ROB 550 Botlab Report

Leo Bringer, Sisir Potluri, Saket Pradhan, and Alyssa Tamvakis
{lbringer, sisir, saketp, atamvaki}@umich.edu

***Abstract*—The goal of the Botlab project is to write software for an MBot (a two-wheel differential drive robot) that allows for autonomous mapping, SLAM-based localization, and path traversal. To achieve this, we implement motor control through PID tuning and odometry, particle filter and mapping algorithms, as well as path planning functionality. We also design and build a forklift to enable block movement. To evaluate the performance of the complete system with these components, we participate in a competition of four events.**

I. Introduction

This lab report uses the classic Mbot design, which includes two differential drive motors with two parallel wheels with a caster in the back. For one MBot, modification is done to the original design to attach a forklift for block movement. For environment sensing, this robot has a 2D-LIDAR, the rays of which are used in SLAM. A 3-DOF odometry system that relies on motor encoders and an IMU is used for pose estimation. Remote SSH access allows development, while relevant information is transferred via LCM communication. The use of a Jetson Nano board enables high-level connectivity between the sensing tools and the MBot's firmware, while the Pico+ Controller executes the firmware, which performs motor control and reads IMU data.

This project focuses on enhancements made to both the firmware code (for improved odometry and motor control) and autonomy implementations, such as optimal path-finding, SLAM, and motion control. We describe the implementation methods for each of these components, the results from each, and conclusions we can draw from those data.

II. Methodology

A. Motion Controller and Odometry

1) Calibration: We use the PWM calibration function that takes an MBot's calibration data and polarities to compute each motor's PWM command from a wheel velocity. We consider a calibration routine successful when the polarity is correct, both encoders function, and neither the slope nor punch of a single motor's calibration function has a magnitude over 0.10.

2) Odometry: We implement an updated method to compute the angular body velocity of the MBot. In the basic functionality, this is calculated using each wheel's measured velocities based on its motor's encoder ticks. Formulas for the linear body velocity $\hat{v}_x$ and angular body velocity $\hat{w}_z$, along with the odometry pose calculations, are specified in Equation 1. R is the wheel radius, $\hat{w}_L$ is the left wheel's angular velocity, and $\hat{w}_R$ is that of the right wheel.

$$\begin{aligned} \hat{v}_x &= R * (\hat{w}_L - \hat{w}_R)/2 \\ \hat{w}_z &= R * (-\hat{w}_L - \hat{w}_R)/2R \end{aligned}$$

$$\begin{aligned} x_t &= x_{t-1} + \hat{v}_x * \cos\theta_{t-1} * T \\ y_t &= y_{t-1} + \hat{v}_x * \sin\theta_{t-1} * T \\ \theta_t &= \theta_{t-1} + \hat{w}_z * T \end{aligned} \tag{1}$$

In the updated odometry, we rely solely on the φ from IMU data to compute $\hat{w}_z$. Our updated $\hat{w}_z$ calculation is specified in Equation 2. T is the firmware loop's time period. The remaining calculations are the same.

$$\hat{w}_z = (\varphi_t - \varphi_{t-1})/T \tag{2}$$

3) Motor Control: For wheel velocity controllers, we connect the feed-forward term with a PID-based feed-back term to create a closed loop controller. We also add several low-pass filters to reduce noise in reference and output terms. These include filters for the commanded v and w, the measured velocity of each wheel, and the final PMW output after combining the feed-forward term and the feed-back term. While we have separate PID filters for each wheel's velocity, the proportion term K_P, integral term K_I, and derivative term K_D are the same for both. We use separate filters for each motor's K_I gain and reset it if it accumulates to 1.0 to handle integral windup. Without loss of generality, if we consider the left wheel, our input to the PID controller is the difference between the commanded wheel velocity w_L and the measured $\hat{w}_L$. The PID output is added to w_L before that sum is input to the PMW calibration function (or subtracted, if the wheel polarities multiply to -1). The terms w_L and $\hat{w}_L$ are graphed to support the tuning process:

- Set K_P, K_I, K_D to 0 and execute a straight path. $\hat{w}_L$ reaches w_L after a delay.

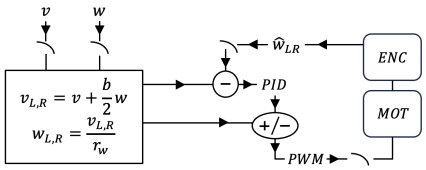

Fig. 1: Closed Loop Controller

- Increase K_P until $\hat{w}_L$ doesn't reach w_L any sooner and slight oscillations appear. Reduce K_P to the previous hypothesis.
- Increase K_D until the MBot starts to jerk. Reduce K_D to the previous hypothesis.
- Increase K_I until the steady state $\hat{w}_L$ is close to w_L. We ensure the value is within 0.02 m/s of w_L.

In our first tuning attempt, we tune the PID parameters without low-pass filters and then add the filters with arbitrary time constants that make acceleration and turns more smooth. This causes strong swerves in trajectories with turns, like a square. In our second attempt, we first gradually increase time constants of the low-pass filters from 0 until sudden velocity shifts are performed smoothly. This produces slight swerves in the square-path. We then tune our PID parameters using the above method, and find the swerves disappear. Once the graph of w_L and $\hat{w}_L$ for a straight path appears the way we expect (a small offshoot, no oscillations, and an accurate steady state), shifts in v and w are executed smoothly, and a square run has no swerves, tuning is done. Our closed loop controller is Figure 1.

The gains for our left and right motor control PID filter are in Table I. The parameters of our low-pass filters are in Table II.

TABLE I: Left and Right Motor PID Parameters

K_P	K_I	K_D	Tf	dt
2.2	0.00002	0.00001	0.00066	0.001

TABLE II: First Order Low-Pass Filter Parameters

Term:	v	w	$\hat{w}_L, \hat{w}_R$	PWM_L, PWM_R
dt	0.02	0.02	0.02	0.02
tc	0.15	0.07	0.05	0.1

4) Motion Control: We rely on the provided `SmartManeuverController` after finding that it performs fairly well with our updated odometry. The formulas for calculating the v and w to command are in Equation 3. x' is the target, while x is the current one, and the same is true for y.

$$\begin{aligned} v &= \sqrt{(x'-x)^2 + (y'-y)^2} \\ w &= 2.5 * (\arctan(y'-y, x'-x) - \theta) \end{aligned} \tag{3}$$

When $\arctan(y'-y, x'-x) - \theta$ has magnitude above $\pi/4$, we set $v_x = 0$ and only turn. Until the target pose is met, the controller commands a v and w; it is similar to a Rotate-Translate-Rotate strategy.

B. Simultaneous Localization and Mapping (SLAM)

Since odometry errors accumulate over time, we rely on a SLAM system that leverages Monte-Carlo localization with a Particle Filter to estimate the MBot's pose. Figure 2 gives an overview of this.

1) Mapping: To create a map of the environment, we use a mapping update algorithm following Bayes' rule with Log Odds, the log of the ratio of the probability a cell is occupied divided by the probability that it is free. We convert odometry poses into a grid and cast a LIDAR laser scan into the cells to update their Log Odds. For each ray scan, we find the endpoint in the grid and increase its Log Odds by a certain amount. This adds edges in the map at locations where the laser rays terminate. We then use Bresenham's Algorithm to project the LIDAR rays into cells that are passed. We decreased Log Odds by a certain amount for all cells passed by the rays to indicate they're free. Since the odometry and laser scans are updated at different rates, we interpolate the odometry readings to align them with the laser scans.

2) Particle Filter & Resampling: The Particle filter is a Bayes filter represented using weighted samples. Each sample represents a pose (x, y, θ) weighted by their likeliness to be correct. At each time step, the distributions are initialized with a resampling of the previous distribution and are updated using the Action & Sensor models. For the resampling algorithm, we perform a low variance sampler [1] to avoid over-amplified variance and create a prior distribution. The sampling is based on the weight of the particle of the previous distribution, and the high-weighted particles should be sampled more frequently. We only perform resampling when the MBot has moved, according to odometry data.

3) Action Model: The action model predicts the MBot movement using probabilistic methods, taking into account the uncertainty in motion due to factors like sensor inaccuracies and mechanical imprecision. It calculates the robot's translation and rotation based on odometry data and models the uncertainties in these movements. We create a proposal distribution by updating the particles of the prior distribution resampled using the changes in odometry and adding some Gaussian randomness to model the errors and disperse the resulting pose, and this is defined in Equation 4. We note that $dx = x_t - x_{t-1}$, $dy = y_t - y_{t-1}$, and $d\theta = \theta_t - \theta_{t-1}$.

$$\begin{aligned} \text{trans\_} &= \sqrt{(dx)^2 + (dy)^2}, \\ \text{rot1\_} &= \text{angle\_diff}(\text{atan2}(dy, dx), \theta_{\text{t-1}}), \\ \text{rot2\_} &= \text{angle\_diff}(d\theta, \text{rot1\_}). \end{aligned} \tag{4}$$

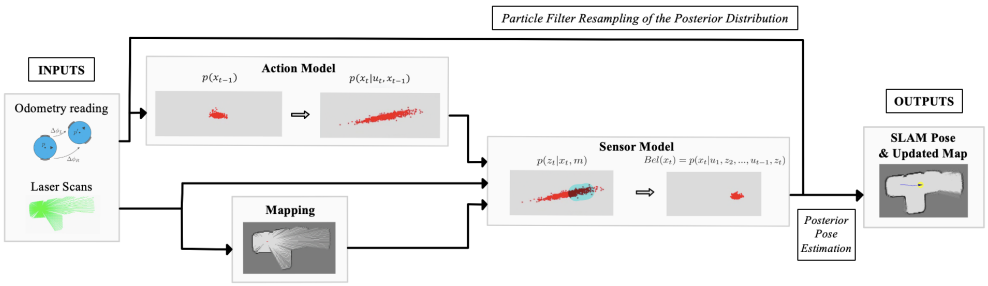

Fig. 2: SLAM Block Diagram

The Standard Deviations for Probabilistic Modeling Equations are given in Equation 5, where $k1$ and $k2$ represent the rotation and translation uncertainty factors.

$$\begin{aligned} \text{rot1Std\_} &= \sqrt{k1 \cdot |\text{rot1}|}, \\ \text{transStd\_} &= \sqrt{k2 \cdot |\text{trans}|}, \\ \text{rot2Std\_} &= \sqrt{k1 \cdot |\text{rot2}|}. \end{aligned} \tag{5}$$

While fine-tuning the coefficients for rotation ($k1$) and translation ($k2$) uncertainty, we aim to strike a balance between the accuracy of the model and its ability to handle real-world uncertainties, ensuring reliable and robust performance of the robot. Hence, to choose $k1$ and $k2$, we perform hyper-parameter tuning with various combinations of $k1$ and $k2$ on different log simulations where we focus on the evolution the particles across a complete trajectory. We leverage the action-only mode of the SLAM algorithm on different pre-set log simulation files, and obtain the following parameters: $k1 = 0.001$ and $k2 = 0.001$.

4) Sensor Model & Pose Estimation: After obtaining the proposal distribution, we use sensor data and the updated map to improve the particle distribution with a Sensor Model, referred as the Simplified Likelihood Field [1]. We find the probability that a LIDAR scan matches the hypothesis pose given the map. In our implementation, we assess the endpoints of the laser scans; if they correspond to an obstacle on the updated map (using the Log Odds), we add this value to a score for each particle, and if not, we do the same for the cell before and after the endpoint. These fitness scores enable the creation of new weights by normalizing scores across particles, resulting in the Posterior Distribution. Then, we estimate the Posterior Pose using a Priority Queue of the particles (each with a x, y, θ pose) and taking a weighted-average of the 15% highest-weight ones.

C. Planning and Exploration

1) Path Planning: Our chosen algorithm for path planning is A-Star, and we present some specifics of our algorithm:

- `g_cost()` has two components — the distance from the parent node to the child node (1 or $\sqrt{2}$ if diagonal) and a penalty for closeness to obstacles to avoid trajectories that follow walls.
- `h_cost()` uses a heuristic that allows for diagonal travel, and its value is the minimum distance from the goal node.
- `expand_node()` finds the 8 adjacent kids of a node, checks if they are sufficiently far from obstacles using the occupancy map, and ensures they are not in the set of closed nodes.
- If a kid node is already in the open set, its g-cost and parent node are updated if its current discovery has a lower g-cost than the existing one.

Our A-Star algorithm has some unique differences from the regular version, and we define its pseudo-code in Algorithm 1.

Algorithm 1 A-Star Algorithm with Easing Factor

Input: start, goal, radius = 0.2
Output: path

if $start = closed$ **then** return $makePath(start)$
$open \leftarrow start$; $closed \leftarrow \emptyset$; $count \leftarrow 0$
$easingFactor \leftarrow -0.1$ ▷ 10cm of strictness
while $open \neq \emptyset$ & $count < 50000$ **do**
  $n \leftarrow open.pop()$; $count \mathrel{+}= 1$
  for $kid : expand(n)$ **do**
    if $kid = goal$ **then** return $makePath(kid)$
    $dist = obstacleDist(kid)$
    if $dist > radius - easingFactor$ **then**
      if $kid \cap closed = \emptyset$ **then**
        $kid.f = g(n, kid) + h(kid)$
        $open \leftarrow kid$
    $closed \leftarrow n$
  if $open = \emptyset$ & $easingFactor < 0.04$ **then**
    $open \leftarrow start$; $closed \leftarrow \emptyset$
    $easingFactor \mathrel{+}= 0.01$ ▷ Ease by 1cm
return $\emptyset$

The Easing Factor is an offset for the minimum obstacle distance required for a cell to be considered for

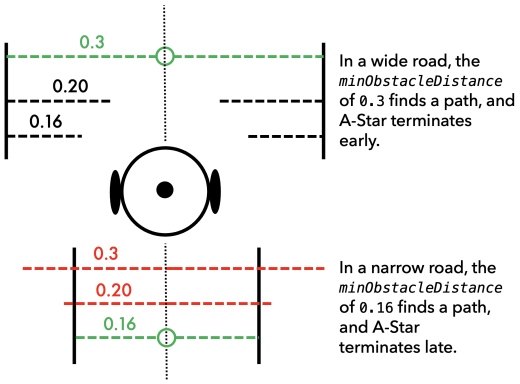

Fig. 3: Strict and Easy Obstacle Distances

Algorithm 2 Map Exploration Algorithm

Input: map, home
Output: map

$frontiers \leftarrow findFrontiers()$;
$current \leftarrow home$;
while $frontiers \neq \emptyset$ **do**
  $path \leftarrow findPath(current, frontiers)$
  $goal \leftarrow path[-1]$;
  while $abs(current - goal) > 0.05$ **do**
    $motionControl(path)$
    $map, current \leftarrow SLAM(map, current)$
  $frontiers \leftarrow findFrontiers()$;
return map

traversal. It starts at -0.1, adding 10cm of strictness. If no path optimal exists, the offset increases by 0.01, and A-Star repeats. This continues until either a path is found or the offset exceeds 0.04, at which point the constraint has been maximally eased. The intention is to consider paths with the widest traversals before increasingly narrow ones. In practice, we use a robot radius of 0.2 or 20cm, so the algorithm repeats with offset minimum obstacle distances of 0.3 to 0.16 at decrements of 0.01. A visual of this logic is Figure 3.

Instead of having a vector of nodes be the closed set, which would require $O(n)$ time to traverse when a candidate kid is evaluated, we utilize an unordered set, which can be traversed in $O(log(n))$ time. Since each node has a unique (x, y) cell, a unique decimal can be constructed as $x.y$ (for example, 30.45 would correspond the cell at $(30, 45)$). For a candidate kid, we check if its unique decimal is already in the set prior to appending it to the parent's kids, and we push the decimal of a node after it's explored. We prune nodes (except start and goal) that are displaced 5 cells or less in the x or y directions from the previous node that isn't pruned.

2) Exploration: We implement logic to traverse and map an environment from a starting pose before returning to the home pose. A few of the components from the pre-set implementation are specified here:

- A frontier is a border between free and unexplored space. The logic to find all frontiers and to construct a path to the nearest one is pre-set.
- A state machine to identify the state and status of the robot lets the exploration program determine what to do next.
- A motion planner server plans paths for the exploration program and evaluates the validity of goal cells.

We specify the logic for planning paths during initial exploration in Algorithm 2.

Here are some important aspects of our implementation that aren't specified in the algorithm:

- After completing a path and before traversing a new one to the next frontier, we do a sanity check that the starting node of the new path is within 0.05m of the previous path's goal pose.
- For the logic to return the the home pose, we plan a path from the current pose after initial exploration to the global pose marked as home.
- The method `initializeFilterRandomly()` from the Particle Filter program defines logic for initial localization when the initial location isn't known. This method samples particles randomly throughout the map with equal weights for all particles.

D. Forklift Mechanism

The forklift mechanism enables the MBot to lift and stack cardboard crates, based on a design featuring a doubled-threaded screw system. 2 intricately designed threaded screws, each accompanied by a plate guide, work in tandem to provide enhanced stability and precision. These plate guides are securely fastened to the forklift's fork, forming one robust and cohesive unit. To ensure reliability, we opt to manufacture the entire assembly via 3D printing using PLA with a 70% infill. However, recognizing the specific stress loads of the 2 screws and plate guides, we fabricate them in SLS Nylon, to withstand rigorous operational conditions.

Two Daisychain Dynamixel XL-320 motors are used to rotate the two threaded screws counterclockwise. These motors, controlled by the Pico+, are placed in 2 symmetrically mirrored mounts on the right and left extremities of the front of the M-Bot, attached to the bottom of its base plate. Each motor mount has a cover that secures the motor in place. We use 4x 6mm Lego connectors per motor to mount the threaded screws normally on top of the motors, such that the gear ratio of the

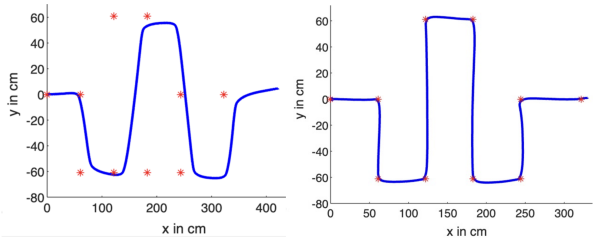

Fig. 4: (x, y) Odometry Before and After Using IMU

motors and the screws remains 1:1. To further enhance the system's stability, the plate guides are designed to rest against two support beams, effectively curbing any undesired movements, wobbles or rotations.

III. Results

A. Motion Controller and Odometry

1) Motor Control: Analysis of the Motor Calibration slope and intercept from 6 calibrations of our team's MBots on a concrete floor is specified in Table III.

TABLE III: Calibration Functions

Positive	Slope (R)	Punch (R)	Slope (L)	Punch (L)
μ	0.065	0.071	0.071	0.065
σ^2	$3.96e-6$	0.0002	$1.93e-6$	$5.89e-5$
Negative	**Slope (R)**	**Punch (R)**	**Slope (L)**	**Punch (L)**
μ	0.067	-0.070	0.066	-0.070
σ^2	$2.147e-5$	0.0019	$4.277e-6$	0.00013

To evaluate our changes to odometry, we command the robot to traverse a simple maze by specifying v and w values that align with the maze (without using the Motion Controller) and plot the odometry readings. The resulting plots are Figure 4. Red stars are target turns in the maze, while the blue line is the (x, y) odometry output.

To perfect the motor control, we command the robot using the Motion Controller to traverse a square once, and use the improved odometry to compare pose readings when executing with no PID or low-pass filters, with only low-pass filters, and with both PID and low-pass filters in motor control. The resulting odometry plots and the odometry readings after traversal (expected to be $x = 0, y = 0, \theta = 0$) are Figure 5.

2) Motion Control: While commanding the MBot to drive a square four times using the Motion Controller, we plot the dead reckoning readings using two different versions of the odometry — the original and improved versions. These plots are Figure 6.

Finally, as shown in Figure 7, we plot the measured linear and angular body velocities, $\hat{v}_x$ and $\hat{w}_z$, after having the robot execute a square path once. In this graph, the blue line is $\hat{v}_x$, and the red line is $\hat{w}_z$.

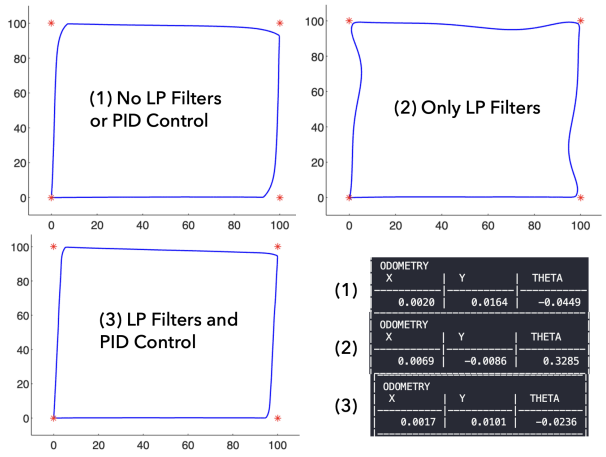

Fig. 5: (x, y) Square Path Odometry Outputs

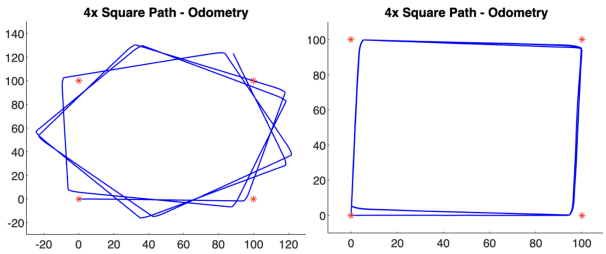

Fig. 6: (x, y) Dead Reckoning Pose for 4x Square Path Before and After Odometry Enhancement

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: We fine tune the magnitude of Log Odds to increase for cells where a laser ray terminates and decrease on cells where it passes. We perform some trial and error with different values of `kHitOddsArg` and `kMissOddsArg`, test them on pre-set log files, and chose those which produce the sharpest map with bold edges. In Figure 8 there are some examples of

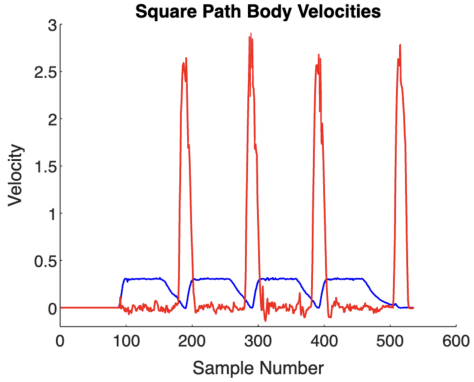

Fig. 7: $\hat{v}_x$ and $\hat{w}_z$ in Square Path

Fig. 8: Map Improvement when Tuning Log Odds

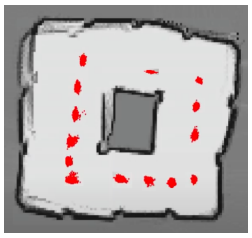

Fig. 9: Drive Square with 300 Particles at regular intervals

the obtained results. For the left one, `kHitOddsArg` = 3 and `kMissOddsArg` = 2, and for the right one, `kHitOddsArg` = 5 and `kMissOddsArg` = 1.

2) Particle Filter: We report the computational performance of the Sensor Model for different particle counts. The time taken to update the particle filter at varying particle amounts is in Table IV. These results help estimate the maximum number of particles that the filter can support while operating at a frequency of 10Hz which is approximately 10000 Particles in our case. Figure 9 represents the particle filter applied using 300 particles on the a square-path's log file, with particles plotted at regular time intervals to illustrate the evolution of the particle distribution.

We compute the error in x, y, θ values of the SLAM pose computed by the full SLAM implementation using a log with ground-truth pose data; the log is a playback of a run through a maze-like arena, given in the pre-set `drive_maze_full_rays.log` file. These results are given in Table V. Figure 10 compares the SLAM pose to odometry pose when the MBot traverses a simple maze with several turns.

TABLE IV: Time taken to update the particle filter with varying number of particles

Number of Particles	Time (ms)
100	4.89
500	6.78
1000	10.46

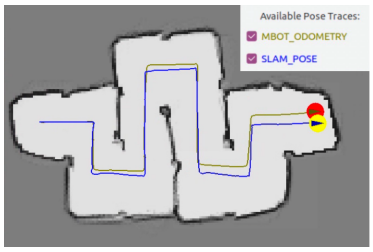

Fig. 10: Comparison SLAM & Odometry

TABLE V: Error Between Ground-Truth and SLAM Pose of a Maze Localization with 1000 Particles (in m)

$n = 532$	$e = \|x - \hat{x}\|$	$e = \|y - \hat{y}\|$	$e = \|\theta - \hat{\theta}\|$
$Max(e)$	0.2049	0.3077	0.3680
$\mu(e)$	0.1206	0.1668	0.0938
$\sigma(e)^2$	0.0047	0.0068	0.0022
$RMSE$	0.1389	0.1861	0.1050

C. Planning and Exploration

1) Path Planning with A-Star: We run existing test cases that to evaluate our A-Star implementation on various maps. Out of 5 tests cases (a filled, narrow, wide, convex, and maze map), all of them pass the final implementation. One sub-test fails for the maze map prior to the introduction of the Easing Factor. We present execution time statistics of the final implementation in Table VI, along with the average runtimes in two other implementations: one without the Easing Factor and one with a vector-based representation of the closed set instead of an unordered set. These are in Tables VII and VIII.

TABLE VI: A-Star with EF (Time in μs)

PASSING	Filled	Narrow	Wide	Convex	Maze
$Min.$	-	2211	3026	209	1762
$Max.$	-	3545	54703	1120	12541
μ	-	2878	21136	665	4805
σ	-	667	23760	456	4488
FAILING	Filled	Narrow	Wide	Convex	Maze
$Min.$	23	23	24	23	-
$Max.$	25	437683	24	45	-
μ	23.6	145918	24	34	-
σ	0.8	206309	0	11	-

TABLE VII: A-Star without EF (Time in μs)

PASSING	Filled	Narrow	Wide	Convex	Maze
μ	-	2880	21020	618	1725
FAILING	Filled	Narrow	Wide	Convex	Maze
μ	22.6	105820	21	27.5	687.667

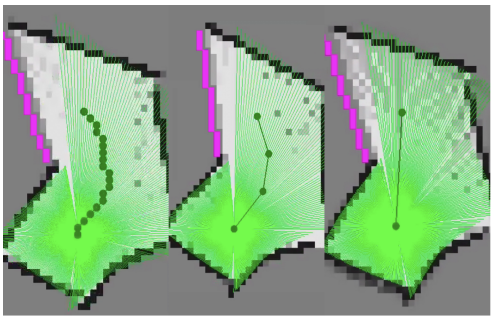

Fig. 11: Path when Pruning Every 0, 5, and 10 Cells

Fig. 12: Sequential Paths and Resulting Traversals

TABLE VIII: Sub-Optimal A-Star with EF (Time in μs)

PASSING	Filled	Narrow	Wide	Convex	Maze
μ	-	3163	114186	625.5	13025
FAILING	Filled	Narrow	Wide	Convex	Maze
μ	27	$1.466e+07$	24	29.5	-

In Figure 11, we present the results of path pruning for a path in a U-shape environment. The cases are when there is no pruning, when a node in every 5 map cells is pruned, and when a node in every 10 cells is pruned.

2) Exploration: We present a path planned by the Motion Planner to reach a sequence of three frontiers as well as the actual traversal (using the SLAM pose) in Figure 12. In each sequence, the green dot-line is the planned path while the blue line is the SLAM pose.

3) Competition: The performance of our team's MBots in the four competition events, as well as the number of attempts, are specified in Table IX.

TABLE IX: Competition Outcomes

Event	Result	Attempts
1	Successful, High-Rank	2
2	Successful, No Autonomy, High-Rank	1
3	Successful Map, Okay Final Pose	5
4	Successful, No Autonomy, High-Rank	1

We asses how much Easing Factor (which starts at -10 cm and increases by 1 cm in each repetition) applied to the `minDistanceToObstacle` search parameter allows for the discovery of a path during various path-planning attempts in Event 3. We present statistics based on all the path-planning attempts during one successful mapping. This data is presented in Table X.

TABLE X: Easing Factors Statistics in Event 3 (in m)

	Easing Factor (EF)
$Min.$	-0.02 (Resulting Min. Distance $= 0.22$)
$Max.$	0 (Resulting Min. Distance $= 0.20$)
μ	-0.00559 (Resulting Min. Distance $= 0.20559$)
σ	0.00552

IV. Discussion

A. Motion Controller and Odometry

The calibration results reveal that there's more variation in the punch of our MBots' calibration function than the slope. The static friction that needs to be overcame isn't as consistent across calibrations, which makes sense, as the slope depends more on the mechanical structure of the motors. The area where a calibration is done could vary, and certain non-systematic factors like slippage or an uneven floor might affect the static friction.

The usage of IMU to estimate the MBot pose's θ value enhances the pose estimation compared to relying solely on the encoder readings; the odometry output after the change has readings that align with the waypoint locations from the simple maze. Furthermore, for results that rely on the Motion Controller, the dead-reckoning pose computed using the enhanced odometry is far more reflective of the four-square path that the MBot performed compared to the rudimentary odometry.

The three plots of a square-path traversal at different stages of the Motor Control tuning reflect the tuning process. Having no motor control produces a fairly accurate traversal and odometry output, and having the low-pass filters allows for smoother motion. However, it causes small swerves, and the final pose odometry is more displaced in x, y, θ. The plot that combines the filters with PID tuning reflects a very smooth and accurate traversal, and it has the most accurate odometry at the final pose.

B. SLAM

In Figure 8, the bolder map with `kHitOddsArg` = 5 and `kMissOddsArg` = 1 motivates our choice of those parameters. Regarding the action model uncertainty parameters, $k1 = 0.001$ suggests a low uncertainty in rotation, indicating that the robot's rotational movements are relatively precise and consistent. This aligns with our fairly accurate θ odometry readings. $k2 = 0.001$ implies

that the robot's linear motion is also quite accurate, with minimal straight deviation from the expected path.

The square path traversal with particles at various time intervals indicates that the particles stay close to the actual robot traversal, indicating a robust action and sensor model implementation. Visual inspection of the MBot during competition traversals aligns with the particle distributions we observe and resulting SLAM poses; there is never a case when the MBot hits an obstacle but the SLAM pose doesn't report it or vice-versa.

As we can see in Figure 10, the odometry path accumulates small divergence across the trajectory (especially on turns) compared to the SLAM pose, which results in an end position that is slightly off on a short trajectory. As a result, relying on SLAM allows for a much more accurate real-time localization. The SLAM pose vs ground-truth pose results indicate that the error is highest for y, while the SLAM pose is fairly accurate in θ, even though there is at least one point with a very high error in this component (it has the highest maximum error).

C. Path Planning and Exploration

When running tests on the final A-Star implementation, none of the successful test cases exceeded 0.1 seconds of execution time. In the failing cases, the maximum time sub-case for the narrow map has a run time far greater than the rest, but it is still within 1 second. Comparing this to the version without an Easing Factor, we observe that the average execution time is generally higher for the full implementation, but not by a magnitude of over 3. This is expected since the full implementation has more iterations. The difference in execution time between both implementations is lower for most of the successful cases (except the maze) when compared to the failing cases; even a strict distance requirement results in a path for the successful tests, except for the maze map. The sub-optimal implementation had significantly higher run times than the full implementation except for the convex map. The failing narrow map tests resulted in an average execution time of about 10 seconds when using a vector for the closed set, indicating a high number of candidate child nodes and corresponding checks for previous exploration. There are cases when the additional complexity from pushing to a set instead of a vector isn't offset by the faster lookup when finding a child node that an unordered set provides, and a concave map appears to be one such case.

The Easing Factor isn't a factor in the actual competition; for the path-planning between any two nodes in the Event 3 maze, there isn't ever more than one valid path.

We find that the exploration implementation is accurate when moving through the map to frontiers, but frequently fails to return to the home pose. This is likely due to inaccuracy in the odometry estimation of the robot that accumulates over time.

D. Improvements

Although our goal is for the forklift to be autonomous, that functionality is skipped due to time. Nevertheless, the forklift is very precise in the competition via manual control, and we score well in the relevant tasks without multiple attempts. An improvement that we'd prioritize is to achieve smoother MBot movement when traversing paths. One way is to implement a smarter pruning strategy, rather than simply removing each node within 5 map cells of the other (this can actually be problematic if the global distance represented within a cell change). Ideally, the pruning should remove all nodes along a path with no changes in θ. Another way to achieve smoother motion is to implement a Carrot-Following or Pure-Pursuit method for the motion controller. One more improvement is to try out different selections of the highest-weight particles for use in pose estimation; 15% is arbitrarily chosen. Finally, implementing a better IMU fusion mechanism using a complementary or Kalman filter and assessing the change in performance would be ideal. It's possible that purely relying on the IMU data isn't the best choice for all MBots, and further testing of this would help.

V. Conclusion

In conclusion, the proposed MBot implementation can autonomously explore and map an environment fairly accurately. We find that the closed loop motor controller and pre-set motion controller perform well, as the MBot can follow waypoints with precision and traverse complicated mazes without crashing into obstacles. Although we don't implement autonomy for forklift-related tasks, the mechanism itself is very sturdy and precise when picking and placing blocks. Overall, we believe that motor control, path planning, and map construction are the most reliable aspects of our implementation, while pose estimation, odometry, and the jerky style of motion have scope for improvement.

References

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

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