PWM}}$)
19:                            ▷ IMU Fusion: Estimate angular velocity
20:     $\omega_{\text{IMU}} \leftarrow$ ReadIMU()      ▷ IMU angular velocity
21:     $\omega_{\text{odo}} \leftarrow \frac{r_w}{b}(\omega_{R,\text{enc}} - \omega_{L,\text{enc}})$   ▷ Odometry-based
22:     $\hat{\omega} \leftarrow$ ComplementaryFilter($\omega_{\text{IMU}}, \omega_{\text{odo}}$) ▷ Fusion
23:     $\omega \leftarrow \omega + K(\hat{\omega}_{\text{desired}} - \hat{\omega})$   ▷ Correction
```

Algorithm 3 Pure Pursuit

```
1: $\Delta$x = $x_{target} - x_{pose}$
2: $\Delta$y = $y_{target} - y_{pose}$
3: $dist = \sqrt{\Delta x^2 + \Delta y^2}$
4: $\alpha = angle\_diff(\arctan 2(\Delta y, \Delta x), \theta_{pose})$
5: $L = 0.4$                          ▷ Lookahead Distance
6: $fwd\_vel = pid\_vel * dist$
7: $turn\_vel = pid\_turn * 2 * \Delta y / L^2$
```

3) PID Controller: While it is technically feasible to operate the Mbot without a PID motor speed controller, doing so often leads to erratic motion due to the direct use of unfiltered PWM signals, making it difficult for the Mbot to track the set speed precisely and sometimes tip over. By incorporating a Proportional-Integral-Derivative (PID) controller and effective filters, the robot achieves smoother and more precise speed control. Our implementation of the PID controller leverages calibration data and feedback from the wheel encoders to regulate wheel speed effectively.

The motor speed from the encoders is filtered with a low-pass filter to reduce discretization noise and smooth out sharp initial changes. The filtered speed is compared to the desired velocity, and the resulting error is processed by an RC filter to generate control commands for the left and right motors. These commands are then converted into PWM signals. The control algorithm and its tuned parameters are detailed in Table I and Algorithm 2.

Parameters	Wheel Velocity	Angular Velocity
P	0.1	0.5
I	0.01	0.0
D	1×10^{-8}	1×10^{-3}
time constant τ	N/A	0.01

TABLE I: PID Parameters and Low-Pass Filter

4) Motion Controller: While the original `diff_motion_controller` was adequate for basic path following, we implemented the **Pure Pursuit algorithm** to achieve more precise control. The implementation is detailed in Algorithm 3. We selected a look-ahead distance $L = 0.4$, resulting in a more aggressive yet accurate tracking of waypoints. The PID gains were set to 1 for forward velocity and 2 for turn velocity.

B. Simultaneous Localization and Mapping (SLAM)

Simultaneous localization and mapping (SLAM) refers to the process of constructing or updating a map of an unknown environment while simultaneously tracking the location of the agent [2]. In this checkpoint, SLAM is divided into two distinct components: **Mapping** and **Localization**. Figure 1 provides an overview of how the SLAM algorithm works, and how each components interact with each other.

1) Mapping - Occupancy Grid: In the occupancy grid map, we utilized LiDAR laser scan data to determine whether a given space is occupied. The laser range was limited to $5.5f$. At the endpoint of each laser scan, we assume the presence of an obstacle and increase the occupancy odds of the corresponding cell. A cell is considered occupied if its odds value is greater than 0; otherwise, it is treated as free space. The odds values range between -127 and 127.

To determine which cells the laser beams pass through, we implemented both **Bresenham's algorithm** and the **divide-and-step-along-ray method**. After comparison in Figure 2, we chose **Bresenham's algorithm** due to its higher accuracy in identifying the traversed cells.

Additionally, the LiDAR on the mbot does not scan fast enough to synchronize with the encoder readings from the motors. As a result, the laser scan data becomes misaligned with the robot's actual motion, causing the scans to shift and leading to poor mapping quality. To address this issue, the laser scans can be interpolated to align them more accurately with the encoder readings.

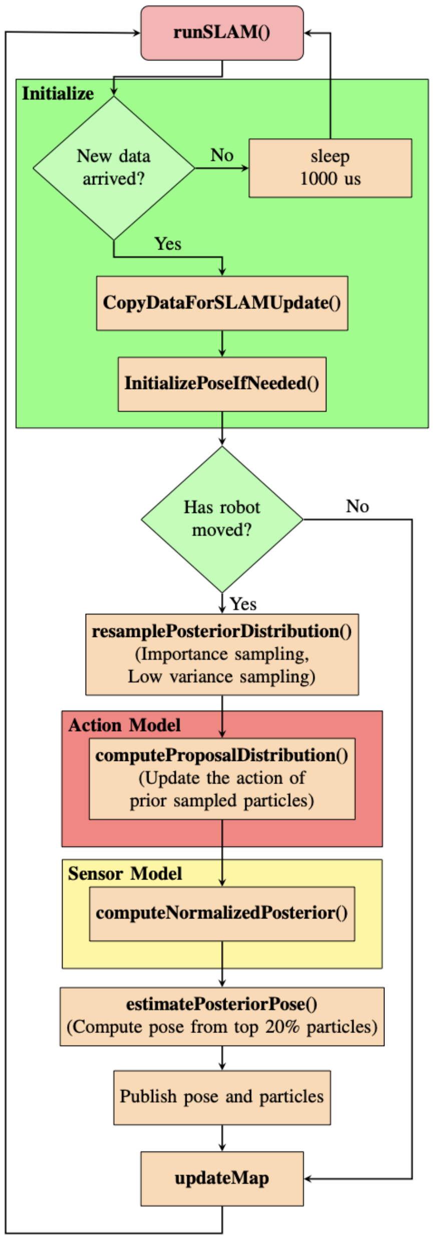

runSLAM()
Initialize
New data arrived?
No
sleep 1000 us
Yes
CopyDataForSLAMUpdate()
InitializePoseIfNeeded()
Has robot moved?
No
Yes
resamplePosteriorDistribution() (Importance sampling, Low variance sampling)
Action Model
computeProposalDistribution() (Update the action of prior sampled particles)
Sensor Model
computeNormalizedPosterior()
estimatePosteriorPose() (Compute pose from top 20% particles)
Publish pose and particles
updateMap

Fig. 1: Overview of SLAM Algorithm

2) Localization - Action Model: Monte Carlo Localization (MCL) is a particle-filter-based localization algorithm. And implementing MCL requires 3 components:

- Action model: Odometry model or velocity model
- Sensor model: Beam model or likelihood field
- Particle filtering: Importance sampling or low variance sampling

We use an odometry model to compute the action the robot has taken. Each timestep, the robot moves in a sequence of rotation $\rightarrow$ translation $\rightarrow$ rotation. First, we compute $(dx, dy, d\theta)$ by subtracting odometry from the

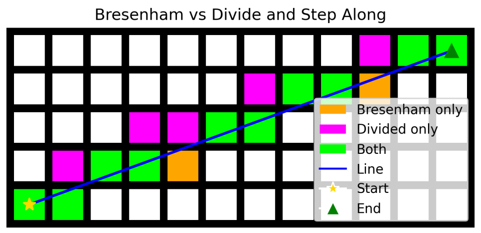


Fig. 2: Cell points between Bresenham's algorithm and divide and step along ray.

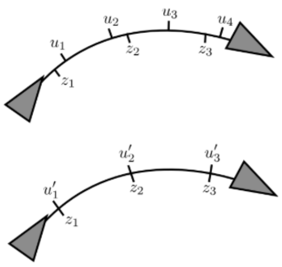


Fig. 3: Interpolating observations, u_i, $i = 1,2,3,4$ are the laser scans received during the odometry, z_j, $j = 1,2,3$ are the readings from the motor encoders. Image credit: Prof. Peter Gaskel

previous pose of mbot.

$$\Delta x = x_t - x_{t-1}$$
$$\Delta y = y_t - y_{t-1}$$
$$\Delta\theta = \theta_t - \theta_{t-1}$$

Then the first rotation is the angle difference between the translation difference and mbot's previous heading, the translation will be the L2-distance of dx, dy, and the second rotation will be the angle difference between $dtheta$ and rotation 1.

$$\delta_{rot1} = \arctan(\Delta y, \Delta x) - \theta_{t-1}$$
$$\delta_{trans} = \sqrt{\Delta x^2 + \Delta y^2}$$
$$\delta_{rot2} = \Delta\theta - \delta_{rot1}$$

Then we sample some normal distribution error to $\delta_{rot1}, \delta_{trans}, \delta_{rot2}$

$$\epsilon_{rot1} \sim N(0, k_1|\delta_{rot1}|)$$
$$\epsilon_{trans} \sim N(0, k_2|\delta_{trans}|)$$
$$\epsilon_{rot2} \sim N(0, k_1|\delta_{rot2}|)$$

Last, we update x, y, θ from $\delta_{rot1}, \delta_{trans}, \delta_{rot2}$ and $\epsilon_{rot1}, \epsilon_{trans}, \epsilon_{rot2}$

$$x+ = (\delta_{trans} + \epsilon_{trans})\cos(\theta_{t-1} + \delta_{rot1} + \epsilon_{rot1})$$
$$y+ = (\delta_{trans} + \epsilon_{trans})\sin(\theta_{t-1} + \delta_{rot1} + \epsilon_{rot1})$$
$$\theta+ = \epsilon_{rot1} + \epsilon_{rot2}$$

k1_	**k2_**
0.002	0.025

TABLE II: Parameters for odometry action model

k_1, k_2 are determined by running example logs provided, in `drive_maze.log`, there is a huge turn on the bottom right corner, initially $k_1 = 0.005$, and this makes the angle after that turn differ a lot from ground truth, so we lowered k_1 to 0.002 and retrieved a better result.

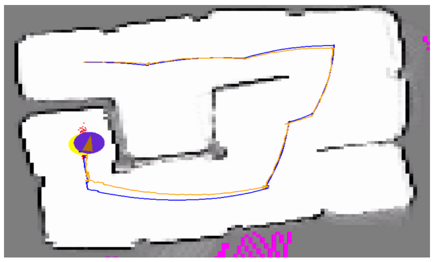


Fig. 4: $k_1 = 0.005$ causing the turn estimate to be bad, and the slam position differs a lot from ground truth

3) Localization - Sensor Model: Sensor models are important in SLAM because they find the probability that a lidar scan matches the hypothesis pose given a map, and they resample the particles based on weights before applying the action model again. Therefore, it converges the particles in the map, and gains a better prediction of where the robot is in the map. And two models were provided, **beam model**, and **likelihood field**.

For **beam model**, we cast the ray in the grid and assign log odds to each case:

1) Ray terminates at the first obstacle
2) Ray terminates before an obstacle
3) Ray terminates after an obstacle

If a ray ends at the first obstacle, it likely identifies the correct grid cell. Rays ending before obstacles provide little information, while those ending beyond obstacles suggest incorrect localization, as rays can't pass through obstacles. Initially, we avoided the beam model due to its computational cost, requiring Bresenham's algorithm to evaluate ray paths. However, by processing every 10th scan, it can run in real time. During competition tasks, we found the beam model more accurate than the simplified likelihood field. Scoring for each case is shown in Table III.

For **likelihood field**, we implemented the simplified version only. We only check the endpoint of the laser scan.

1) If the laser hits an obstacle, add the log odd to the scan score

On obstacle	Before obstacle	After obstacle
`NormalPdf(2)`	`NormalPdf(8)`	`NormalPdf(12)`

TABLE III: Beam model score for each ray

2) If there isn't a hit, check the cell before and after along the ray and take a fraction of the log odds into the score

Therefore, we do not need to go through the whole laser scan and only focus on the end of the laser, also because the likelihood of the point is given by a Gaussian distribution:

$$\frac{1}{\sqrt{2\pi\sigma}}\exp(-\frac{d^2}{2\sigma^2})$$

where d is the distance to the nearest obstacle and σ is the standard deviation, so the field will be smoother.

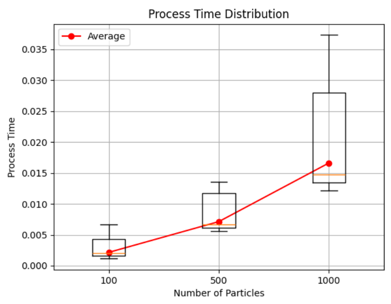


Fig. 5: Process time distribution for 100, 500, 1000 particles

4) Localization - Particle Filter: When there are more particles, the particle filter works better, but also slower. We want the filter to be running at least 10 Hz or updating every 0.1 seconds. With the table below, we

Number of particles	100	500	1000
Average process time (sec)	0.00216	0.01427	0.01662

TABLE IV: Average process time for 100, 500, 1000 particles, and the outliers are removed

can approximate it with a one-degree line $y = 1.619 * 10^{-5} \cdot x + 2.928 * 10^{-6}$, where y will be the average time to update, so let $y = 0.1$, x will be approximately 6,177 particles. Therefore, if we want a minimum 10 Hz update rate, we can use about 6,000 particles.

C. *Path planning with A* algorithm*

Among various pathfinding algorithms, A* efficiently computes the shortest path between two points by assigning a cost to each explored grid. The detailed implementation is provided in Algorithm 4.

Algorithm 4 A* Search Algorithm

```
Input: start node, goal node, distance map
 1: Initialize open set $OPEN \leftarrow \{start\}$
 2: Initialize closed set $CLOSED \leftarrow \emptyset$
 3: $g(start) \leftarrow 0$
 4: $f(start) \leftarrow g(start) + h(start)$
 5: while $OPEN$ is not empty do
 6:     $current \leftarrow$ PriorityOpenList.pop()  ▷ Node in $OPEN$ with lowest $f$ score
 7:     if $current = goal$ then
 8:         return extract path from current
 9:     Add $current$ to $CLOSED$
10:     for each neighbor $n$ of $current$ do
11:         if $n \in CLOSED$ then
12:             continue
13:         $g(current) \leftarrow g(parent) +$ distance$(parent, current) +$ distance$(obstacle)$
14:         if $n \notin OPEN$ then
15:             Add $n$ to $OPEN$
16:         else if $g(current) \geq g(n)$ then
17:             continue
18:         $parent[n] \leftarrow current$
19:         $g(n) \leftarrow g(current)$
20:         $f(n) \leftarrow g(n) + h(n)$
21: return failure
```

Algorithm 5 Exploration

```
Input: map, SLAM pose, frontiers
 1: Receive current map and pose from SLAM
 2: Find frontiers
 3: Plan a path to the closest frontier using A* algorithm.
 4: if path exist then
 5:     if selected path is valid and time threshold passed then
 6:         Set current path to selected frontier path
 7:         Publish current path
 8:     Publish status as IN_PROGRESS
 9: else if no reachable frontiers then
10:     Publish status as COMPLETE
11:     RETURN_HOME
12: else
13:     Publish status as FAILED
```

D. Map exploration

While the MBot is capable of performing SLAM and planning a path to a known destination, it still requires an exploration algorithm to autonomously navigate toward unexplored areas. The ability to move safely and efficiently in unknown environments is crucial for robust autonomous operation. To address this, we developed a map exploration algorithm that enables MBot to actively seek out and explore unknown regions of the environment.

In the exploration algorithm 5, we introduce a time threshold for updating the path. This ensures that the MBot does not immediately switch to a new frontier upon detecting the current one. Without this delay, the robot may prematurely abandon its path, missing the opportunity to gather richer sensor data and improve map quality, which can lead to suboptimal SLAM results.

E. Map Localization with Estimated and Unknown Starting Position

In order to localize MBot in a known environment, we first randomly spread the particles across the map's open space, then it is updated using the sensor model only.

We used `RandomPoseSampler` to sample around the map with uniform distribution. We found a bug here, the function `isOccupied` determines whether the cell is occupied with the statement `map.logOdds(x,y)>0`, however, the occupancy grid cell is initialized with 0, therefore, the space outside the map (gray area) is determined as free space, causing `RandomPoseSampler` to sample across the whole occupancy grid map instead of the white areas (`map.logOdds<0`) only.

After the particles are randomly initialized, we update prior particles by taking the top 10 weighted particles and generating 25 additional particles around each one. Each new particle is defined as $(x + \Delta x, y + \Delta y, \theta)$, where Δx and Δy are sampled from a normal distribution with the range of $-0.05 \sim 0.05$, and θ is randomly chosen from a uniform distribution in the range $-\pi \sim \pi$. To maintain a constant number of particles, the 250 lowest-weighted particles are removed.

In the end, the highest-weighted particles will converge toward the correct position, and the randomized particles at each iteration will have little to no effect on the final result. Occasional slight misalignments may occur, but this is acceptable, as the robot's movement allows the action model updates to gradually correct them over time. This part of the code is in `particle_filter.cpp` line 218-287.

III. Results

A. Motion Controller

1) Calibration: Tables V present the calibration results showing positive slope(S_{+ve}), negative slope(S_{-ve}), positive intercept(I_{+ve}) and negative intercept(I_{-ve})

	μ_a	μ_b	σ_a^2	σ_b^2
S_{+ve}	0.068	0.063	2.45×10^{-7}	3.72×10^{-7}
S_{-ve}	0.064	0.070	2.00×10^{-7}	4.95×10^{-7}
I_{+ve}	0.050	0.056	1.80×10^{-5}	1.08×10^{-5}
I_{-ve}	-0.065	-0.048	1.31×10^{-5}	4.46×10^{-6}

TABLE V: Mean and variance for Motor A and B

The results in Figure 6 showed a generally linear relationship between PWM input and wheel speed, but with some notable variation. Repeated calibration runs yielded slightly different values, indicating minor but consistent variation in the system. There are several factors that could contribute to this variation:

- Surface imperfections in the concrete can cause inconsistent traction.
- Motor and gearbox tolerances, such as backlash or minor manufacturing differences.
- Battery voltage fluctuation during calibration may slightly affect motor output power.
- Encoder noise introduces measurement uncertainty when estimating actual wheel speed.
- Slippage could occur when the robot is accelerating or decelerating too quickly.

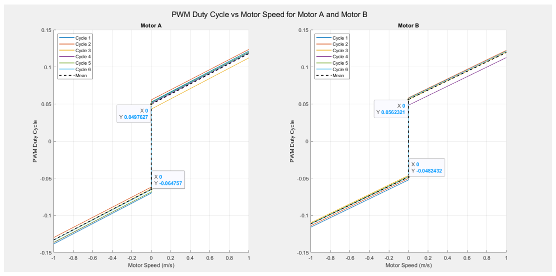


Fig. 6: Motor PWM Linear Mapping Calibration

2) *Trajectory Following:* Table VI shows the dead reckoning of our Mbot when we commanded it to drive a 1m square four times using the pure pursuit algorithm for motion control. Figure 7 shows our linear velocity and angular velocity when Mbot is driving a square.

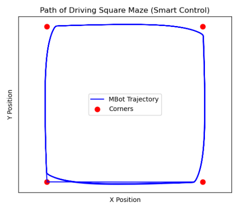
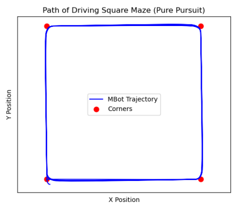
Dead Reckoning Comparison of 4x1 m square	
	
Smart control path	Pure pursuit path

TABLE VI: Smart Controller and Pure Pursuit Controller

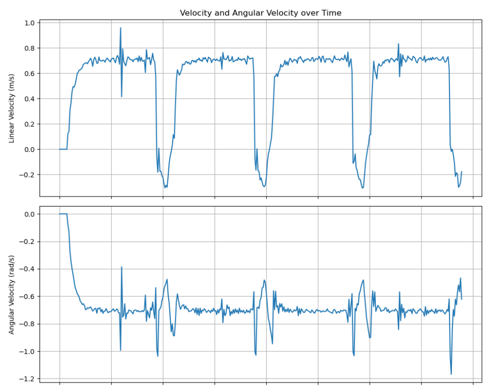


Fig. 7: MBot's linear and angular velocity—driving a square

B. *Simultaneous Localization and Mapping (SLAM)*

1) *Occupancy Grid Map:* The mapping result from `drive_maze.log` is included in Figure 8. The result shows a clear map, and all of the walls and poles were fully captured.

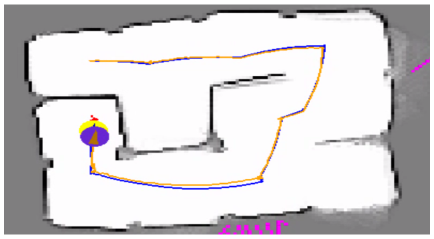


Fig. 8: Mapping from drive_maze.log

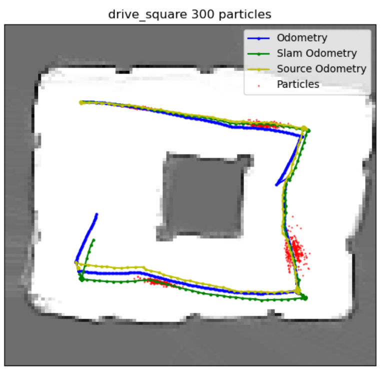


Fig. 9: Plot of 300 particles from `drive_square.log`, the log-player is ended earlier, because the log file is corrupted at the end

2) *Localization with Particle Filter:* Our particle filter provided highly accurate estimates of the MBot's pose, resulting in SLAM odometry that closely matched the ground truth trajectory. Additionally, after implementing

random initialization of particles across the map, the system was able to correctly and consistently localize the robot within the environment. A demonstration of localizing MBot in a simplified environment with a random initial pose can be seen here.

3) System Performance: To evaluate our SLAM system, we ran the robot with the provided `drive_maze_full_rays.log` and compared it with ground truth pose data; the result can be seen in Figure 10. We can see that our SLAM aligns with the ground truth quite well. The RMSE between points can be seen in Figure 11. The maximum error is only about 3.25 cm.

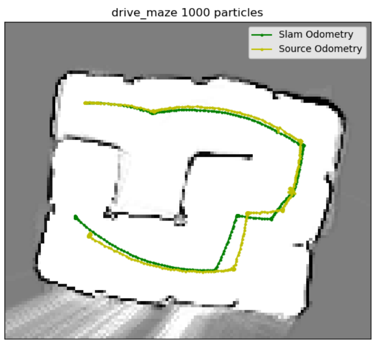


Fig. 10: The estimated pose (green) from our SLAM system against the ground-truth poses (yellow) in drive_maze_full_rays.log

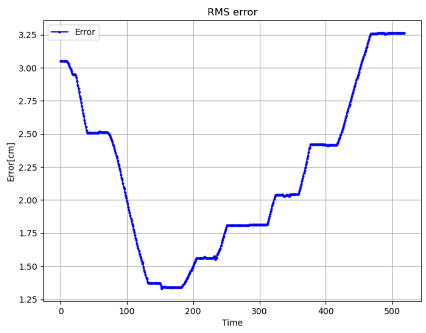


Fig. 11: The RMS error as the SLAM system goes through drive_maze_full_rays.log

C. Path Planning

The A* path planning algorithm initially completed 5/6 of the given test cases successfully. The one test case that failed was the Narrow Constriction Grid test three. The A* test algorithm incorrectly finds a path through the narrow gap. And we improved the algorithm by increasing the params.minDistanceToObstacle to 1.01 to increase the safety margin required for a cell to be considered obstacle-free.

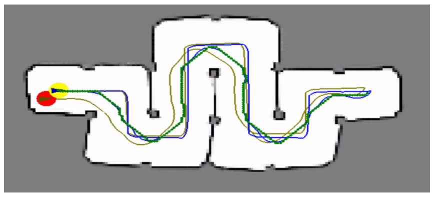


Fig. 12: Comparison of Planned and Executed Paths for Mobile Robot Navigation; Green line representing the planned path and blue line representing the real path.

As observed in Figure 12, where the planned path is shown in green while the actual path taken by the mBot is shown in blue, the driven path closely follows and overlaps with the planned path, indicating accurate path tracking. The deviation between the A* planned path and the SLAM pose indicated the small inaccuracies of the localization algorithm rather than errors in path planning.

The statistics on our path planning execution times for each of the example problems are reported below:

Test Cases	Min (μs)	Mean (μs)	Std dev
`convex_grid`	39	58	19
`empty_grid`	1260	1493	200.3
`maze_grid`	1304	4026.75	2639.47
`narrow_constriction`	952	1120.5	168.5
`wide_constriction`	871	1132.33	306.835

TABLE VII: Execution Time—Successful Plan Attempts

Test Cases	Min (μs)	Mean (μs)	Std dev
`convex_grid`	15	15.5	0.5
`empty_grid`	83	91.5	8.5
`filled_grid`	18	304.6	470.1
`narrow_constriction`	14	8.4e+06	1.3e+07
`wide_constriction`	18	18	0

TABLE VIII: Execution Time—Failed Planning Attempts

IV. Discussion

A. Motion Controller & Odometry

Our calibration results confirm a linear relationship between the motor speed and the PWM input signal. The nonzero intercept, known as the punch, arises due to wheel-floor friction and inherent motor losses. By combining IMU data with odometry, we achieved improved accuracy in motion estimation.

During both linear and rotational motion, we observed oscillations caused by noise in the encoder and sensor data. To address this, we applied a low-pass filter, which effectively reduced the disturbances and led to smoother responses from the PID controller.

B. Simultaneous Localization and Mapping (SLAM)

Our implementation of SLAM enabled MBot to autonomously navigate and explore unknown environments. The system utilized a particle filter-based localization algorithm working side by side with an occupancy grid mapping. Throughout our tests, the system demonstrated a strong alignment between estimated SLAM pose and ground-truth poses, with a maximum localization error of only 3.25 cm.

On the localization side, we employed the Monte Carlo Localization (MCL) using an odometry-based motion model and both beam and likelihood field sensor models. We finally found that the beam model provided higher localization accuracy during competition. The number of particles used in the filter was another important consideration. Our analysis estimated that maintaining around 6000 particles allowed for real-time updates at a 10 Hz rate, which was sufficient for smooth performance without compromising accuracy. Overall, the SLAM system provided a reliable foundation for map building and localization.

*C. Path Planning A**

The A* planning algorithm is particularly well-suited for this project due to the availability of known goal locations—specifically, the centroids of detected frontiers. This informed search strategy leverages heuristic guidance to efficiently compute paths in grid-based environments. In contrast, if frontier locations were unknown, such as in the absence of a LIDAR sensor or a frontier detection algorithm, an uninformed search method would be necessary.

An alternative approach worth considering is the Rapidly-exploring Random Tree (RRT) algorithm. RRT generates paths by expanding a tree towards randomly sampled points in the configuration space, enabling broad exploration. While RRT is advantageous in high-dimensional or continuous spaces, it is less optimal in low-degree-of-freedom systems like the mBot.

A hybrid strategy, such as combining RRT with A* (e.g., RRT*), could incrementally improve path quality through random exploration and rewiring. However, given our application's constraints and the availability of reliable frontier information, A* remains the most efficient and effective solution.

Our A* implementation demonstrated strong performance in both test cases and physical deployment, validating its suitability for structured, low-complexity mobile robot navigation tasks.

D. Map Exploration

Successful autonomous exploration required the seamless integration of multiple subsystems, including odometry, SLAM, motion control, and A* path planning. A critical component of our exploration strategy was frontier detection, which is the process of identifying the boundary between known and unknown areas in the occupancy grid. These frontiers serve as navigation goals for the robot to continue exploring unmapped regions.

One major challenge we encountered was the A* algorithm frequently switching paths when multiple frontiers were detected simultaneously. This behavior caused the robot to abandon partially explored areas prematurely, resulting in incomplete exploration within the maze. After we integrated a time threshold into the algorithm, the exploration efficiency was significantly improved, and the overall robustness of the algorithm was enhanced.

V. CONCLUSION

In this project, we successfully integrated hardware and software components to develop an autonomous MBot system capable of navigation, localization, mapping, and smooth movement.

The implementation of a PID wheel speed controller, gyrodometry, and the pure pursuit algorithm enabled smooth and accurate trajectory tracking. Through the use of a particle filter-based SLAM system and occupancy grid mapping, the MBot effectively built a 2D map of the unknown environment and localized itself. We also demonstrated the utility of both beam and likelihood field sensor models for robust localization. A* path planning provided efficient and reliable navigation, and the exploration module enabled autonomous frontier-based map expansion. Together, these systems formed a cohesive and adaptable robotic platform.

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, 1996, pp. 423–428 vol.1.

[2] Simultaneous localization and mapping. Available: https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping. [Accessed: 13 April 2025].

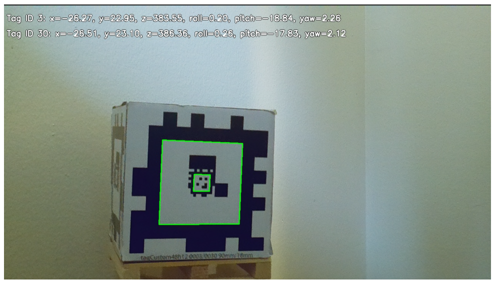
[3] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2006. [Online]. Available: http://www.probabilistic-robotics.org/

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

VI. Appendix

A. AprilTag-Based Navigation

After a successful calibration of the MBot to determine the intrinsic matrix using the checkerboard, the extrinsic matrix was subsequently derived from the spatial offsets between the robot frame and the camera frame, as determined from the CAD model.



```
Tag ID 3: x=-26.27, y=22.95, z=383.55, roll=0.20, pitch=-18.84, yaw=2.26
Tag ID 30: x=-26.51, y=23.10, z=386.36, roll=0.26, pitch=-17.83, yaw=2.12
```

Fig. 13: AprilTag detection results showing Tag ID(s) identified in the camera frame.

The robot was controlled toward the AprilTag by publishing velocity commands based on the position and orientation estimates from the apriltagandcone detection Python script, as shown in Fig. 13, while maintaining a safe stopping distance from the tag.

B. Competition

1) Event 1: In this task, two runs were required. In the first run, the MBot followed a provided path while constructing a map of a convex maze. Using our SLAM system, we accurately mapped the arena and successfully returned to the origin pose with high precision. In the second run, the MBot was tasked with navigating through the maze twice as quickly as possible. Leveraging fine-tuned PID wheel controllers and a Pure Pursuit motion controller, we completed the task within 40 seconds, returning to the origin with a final position error of less than 10 cm.

2) Event 2: The task is to use SLAM to build a map of the environment while simultaneously detecting the positions of colored cones. Once the map and cone positions are established, the mBot is placed at a random location within the maze. From there, it must navigate to each cone in rainbow color order and finally return to its starting point.

Mapping is performed using an exploration algorithm that identifies frontiers and builds the map using SLAM. Cone detection is handled by a Python script that utilizes a fine-tuned YOLO model to detect cone colors and their positions, which are then transformed into map coordinates. To reduce the impact of false positives, a cone's position is only updated when a detection with higher confidence is received.

After mapping is complete, the cone positions are written to a text file and sorted in rainbow order. We also found that LCM message handling and cone detection needed to run in separate threads—since cone detection takes longer to process, running both in the same thread caused timing issues and resulted in incorrect cone positions.