# ROB 550 Botlab Report

Lilly Mei, Ashwin Saxena, Ziyu Guo
{mlillian, ashwinsa, ziyuguo}@umich.edu
Team 6 (AM)
April 25, 2023

***Abstract*—In the Botlab project, the team explored strategies in action, perception, and reasoning in order to program a mobile robot to drive autonomously. Specific tasks included creating controllers for the motion of the robot programmed to a Pico controller board as well as mapping, localization, planning, and exploration which were executed by the Raspberry Pi 4 with the use of the 2D RPLidar. The project culminated in a competition with events involving mapping while driving, completing simultaneous localization and mapping (SLAM) as fast as possible, and path planning and exploring in a maze. The team was able to execute a majority of the tasks and learned a lot from this project.**

## I. Introduction

As society pushes for more efficiency, there has been a rise in mobile robotic technology in spaces such as self-driving cars, warehouse organization, and package delivery. These technologies rely on their ability to perceive their surroundings and travel safely to the destination. The ROB 550 Botlab project involved such mobile robotics concepts including motion control, odometry, simultaneous localization and mapping (SLAM), planning, and exploration. The equipment used in this project primarily consisted of a small two-wheeled robot with a rear caster called the 'MBot', a Raspberry Pi 4 onboard computer, a Pico controller board, and a 2D RPLidar. The MBot is shown in Fig. 1.

The project began with controlling the motion of the robot by first calibrating the motor encoders and PWM duty cycles and then performing odometry to convert encoder data to position and orientation. The next step was creating PID and body velocity controllers to make the bots drive to specified locations or 'waypoints' with the desired speed and positional accuracy. Following that, SLAM was accomplished by using the 2D lidar to map the surroundings as well as determine the bot's location in space, the latter of which also involved using a particle filter. Path planning was programmed using the A* algorithm. The team was also able to attain partially functional exploration capabilities. All of these functionalities were combined and showcased in the culminating Botlab competition that had events which involved creating a map while driving a predetermined set of waypoints, completing SLAM as fast as possible, and employing path planning and exploration in order to traverse and map a large unknown maze.

Fig. 1. The MBot, a two wheeled mobile robot with a rear caster, a Raspberry Pi, a Pico controller, and RPLidar that was used in the Botlab project.

## II. Methodology

### A. MBot Setup

Before we could start running any experiments on the MBot, we first got the Raspberry Pi set up with all the necessary network configurations and libraries. The mini SD cards inserted into the Pis were flashed with custom 32 GB image and this process took about 30 minutes. Since the SD cards corrupted at least 3 times, we had to repeat this process several times. Network configuration scripts were run to allow the MBot to automatically publish any IP address changes to a repository for easier SSH access, while NoMachine and Minicom were installed to allow for visualization of data and system performance.

### B. Odometry

*1)* ***Motor Encoder Calibration****:*
The first step in controlling the motion of the robots was to calibrate the motor encoders for each of the two wheels. This ensured that an appropriate PWM signal would be sent to the motors for a desired angular displacement. Each wheel was run in each direction for incrementally increasing duty magnitudes, and the speed in RPM was recorded. The calibration curve was obtained by plotting the speed against duty cycle for each wheel and finding the slope for each of the linear portions. The calibration results can be found in the Results section.

*2) **Odometry**:*
The odometry, or the position and orientation of the robot relative to its original configuration, was calculated from the wheel encoder readings. The equations used for odometry essentially used geometry, the width of the bot, and the wheel diameter to convert encoder data to position and orientation in real time, and are shown below. $\Delta s_L$ and $\Delta s_R$ represent the distance travelled by left and right wheel respectively. $\Delta\theta$ represents the change in robot's orientation while $\Delta x$ and $\Delta y$ represent change in robot's position.

$$\Delta\theta = (\Delta s_L - \Delta s_R)/b \tag{1}$$

$$\Delta s = (\Delta s_L + \Delta s_R)/2 \tag{2}$$

$$\Delta x = \Delta s * \cos(\theta + \Delta\theta/2) \tag{3}$$

$$\Delta y = \Delta s * \sin(\theta + \Delta\theta/2) \tag{4}$$

*3) **Gyrodometry**:*
Typically, MEMS Gyros will drift due to temperature dependent bias. Therefore, our odometry is based on readings from encoders. However, the noise of the heading sensor cannot be ignored when using encoder-based odometry since wheel slip may happen during turns. Gyrodometry [1] is a method for combining data from gyroscope and odometry (Encoder), which would theoretically improve the accuracy of odometry. Algorithm 1 briefly shows the whole process of gyrodometry.

**Algorithm 1** Gyrodometry

**Require:** $\Delta\theta_{gyro}, \Delta\theta_{enc}$
**Ensure:** $\theta_{odom}$
$\Delta_{diff} \leftarrow \Delta\theta_{gyro} - \Delta\theta_{enc}$
**if** $|\Delta_{diff}| > threshold$ **then**
  $\theta_{odom} \leftarrow \theta_{odom} + \Delta\theta_{gyro}$
**else**
  $\theta_{odom} \leftarrow \theta_{odom} + \Delta\theta_{enc}$
**end if**

Gyrodometry method is able to decide whether to trust the IMU data or the encoder data. If the difference between $\Delta\theta_{gyro}$ and $\Delta\theta_{enc}$ is larger than the preset value, the odometry will trust the IMU data. Therefore, by tuning the $threshold$, we can mitigate the bad effect of IMU drift and make our odometry more accurate.

### C. Motion Control

In order for the MBot to move to the desired pose with the desired accuracy and speed, we implemented multiple controllers to tune the motion of the bot. We first implemented open loop and PID controllers to tune the motion of each wheel after which we tuned the robot frame velocity controller.

*1) **Open Loop & PID Controller**:*
We created an open loop controller to control the wheel velocity such that it moves at desired speed. The open loop controller works by linearly mapping the input reference velocity to an output duty cycle. The mappings are determined by motor encoder calibration, which is explained in Part. B 1). However, due to the limitations of the open loop controller, and its inability to correct any errors, we used a PID controller to control the velocity. Figure 2 shows how the various components of a PID controller interact with each other. The controller gets velocity feedback from the motor encoders, and then the velocity error is used to get the new set point velocity. The new set point velocity is then converted to PWM input for the motors. The equation below shows how the new setpoint velocity is calculated.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{5}$$

Here, $u(t)$ is the set point velocity at time $t$, $e(t)$ is the error at time $t$, which is the difference between the set point and the target velocity. $K_p$ is the proportional gain, which determines the amount of control action based on the current error. $K_i$ is the integral gain, which determines the amount of control action based on the accumulated error over time. $K_d$ is the derivative gain, which determines the amount of control action based on the rate of change of the error with respect to time.

We implemented a PID controller for each of the wheels. We tuned the PID gains by driving the robot first in a straight line and then a square. $k_D$ was set to 0.0 because other values caused a very undesirable jerking. We found that increasing $k_P$ caused the robot to move too fast and sometimes overshoot. $k_I$ accounted for steady-state errors particularly in turning. The final PID gains are shown in Table I.

TABLE I
PID CONTROLLER PARAMETERS

| $k_P$ | $k_I$ | $k_D$ |
|---|---|---|
| 1.0 | 0.1 | 0.0 |

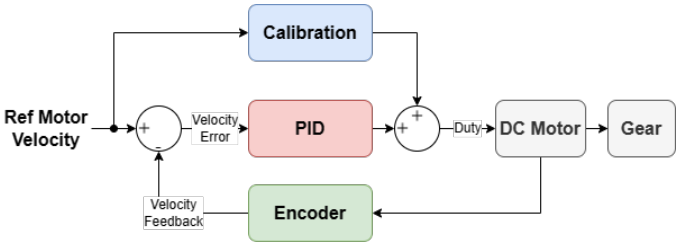


Fig. 2. Wheel PID Controller

*2) **Robot Frame Velocity Controller**:*
After tuning the individual wheel PID controllers, we implemented a robot frame velocity controller on top of the wheel controllers. The purpose of this velocity controller is take in linear and angular velocity set points and determine the left and right wheel velocity set points. The equations to calculate those set points are shown below.

$$v_{left} = v_{fwd} - \frac{radius_{wheel}}{2} \cdot v_{rot} \tag{6}$$

$$v_{right} = v_{fwd} + \frac{radius_{wheel}}{2} \cdot v_{rot} \tag{7}$$

*3) **Motion Controller**:*
Once the robot frame velocity controller was implemented, we implemented a motion controller to get the MBot to drive to a specific pose. The motion controller was designed for driving between known waypoints of type `pose_xyt_t`. The motion controller executes the trajectory using the aforementioned controllers until the robot reaches the final waypoint. The controller took feedback from odometry to drive the Rotate, Translate, Rotate (RTR) values, shown in Fig. 3, to zero in order to get from the current pose to the desired pose [2]. Then, the robot frame velocity controller sent wheel speed commands back to the wheel controllers.

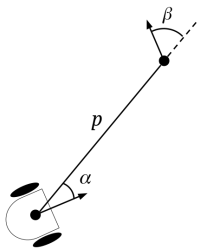


Fig. 3. The Rotate, Translate, Rotate (RTR) values that are driven to zero by the feedback controller.

Parameters $k_p$, $k_\alpha$, and $k_\beta$ were used in the control laws shown in Eq. 8 and 9.

$$v = K_p \cdot p \tag{8}$$

$$\omega = K_\alpha \cdot \alpha + K_\beta \cdot \beta \tag{9}$$

These parameters were then tuned until the MBot tracked desired poses properly. The final parameters are shown in Table II.

TABLE II
ROBOT FRAME VELOCITY CONTROLLER PARAMETERS

| $k_p$ | $k_\alpha$ | $k_\beta$ |
|---|---|---|
| 1.0 | 4.0 | -0.1 |

### D. SLAM

SLAM (Simultaneous Localization and Mapping) is a fundamental problem in robotics, which involves building a map of an unknown environment while simultaneously tracking the robot's position within that environment. SLAM requires the use of a mapper and a localization method. Figure 4 shows how the different components of SLAM interact with each other.

*1) **Mapping**:*
The mapping logic for SLAM was programmed by assigning a probability for each cell in the grid as the odds that the cell was occupied by some obstacle or free space. This was done by taking each laser ray and incrementing the log odds for cells when a ray ended prematurely or hit an obstacle and decrementing the log odds for all the cells in the ray if no obstacle was hit using Bresenham's algorithm. This mapping process is summarized in Algorithm 2. The mapping performance was verified by using provided log files and displaying them on the botgui which is a user interface that allows us to visualize items such as the map, the laser rays, the SLAM pose, the odometry, and other useful information.

**Algorithm 2** Mapping Algorithm

**for** $ray$ in $moving\_scan$ **do**
  Increase odds for endpoint of ray
  $Cells \leftarrow bresenham(ray)$
  **for** $cell$ in $Cells$ **do**
    Decrease odds for open cell
  **end for**
**end for**

*2) **Monte Carlo Localization**:*
The Monte Carlo Localization (MCL) algorithm was implemented to allow the MBot to determine its position in space using a particle filter. This involved using an action model that samples possible positions, a sensor model that determines the likelihood of a pose given lidar readings, and several particle filter functions [3].

In our action model, the particles or probability distribution were dispersed by modeling the action error as a rotation followed by a translation followed by a second rotation. Algorithm 3 shows how we calculated the updated $x, y$, and $\theta$. We modeled the error as Gaussian and manipulated the particle distribution using the parameters $k_1$ and $k_2$ [3]. We tuned these parameters by visualizing the particle distribution in the Botgui. The particles had to spread far enough to include the actual pose which is why chose a relatively large $k_1$ but not so large that the distribution was too wide. The parameter values that we found experimentally to be most suitable are shown in Table III. These values yielded a SLAM pose that was close to the true pose.

TABLE III
ACTION MODEL FINAL PARAMETER VALUES.

| $k_1$ | $k_2$ |
|---|---|
| 0.150 | 0.005 |

The sensor model found the probability that a lidar scan matched the hypothesis pose given a map. We use the simplified likelihood field model by looking at the endpoint of a ray. If the log odds of the endpoint were positive, we added that to the score of the scan [3]. Otherwise, we checked the cell before and after the endpoint along the ray and took a fraction of the log odds into the scan score.

In the particle filter functions, we resampled the particles using the low variance resampling method to prevent the problem of variance amplification. Then, we computed the proposal distribution using the action model. This was followed by normalizing the distribution and calculating the weighted average of the particles to output the final pose estimate.

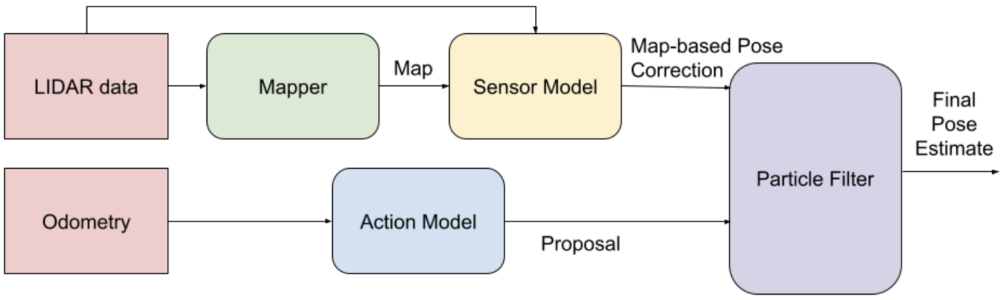


Fig. 4. SLAM architecture

**Algorithm 3** Odometry Action Model

Require $x_{t-1}, \hat{x}_t, \hat{x}_{t-1}$
$\Delta x \leftarrow \hat{x}_t - \hat{x}_{t-1}$
$\Delta y \leftarrow \hat{y}_t - \hat{y}_{t-1}$
$\Delta\theta \leftarrow \hat{\theta}_t - \hat{\theta}_{t-1}$
$\delta_{\text{rot1}} \leftarrow \text{atan2}(\Delta y, \Delta x) - \hat{\theta}t - 1$
$\delta_{\text{trans}} \leftarrow \sqrt{\Delta x^2 + \Delta y^2}$
$\delta_{\text{rot2}} \leftarrow \Delta\theta - \delta_{\text{rot1}}$
$\hat{\delta}_{\text{rot1}} \leftarrow \text{gaussian}(\delta_{\text{rot1}}, k_1\delta_{\text{rot1}})$
$\hat{\delta}_{\text{trans}} \leftarrow \text{gaussian}(\delta_{\text{trans}}, k_2\delta_{\text{trans}})$
$\hat{\delta}_{\text{rot2}} \leftarrow \text{gaussian}(\delta_{\text{rot2}}, k_1\delta_{\text{rot2}})$
$x_t \leftarrow x_{t-1} + \hat{\delta}_{\text{trans}} \cos(\theta_{t-1} + \hat{\delta}_{\text{rot1}})$
$y_t \leftarrow y_{t-1} + \hat{\delta}_{\text{trans}} \sin(\theta_{t-1} + \hat{\delta}_{\text{rot1}})$
$\theta_t \leftarrow \theta_{t-1} + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

### *E. Planning and Exploration*

*1)* ***Obstacle Distance****:*
In order to ensure that the MBot did not collide with or graze any obstacles, we had to determine how far obstacles were from the bot. This was accomplished by using the Brushfire Algorithm to grow the obstacles in the map so that the MBot would not drive to any location where its distance to an obstacle was less than the bot's radius.

*2)* ***Planning****:*
In order to plan a path for the bot to traverse, the team implemented the A* algorithm. The A* algorithm combines the efficiency of the Greedy algorithm and the shortest path from Dijkstra's algorithm in order to quickly find the actual shortest path from start to goal [3].

*3)* ***Exploration****:*
The team added basic exploration functionality to the bot which allowed it to start in an arbitrary location of an arbitrary environment and travel to every frontier or unexplored region. This was accomplished by first identifying frontiers and sorting them by proximity. Frontiers are identified by search for unobserved cells which are next to observed unobstructed cells. Then, the bot used A* to plan a path to each frontier, and the robot starts moving to the nearest frontier center, which is selected as the exploration target. New frontiers keep getting added and visited until no frontiers remain and the environment is fully explored.

*4)* ***Kidnapped Robot Problem****:*
Since we ran out of time, we were unable to implement an algorithm to solve the Kidnapped Robot Problem for event 4. However, we have all the pieces required to eventually implement the solution. Using the exploration mode, a map of the maze would be first created. Then a particle filter would be initialized with a set of random particles that are uniformly distributed across the map with each particle representing a possible pose of the robot. The sensor model would then be used to update the likelihood of each particle being the robot's current pose. Particles would then be re-sampled based on their weights. The robot would move towards the goal position. We would repeat this process until the particle filter converges on the robot's actual pose. Once that happens, the A* path planning algorithm can then be used to plan the way out of the maze.

## III. Results

### *A. Motion & Odometry*

*1)* ***Motor Encoder Calibration****:*
The team performed motor encoder calibrations for all three bots, and the left and right wheel calibration curves are shown in figures 5 and 6. The fitted curves for the linear parts are shown for Bot 1 as an example.

An example of the calibration equations obtained from the calibration curves are shown in equations 10 to 13 for Bot 1. These are the left wheel reverse, left wheel forward, right wheel reverse, and right wheel forward duty cycle calibration equations, respectively. $d$ represents duty cycle out of 1, and $s$ is motor speed in RPM.

$$s = 282 \cdot d + 31.7 \tag{10}$$

$$s = 265 \cdot d - 28.2 \tag{11}$$

$$s = 270 \cdot d + 33.1 \tag{12}$$

$$s = 263 \cdot d - 27.8 \tag{13}$$

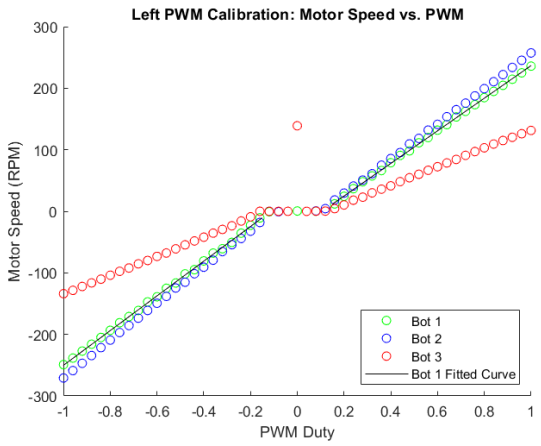


Fig. 5. The left wheel calibration curves for all three bots.

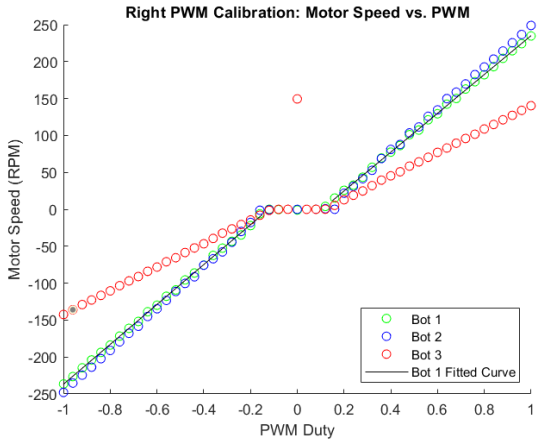


Fig. 6. The right wheel calibration curves for all three bots.

*2) **Odometry**:*

In order to validate our odometry model, we compared the odometry readings to the actual distance traveled or angle rotated by the MBot. For a 1 m straight line distance, the odometry reading was 1.045 m meaning that the error was 4.5 cm per meter. For a $\frac{\pi}{2}$ or 1.571 rad rotation, the odometry read a change in theta of 1.590 rad. This means the angle error was about 0.076 rad per full $2\pi$ rotation. Our team did not end up applying correction parameters since our bot was able to traverse the simple maze for the first checkpoint assignment mostly accurately. Given more time to complete the checkpoint, we would have set correction parameters based on the measured errors to remove systematic odometry errors.

*3) **Gyrodometry**:*

Referencing variables from Algorithm 1, Borenstein et. al. recommended setting the $threshold$ to $0.125°$, and here we also set $threshold = 0.125°$ [1], which empirically performed well enough. For other related parameters, we set `WHEEL_RADIUS` = 0.045 and `WHEEL_BASE` = 0.16, which are actual measured values.

*4) **Velocity Controller**:*

To test the accuracy of the velocity frame controller, the robot x position and robot frame heading vs time were plotted for three different velocity set points for varying amounts of time. As can be seen in figure 7, as the target velocity increases, the robot reaches the position in lower time. This is consistent with the expected behavior for the robot. Similarly, figure 8 shows the the robot heading changing daster as the setpoint angular velocity increases. Since the $\theta$ values can only range between $-\pi$ and $\pi$, when the robot hits $\pi$ radians it suddenly goes back down to $-\pi$. This is also consistent with the expected behavior for the robot.

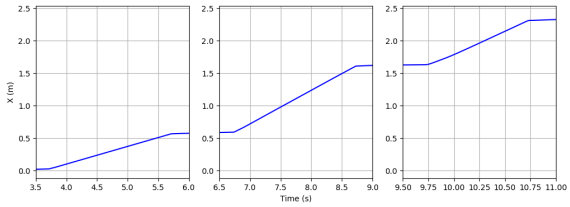


Fig. 7. Robot position (in x direction) vs time for 0.25 m/s for 2s, 0.5 m/s for 2s, 1 m/s for 1s

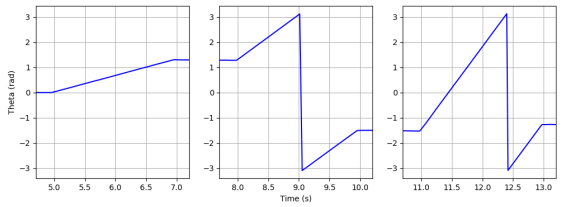


Fig. 8. Robot orientation vs time for $\pi/8$ rad/s for 2s, $\pi/2$ rad/s for 2s, $\pi$ rad/s for 2s

*5) **Motion Controller**:*

Before we tuned our motor controller, the robot's dead reckoning estimated pose was first plotted as shown in figure 9. From the figure, we can see there is a little drift in the estimated pose after every round of a square, and the corners are not perfect. This illustrates the need for a more sophisticated localization method like the Monte Carlo localization method discussed earlier.

The accuracy of the motion controller was tested by making the robot drive in a square path and graphing the angular and forward velocity as shown in figure 10. The maximum velocity for the motion controller was set to 0.3 m/s for a more precise and accurate motion control. The graph has four peaks around 0.3 m/s which correspond to the robot driving straight on the straight leg of the square. When the forward velocity is about 0, the angular velocity is around 1 rad/s which corresponds to the robot rotating 90 degrees. This motion profile is consistent for a robot driving square and made us confident about the performance of our motion controller.

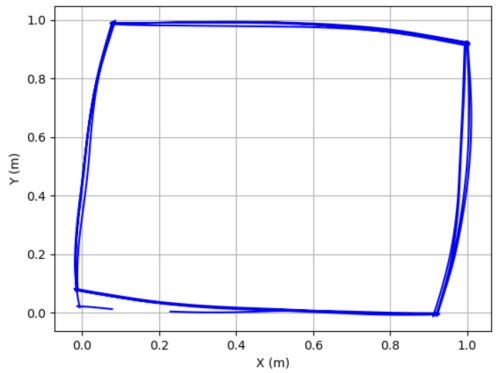


Fig. 9. Robot's dead reckoning estimated pose as the robot is commanded to drive a 1m square 4 times

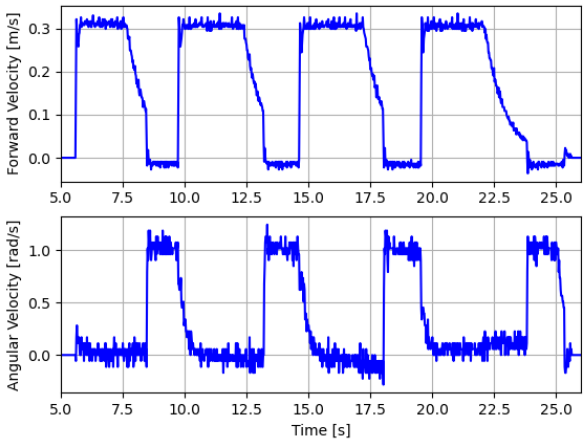


Fig. 10. Robot linear and angular velocity as it drives one loop around the square.

### B. SLAM

*1) **Mapping**:*

The mapping algorithm we wrote produced clean occupied and unoccupied regions. This is illustrated in figure 11 where a log file of a convex space with internal obstacles was used to generate a map with clean white open spaces and clean black boundaries against the obstacles or boundaries of the space.

*2) **Localization**:*

Our localization algorithms allowed the MBot to follow the true pose fairly closely. Table IV shows the update time for different particle quantities.

In order to estimate the time it takes for the particle filter to update, the time required for particle filter to complete one round around the `obstacle_slam_10mx10m_5cm.log` was calculated and then divided by total number of particle filter updates. Results are shown in Table IV

Based on the time required to update the particles, we estimate that the maximum number of particles our filter could support while the RPi is running at 10 Hz would be around 700 particles. Fig. 12 shows the particle distribution for 300 particles using the provided log file `drive_square_10mx10m_5cm.log`.

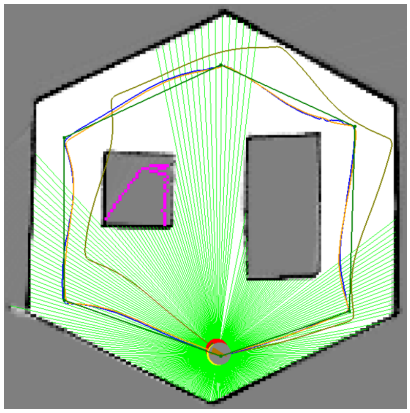

Fig. 11. For the obstacle log file, our mapping algorithm produced this map which has clean white open spaces and clean black boundaries.

TABLE IV
PARTICLE FILTER UPDATE TIME IN $\mu$S FOR DIFFERENT PARTICLE QUANTITIES.

| Number of Particles | Update Time ($\mu$s) |
|---|---|
| 100 | 19,428 |
| 300 | 49,872 |
| 500 | 82,345 |
| 1000 | 159,243 |
| 2000 | 297,893 |

Figure 13 shows the error over time between the SLAM pose and odometry for $x, y$, and $\theta$. Initially, there is very little error but the error grows over time, until it starts converge back as the robot finishes it round around the map.

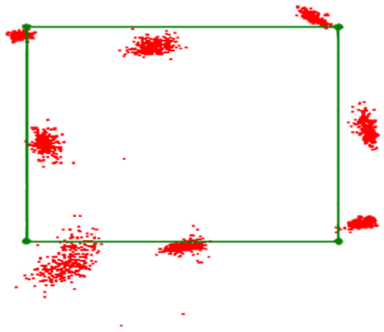

Fig. 12. These are the particle distributions at eight locations around a square log file path for 300 particles.

### C. Planning & Exploration

The planning ability of the robot was validated by running tests from the provided `astar_test.cpp` file. There were six tests with different maps, each with several start and goal positions. If the bot found the correct path or identified correctly that there was no path for each case, that test was

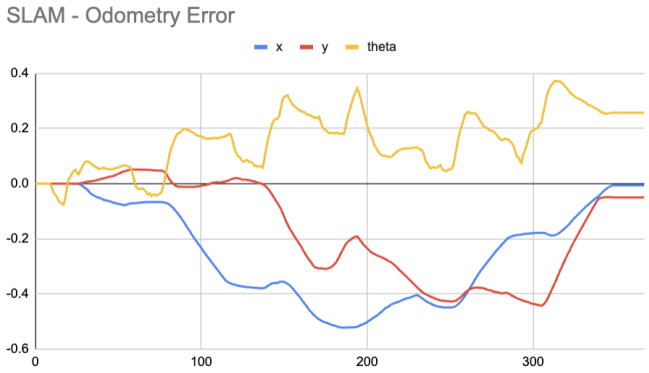


Fig. 13. SLAM-Odometry pose error over time. The error in $x$ is shown in blue, $y$ in red, and $\theta$ in yellow.

passed. Our bot passed three of the tests and failed the other three with the failed tests mostly being due to the bot finding paths that were considered unsafe or too close to obstacles. The mean path planning times for both successful and failed test cases are shown in Table V.

TABLE V
TESTING PERFORMANCE USING ASTAR_TEST.

| Test Grid Type | Successful Mean Time ($\mu$s) | Failed Mean Time ($\mu$s) |
|---|---|---|
| Empty | N/a | 42.8 |
| Filled | 295.8 | N/a |
| Narrow Constriction | 3269.67 | 25 |
| Wide Constriction | 4136 | N/a |
| Convex | 67 | N/a |
| Maze | N/a | 20044 |

By using the A* algorithm, the team was able to execute click-to-move in the botgui. We created a simple environment to demonstrate that our bot planned the most efficient path to the other end of the environment and followed that path. This is shown in Fig. 14.

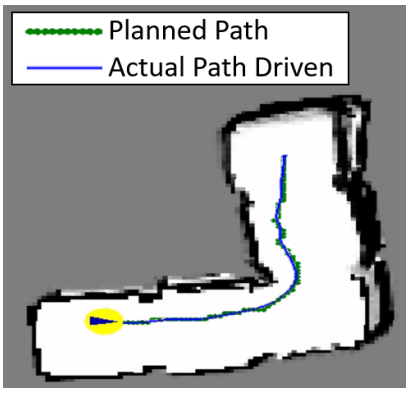


Fig. 14. This map shows a planned path in an environment of our creation with the actual path overlaid on top.

## IV. Discussion

### A. Motion & Odometry

There was definitely room for improvement with the motion and odometry of our bots. Given more time, we would save and use each bot's unique motor encoder calibration. Additionally, we would apply a correction factor for the odometry since the odometry wasn't quite accurate.

### B. SLAM

The mapping algorithm proved to be quite successful as the bot was able to repeatedly produce clean maps. Figure 15 shows a plot of the SLAM pose compared to the true pose and the optometry. With 500 particles, we are able to localize the robot pretty close to the true pose. The RMS (Root Mean Square) error between the true pose and SLAM pose was 0.2361, signifying that the two poses were fairly close. However, the code for the action model, sensor model, and particle filter could be further tuned so that the SLAM pose identically matches the actual pose of the bot.

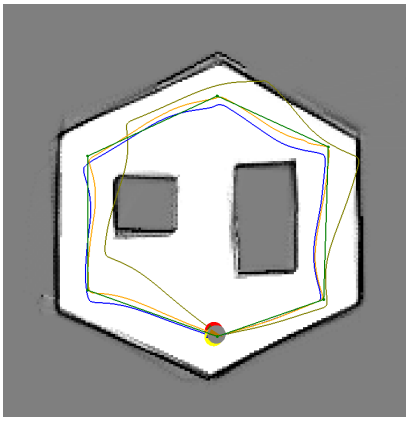

Fig. 15. SLAM pose shown in blue compared to true pose and odometry in yellow and green, respectively. The SLAM pose was able to follow the true pose somewhat closely.

### C. Planning & Exploration

*1)* ***Planning****:*
The A* algorithm used in path planning passed 3 of the 6 tests, and the time required to find the paths could be improved. The performance of the bot's planner could be improved by eliminating paths that are unsafe or too close to obstacles. This is likely a matter of increasing the value being used as the robot's radius since our obstacle distance detecting code (`obstacle_distance_grid`) passed all of its tests.

*2)* ***Exploration****:*
While the team was able to implement basic exploration functionality, it didn't always work perfectly. Further work would be needed to ensure that the robot could start at any place in an unknown environment and fully explore all of the frontiers. Additionally, the team did not have time to actualize the bot's ability to take a provided map and determine where it was in the map. This would require debugging the code to initialize particles uniformly across the unoccupied space

of the grid and writing an algorithm to allow the bot to then figure out its position in the map.

### D. Botlab Competition Analysis

The Botlab competition served as a culminating final test of the overall system as the competition events involved several of the functionalities in conjunction. The team achieved the 4th highest score and attempted 3 of the 4 competition events, earning points in all three. This performance demonstrated great growth in a team with relatively low programming experience and robotics knowledge.

*1) **Event 1**:*

For Event 1, we used the `drive_square` file to drive in the square arena. During the competition, only 1 out of the 3 MBots was actually driving in a perfect square. Nevertheless, we were able to generate a good map of the arena as shown in Fig. 16, and our robot was pretty close to the starting pose. The outcome demonstrates that our mapping capability was very strong, but the motion and odometry could use some adjustment.

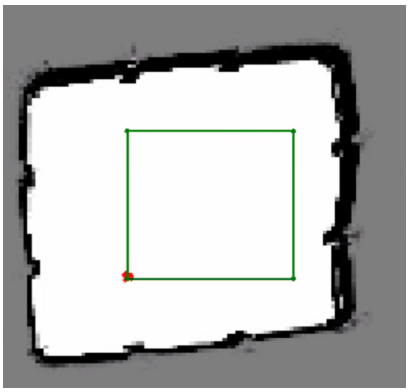

Fig. 16. Map produced for Event 1

*2) **Event 2**:*

For Event 2, instead of using our exploration algorithm, we decide to make a new `drive_event2` file that would use checkpoints to drive through the maze. This would allow us to map the maze much faster than using pure exploration. We obtained a pretty decent map however, on the way back, the MBot kept bumping into the wall, resulting in the diagonal phantom wall in Fig. 17. Otherwise, the map was quite clean, and the MBot was able to traverse the full motion in under 40 seconds which shows that our controllers were relatively successful.

*3) **Event 3**:*

For Event 3, even though we had a basic exploration algorithm coded in, we decided to move the robot via the click-to-move method since it was more reliable and our attempts were limited in number by the popularity of this competition event. Initially we obtained a good map, however there was some drift along the way which caused to map to become somewhat skewed as the MBot returned to its starting position as shown in Fig. 18. This outcome demonstrated that the motion control, mapping, localization, and path planning were working well enough together to manually explore a large maze and map it.

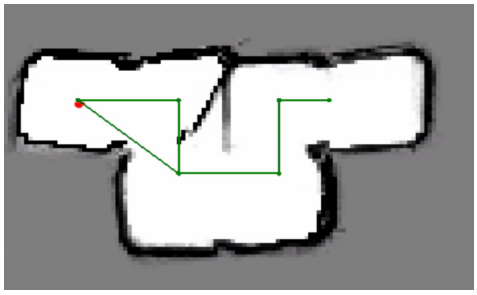

Fig. 17. Map produced for Event 2

Fig. 18. Map produced for Event 3

## V. Conclusion

In the Botlab project, our team was able to learn and implement robotics concepts in action, perception, and reasoning, all of which are essential elements of cutting edge autonomous vehicle and mobile robot technologies. Specifically, the the team integrated odometry, controllers, mapping, localization, path planning, and exploration capabilities on the MBot. Doing so resulted in the bot being able to traverse given paths, map the surroundings, perform SLAM, traverse efficient paths, and explore frontiers. Given more time, future work would include further tuning of parameters and algorithms as well as implementing the logic for the kidnapped robot problem. The team was got the 4th highest score in the Botlab competition. These results and the project as a whole demonstrated the team's growth in knowledge of mobile robotics and programming.

## 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] R. Siegwart and N. Illah R., *Introduction to Autonomous Mobile Robots*. Cambridge, MA: MIT Press, 2004.

[3] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Cambridge, MA: MIT Press, 2006.