Team 12 ROB 550 BotLab Report

Nicole Campbell, Niyanta Mehra, Xingqiao Zhu

***Abstract*—Autonomous mobile robots have various applications ranging from autonomous vacuums to self-driving cars. This report explores foundations of many important components of autonomous mobile robots; implementations of odometry, simulatenous localization and mapping (SLAM), and A* path planning will be demonstrated with a non-holonomic mobile robot equipped with a BeagleBone, RaspberryPi, 3-axis IMU, and 2D LiDAR. It was found that combining gyroscope data in addition to wheel odometry is needed to get more accurate readings at higher speeds as well as during turns. However, odometry-gyroscope fused pose estimates are far less accurate than SLAM pose estimates overall due to a smaller propagation of error. Although giving pre-determined waypoints works fairly well, utilizing A* path planning significantly decreased instances of bumping into obstacles due to the considerations of obstacle distances.**

I. Introduction

Given a non-holonomic robot equipped with a Beaglebone, RaspberryPi, 3-axis IMU, and 2D LiDAR, the goal of this project was to design and implement methods of accurately moving from one point to another in a 2D physical environment. The project consists of three parts: motion and odometry, simultaneous localization and mapping, and path planning and exploration. Each section builds off of the previous to optimize the information of the robot and its environment. The final conglomeration of algorithms and controllers has the capability to explore and navigate an unknown environment.

II. Methodology

A. Motion and Odometry

1) Characterize Wheel Speed for a Open Loop Controller: To calibrate the MBot, the steady state wheel speed and input PWM was measured and plotted against one another to determine a mathmatical relationship between the two parameters. This was performed on concrete floor to avoid frictional forces.

2) Wheel Speed Open Loop PID Controller: The wheel speed controller that was implemented was a PID controller with the feed forward calibration and a low-pass filter on the generated motor velocity. In our case, these parameters were sufficient to get the MBot to drive in an accurate 1m x 1m square. To calculate the proportional, integral, and derivative gain constants of the PID controller, the following equations were used:

$$K_P = 0.6 * K_u$$

$$K_I = 2.0 * K_p / P_u$$

$$K_D = K_P * P_u / 8$$

Using K_u as the K_p gain at which the step response graph dramatically oscillates and P_u as the period between the oscillations observed at K_u.

3) Odometry: Robot poses were estimated with odometry which used the following motion equations:

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{wheel\ base}$$

$$\Delta d = \frac{\Delta s_R + \Delta s_L}{2}$$

$$\Delta x = \Delta d * cos(\theta + \Delta\theta/2)$$

$$\Delta y = \Delta d * cos(\theta + \Delta\theta/2)$$

$$\Delta\phi_{L/R} = 2\pi * \frac{\Delta(encoder\ reading)}{(encoder\ resolution)(gear\ ratio)}$$

$$\Delta s_{L/R} = (wheel\ radius) * \Delta\phi_{L/R}$$

Where for our particular robot, the wheel base was 15.62 cm, the wheel diameter was 8 cm, and the encoder resolution was 20. For these calculations, all values were calculated in meters.

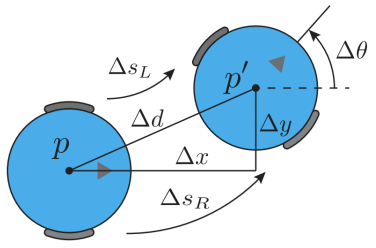

Fig. 1: Geometric Relations for Odometry

The odometry code was validated by manually pushing the MBot one meter straight and turning at a 90 degree angle. Although there was a small error, it was negligible and can be accounted for with some noise since pushing the MBot manually was not perfectly straight nor an exact 90º turn. No correction parameters were added since the performance was adequate.

4) Gyro Sensor Fusion: After fusing the odometry with the gyroscope readings of yaw (heading angle), we were able to decrease the error in the theta reading for a 90º turn. The following pseudo-code logic was used to fuse the two sensors:

$$if(\Delta_{gyro-odo} > \Delta\theta_{threshold})$$
$$then: \theta_i = \theta_{i-1} + \Delta\theta_{gyro,i} * T$$
$$else: \theta_i = \theta_{i-1} + \Delta\theta_{odo,i} * T$$

where T is the time step and $\Delta_{gyro_odo} = \Delta\theta_{gyro} - \Delta\theta_{odo}$ We roughly tuned threshold by increasing and decreasing the value until the MBark mapping accurately represented the robot's driving path. We opted for a higher threshold because we wanted to use the odometry data most of the time since the gyroscope data tended to have a lot of noise, specifically from temperature variations.

5) Robot Frame Velocity Controller: We incorporated another PID controller that acts as a robot frame velocity controller. Here, rather than feeding the left and right velocity it takes as input forward and turn velocities. We use these velocities to calculate left and right velocities and the corresponding errors to ultimately give these as command to the motors. Rather than calculating the left and right errors using set points we use the modified outputs of the velocity controller.The PID is tuned in a manner similar to the wheel speed controller. The final controller design is shown in Figure 2.

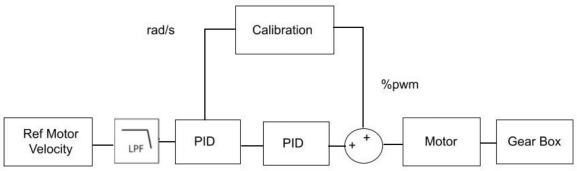

Fig. 2: Block Diagram of MobileBot Controller Design

6) Motion Controller: In order to get the MBot to follow waypoint trajectories, an Rotation Translation Rotation (RTR) controller was designed to have two states: driving and turning. In the driving state the MBot has the forward velocity, v, and the angular velocity, ω until the distance, d, is close to zero (as defined in a distance threshold). In the turning state, the forward velocity is zero and the turning velocity until alpha is close to zero (as defined in an angular threshold). The equations that describe these parameters are as follows:

$$v = K_v * d$$
$$\omega = K_\alpha * \alpha + K_\beta * \beta$$
$$d = sqrt(\Delta x^2 + \Delta y^2)$$
$$\alpha = atan^2(\Delta y, \Delta x) - \theta$$

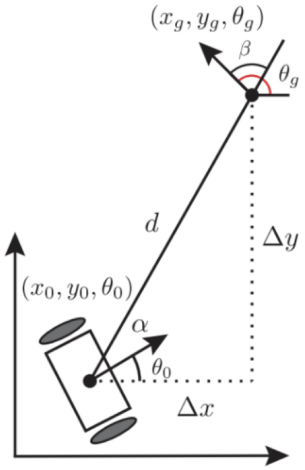

Fig. 3: Geometric View of Trajectory Between Waypoints

$$\beta = \theta_g - \theta$$

At fast speeds ($> 0.5m/s$), the MBot would lose balance from the sudden changes in velocities (e.g. when turning); to solve this, a velocity damping coefficient of 0.7 was used after the MBot was within 0.3 m of its target pose. Additionally, at higher speeds, it was more difficult for the robot to perfect achieve the way points, often overshooting, and so the distance and angular thresholds were increased to allow for more error on achieving each waypoint.

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: The ground-truth poses in the provided log files were used to construct occupancy grid maps. By interpolating the poses from odometry, we get the poses which match the timestamps of the LIDAR scan. Then we find the endpoints of each ray and complete 4 functions to compute new log odds for the cells which the ray touches and passes through. We use Breshenham's line algorithm to find the cells along the ray.

If the laser ray terminates at the cell C_{ij} at distance D:

$$log_2\lambda = log_2\frac{p(z = D|ooc(i,j))}{p(z = D|\neg ooc(i,j))} > 0$$

If the laser ray passes through cell C_{ij}:

$$log_2\lambda = log_2\frac{p(z = D|ooc(i,j))}{p(z = D|\neg ooc(i,j))} < 0$$

2) Monte Carlo Localization:

a) Action Model: The action model used the current and previous time step's odometry data to calculate the movement of the robot for each time step. From these, translation and rotation equations and their standard deviations were calculated as follows:

$$\delta_{trans} = \sqrt{\Delta x^2 + \Delta y^2}$$

$$\delta_{rot1} = atan^2(\Delta x, \Delta y) - \Delta\theta \; for \; [-\pi, \pi]$$

$$\delta_{rot2} = abs(\Delta\theta - \delta_{rot1}) \; for \; [-\pi, \pi]$$

$$trans_{std} = \sqrt{k_2 * abs(\delta_{trans})}$$

$$rot1_{std} = \sqrt{k_1 * abs(\delta_{rot1})}$$

$$rot2_{std} = \sqrt{k_1 * abs(\delta_{rot2})}$$

These were later used to calculate normal distributions to estimate the movement of each particle in the particle distribution as follows:

$$x_{moved} = x + trans_{sample} * cos(\theta + N(rot1, rot1_{std}))$$

$$y_{moved} = y + trans_{sample} * sin(\theta + N(rot1, rot1_{std}))$$

$$\theta_{moved} = \theta + N(rot1, rot1_{std}) + N(rot2, rot2_{std})$$

The uncertainty parameters, k_1 and k_2, were tuned by increasing and decreasing this value between 0 and 1 until the particle distribution was spherical. If the particle distribution was too spread out along the rotational axis, the k_1 term would be decreased whereas if too spread out along the translational axis, the k_2 term would be decreased.

b) Sensor Model: A sensor model was implemented to use the current LiDAR scan of the local position and compare it to the global mapping of the environment to determine the probability of the robot being in a given position in the map. This was calculated using the log likelihood of the endpoint of each ray in the LiDAR scan. If the log odds calculated was positive, it was added to the scan score, if not it was ignored.

c) Particle Filter: At every time step, if the robot has moved, the particle filter will go through the following procedure:

1) Resample from the posterior distribution
2) Compute a proposal distribution using the prior distribution
3) Computer a normalized posterior distribution using the proposal distribution, laser scan, and mapping
4) Estimate a posterior pose using the normalized posterior distribution

At the initial time step, all the particles are weighted equally as 1/N where N is the number of particles. For particle resampling, the low variance resampling method was implemented. The aim is to remove particles that may not strongly represent the system. This method was chosen as it is robust and easy to understand and implement. The algorithm works to favor particles with higher weights.

3) Simultaneous Localization and Mapping (SLAM): Combining all the components of SLAM, the integration can be seen visually in the block diagram in Figure 4.

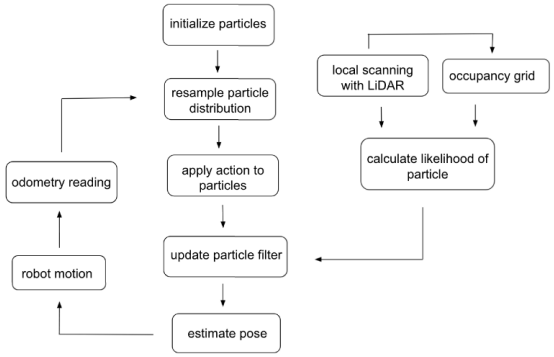

Fig. 4: Block Diagram of SLAM Components

To compare the estimated poses from the SLAM system against the ground-truth poses in `obstacle_slam_10mx10m_5cm.log`, root mean square error (RMSE) values were calculated for x, y, and θ.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} x_i^2}{N}}$$

C. Implementing Planning and Exploration

1) Obstacle Distance Grid: The obstacle distance grid, as the name implies, represents the distances to obstacles for all cells in the map where each cell is denoted as a 5cm square. The algorithm traverses unexplored cells and calculates the log odds. If the log odds are positive, it is denoted as an obstacle. The distance to the nearest obstacle is then calculated and stored in the obstacle distance grid which uses an 8-neighbor expansion of each cell.

2) A Path Planning:* The path planning algorithm implemented was A* which utilizes a priority queue and list to keep track of unexplored and explored cells, respectively, in a given map. The priority queue uses the f cost which takes into account both the distance from the starting cell (g cost) as well as the distance to the goal cell (h cost) relative to the current cell which is shown in the equations below.

$$\Delta x = abs(x_{goal} - x_{curr})$$

$$\Delta y = abs(y_{goal} - y_{curr})$$

$$h_{cost} = \Delta x + \Delta y + (\sqrt{2} - 1.0) * min(\Delta x, \Delta y)$$

$$g_{cost} = g_{cost,curr} + 1$$

$$f_{cost} = g_{cost} + h_{cost}$$

Our implementation uses diagonal distance metrics with eight neighbor cells when determining paths. Direction paths are calculated as $\sqrt{2}$ for diagonal movements and 1.0 for straight movements.

The logic of our algorithm follows that shown in the pseudo-code outlined in Algorithm 1.

Algorithm 1: A*

Input: start, $goal(n)$, $h(n)$, $expand(n)$
Output: path

```
if goal(start) = true then return makePath(start)

open ← start
closed ← ∅
while open ≠ ∅ do
    sort(open)
    n ← open.pop()
    kids ← expand(n)
    forall the kid ∈ kids do
        kid.f ← (n.g + 1) + h(kid)
        if goal(kid) = true then return makePath(kid)
        if kid ∩ closed = ∅ then open ← kid
    closed ← n
return ∅
```

3) Map Exploration: To implement Map exploration, we are given a list of frontiers, which we need to explore in order to drive autonomously. The logic used to pick a frontier is to check all cells in all frontiers and pick the cell which is the closest to the robot position.Squared Euclidean distance is used to find distance between the cell and robot position. Since, it could be possible that the robot cannot make a path to the chosen destination, we have implemented a logic wherein the destination is not a single point but a target region. The destination can be any point in that target region. The target region is chosen to be a small square (0.1*0.1 units) around the cell selected by minimum distance approach.

4) Map Localization with Unknown Starting Position: Given a pre-made map of the environment, the robot is placed in an unknown position in the environment where the goal is to localize itself in the map. The method to achieve this is similar to how the particle filter determines it's location in the map by creating normal distributions of particles around the local position and utilizing likelihoods of the LiDAR scans. In this implementation, however, instead of creating a particle distribution around the known starting position as we did previously, the particle distribution will be distributed across the entirety of the map. One way to optimize this method would be to use an adaptive sampling algorithm where there is a larger amount of particles at the initial particle distribution (especially the unknown starting point) where there is higher uncertainty in the location, and then decrease the number of generated particles as you move in the environment and the model has a better gauge of its localization in order to save computational time.

III. Results

A. Motion and Odometry

1) Characterize Wheel Speed for a Open Loop Controller: The relationship between input PWM and wheel speed is shown in Figure 5 and mathematically related in the following two equations for each wheel:

$$v_{right}[m/s] = 44.075490 * PWM - 2.386845$$

$$v_{left}[m/s] = 51.455893 * PWM - 1.912590$$

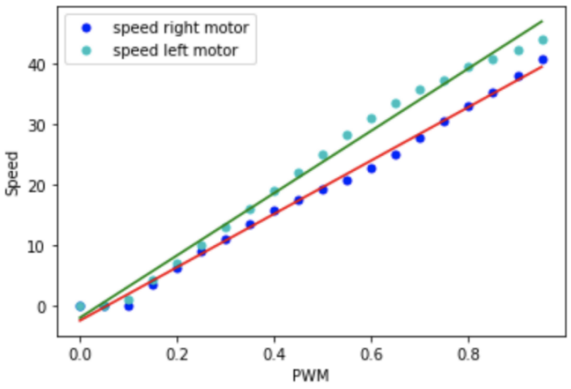

Fig. 5: Calibration Graph of Speed (m/s) v.s. PWM

2) Wheel Speed Open Loop PID Controller: The PID on the wheel speed has the tuned gains tabulated in Table I and the step response is shown in Figure 6.

TABLE I: Parameters for PID Controller

Tuning Parameters	Optimal Value
Oscillation Gain (K_u)	3.32
Proportional Gain (K_p)	1.992
Integral Gain (K_I)	41.08
Derivative Gain (K_D)	0.024

3) Odometry: The expected and measured values of x and theta during a 90 degree turn of the robot is tabulated in Table II.

TABLE II: Measured v.s. Expected Odometry Readings

Parameters	Expected	Measured	Error
Translation (x)	1.00 m	1.004 m	0.004 m
Rotation (θ)	1.571 rad	1.564 rad	0.007 rad

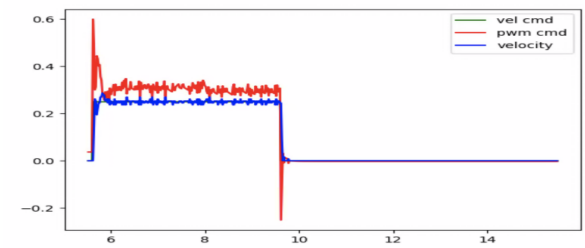

Fig. 6: Amplitude (PWM) versus Time (seconds) for Wheel Velocity PID

4) Gyro Sensor Fusion: The expected and measured values of x and theta during a 90 degree turn with the theta threshold below to integrate the gyro is tabulated in Table III.

$$\Delta\theta_{threshold} = 0.04363 radians$$

TABLE III: Measured v.s. Expected Gyro-Odometry Readings

Parameters	Expected	Measured	Error
Rotation (θ)	1.571 rad	1.567 rad	0.004 rad

5) Robot Frame Velocity Controller: The robot frame velocity controller step response is shown in Figure 7 and the tuned gains are shown in Table IV.

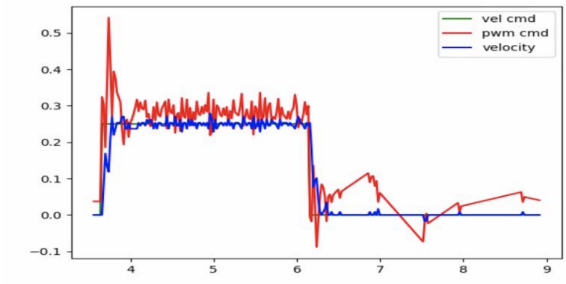

Fig. 7: Amplitude (PWM) versus Time (seconds) for Robot Frame Velocity PID

TABLE IV: PID Controller Gains for Robot Frame

Tuning Parameters	Optimal Value
Oscillation Gain (K_u)	0.8
Proportional Gain (K_p)	0.48
Integral Gain (K_I)	15
Derivative Gain (K_D)	0.003

Plots of the robot frame x position(in meters) versus time (in seconds) for 0.25 m/s for 2 s, 0.5 m/s for 2 s, 1 m/s for 1 s are shown in Figure 8, Figure 9, and Figure 10, respectively.

Plots of the robot frame heading angle (in radians) versus time (in seconds) for pi/8 rad/s for 2s, pi/2 rad/s for 2s, pi rad/s for 2s are shown in Figure 11, Figure 12, and Figure 13, respectively.

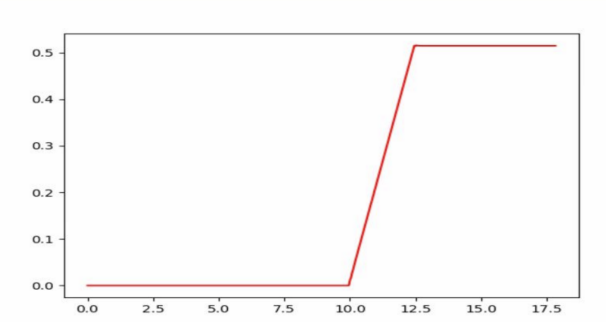

Fig. 8: Robot Frame x Position (meters) versus Time (seconds) for 0.25m/s for 2s

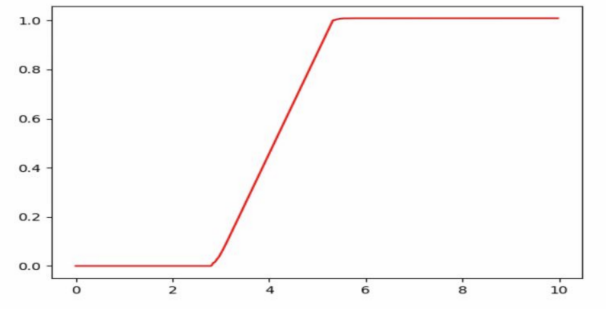

Fig. 9: Robot Frame x Position (meters) versus Time (seconds) for 0.5m/s for 2s

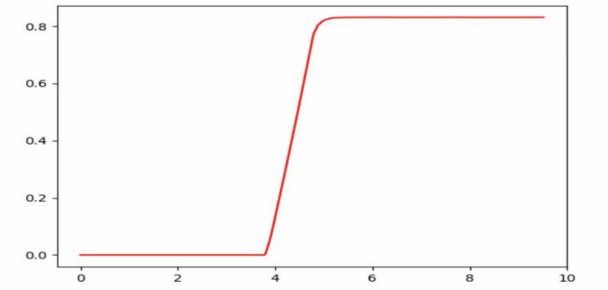

Fig. 10: Robot Frame x Position (meters) versus Time (seconds) for 1.0 m/s for 1s

6) Motion Controller: The tuned gains and thresholds for the RTR controller are tabulated in Table V for slow speeds, fast speeds, and for the 1m square.

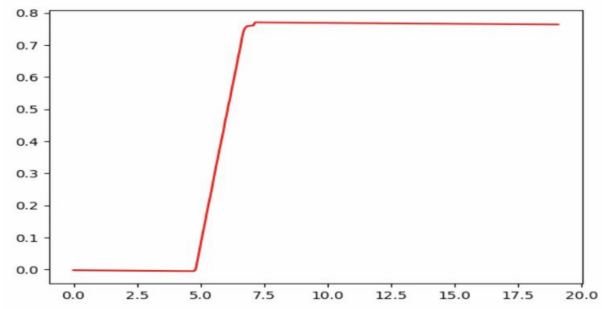

Fig. 11: Robot Frame Heading (radians) versus Time (seconds) for pi/8 rad/s for 2s

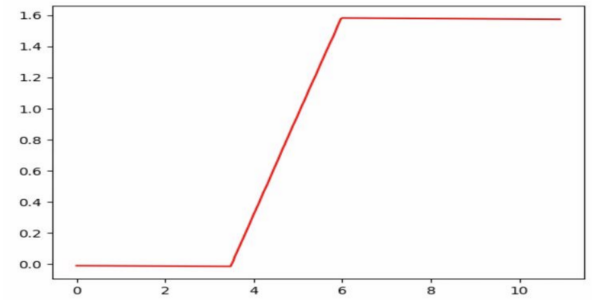

Fig. 12: Robot Frame Heading (radians) versus Time (seconds) for pi/2 rad/s for 2s

The dead reckoning estimated pose for driving in 1m square 4 times is shown in Figure 14.
Plots of the robots linear and rotational velocity as it drives one loop around the 1m square are shown in Figure 15 and Figure 16, respectively.

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: The occupancy grid made from the log file for the 10m x 10m x 5cm obstacle course is shown in Figure 17 where the green line is odometry pose, the yellow is true pose, and the blue is SLAM pose.

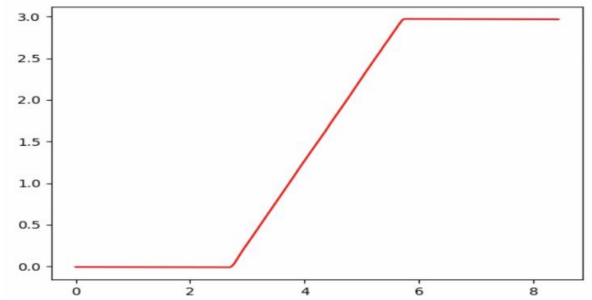

Fig. 13: Robot Frame Heading (radians) versus Time (seconds) for pi rad/s for 2s

TABLE V: Parameters for RTR Controller

Tuning Parameters	Slow	Fast	Square
Translational Gain (K_v)	1.00	1.00	1.00
Rotational Gain (K_w)	1.00	1.00	1.00
Beta Gain (K_b)	0.5	0.5	0.5
Distance Threshold	0.01	0.18	0.05
Angular Threshold	0.07	0.1	0.07

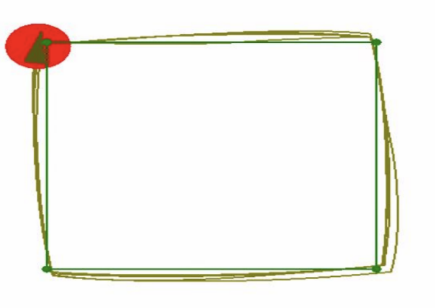

Fig. 14: Robot's Dead Reckoning Estimated Pose Driving in a Square 4 times

2) Monte Carlo Localization:

a) Action Model: The tuned dispersion constants for the action model are tabulated in Table VI.

TABLE VI: Dispersion Constants for Action Model

Dispersion Constants	Optimal Value
Turning Dispersion (K_1)	0.8
Translational Dispersion (K_2)	0.1

b) Sensor Model Particle Filter: The time to update the particle filter with 100, 300, 500, and 1000 particles are tabulated in Table VII. Interpolating these data points, the maximum number of particles that could be supported on 10Hz was calculated to be 2,443 particles.

In Figure 18, 300 particles were plotted at each midpoint of the 1m translation and at the corners after having turned 90 degrees.

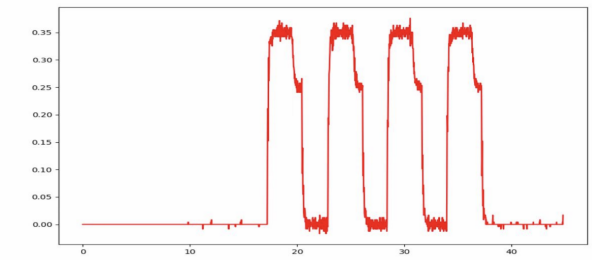

Fig. 15: Robot's Linear Velocity (PWM) versus Time (seconds)

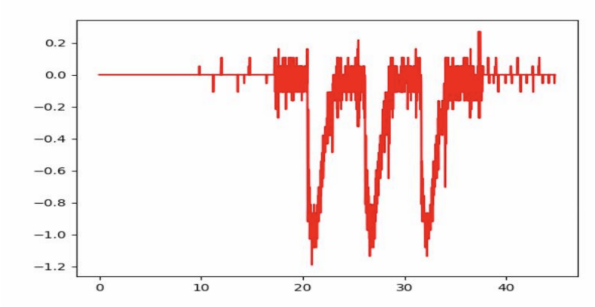

Fig. 16: Rotational Velocity (radians) versus Time (seconds)

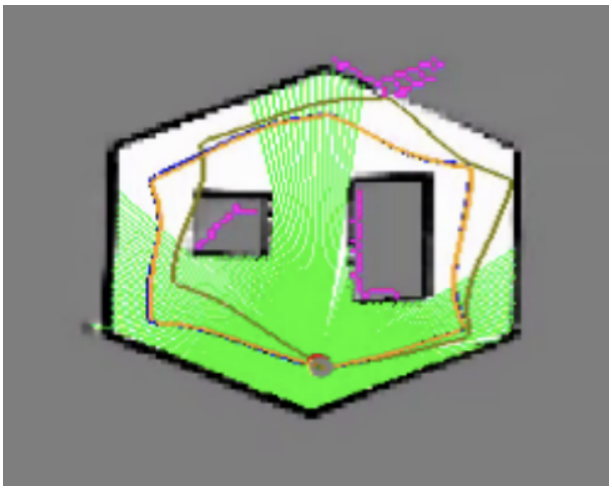

Fig. 17: Mapping Occupancy Grid

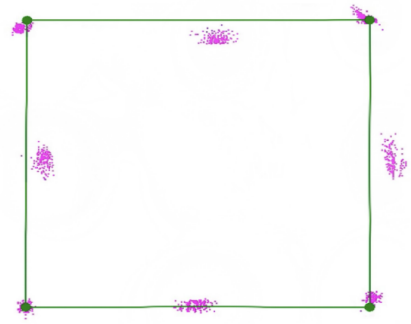

Fig. 18: Plot of 300 particles at corners and midpoints

TABLE VII: Time to Generate Particles

Number of Particles	Time to Update Filter (microseconds)
100	4,597.941
300	13,928.144
500	22,637.302
1000	42,064.125

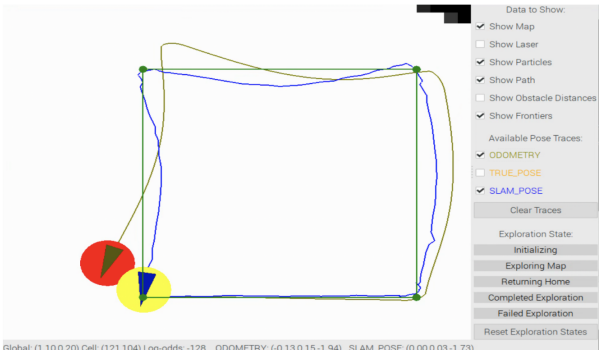

Fig. 19: Plot of Pose Error of SLAM versus Odometry

3) Simultaneous Localization and Mapping: The RMSE values that were calculated for x, y, and theta at each time step are shown in Figure 20.

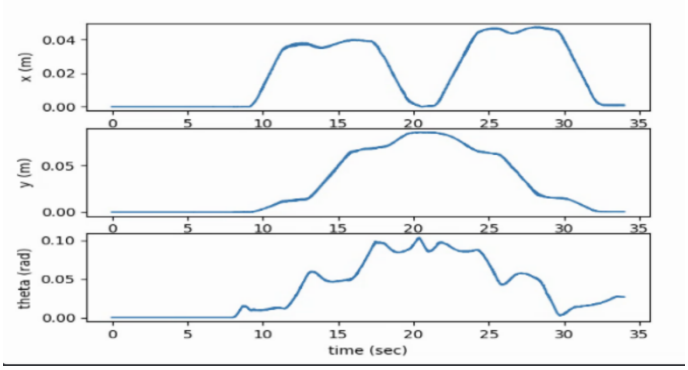

Fig. 20: Plots of RMSE for x, y, and theta over time

C. Implementing Planning and Exploration

1) Obstacle Distance Grid: A visualization of the obstacle distance grid is shown in Figure 21 where the red regions are areas to avoid for the robot when planning paths.

2) A Path Planning:* The planned path from A* and the executed path for the custom environment is shown in Figure 22. The execution times of the A* obstacle tests are shown in Table VIII.

IV. Discussion

Overall, this report has shown that SLAM offers much greater accuracy in pose estimation over odometry and

Fig. 21: Obstacle Distance Grid

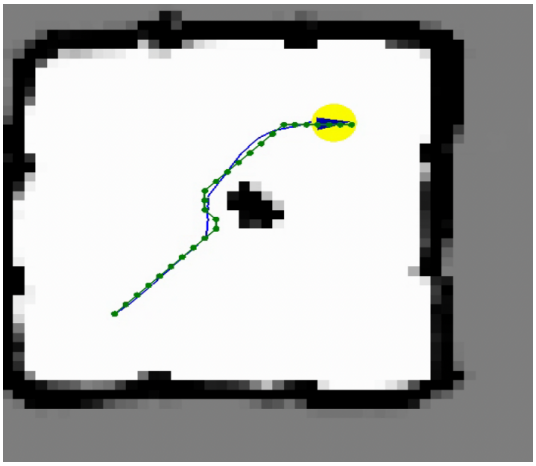

Fig. 22: Planned Path for Custom Environment

TABLE VIII: A* Performance Statistics (microseconds)

Test Name	Min	Mean	Max	Median	Std Dev
Convex	140	193.5	247	0	53.5
Empty	9169	10641.7	12689	10067	1493.38
Maze	2742	26692	61517	4608	24493.5
Narrow Constriction	6153	8022.67	9053	9053	1324.35
Wide Constriction	3127	6072	9190	9190	2478.23
Filled	13	24.2	49	22	13.197

gyroscope estimations. This is clear from comparing the two poses in 17 and 19. Although fusing gyroscope data with odometry improves the odometry pose estimates for turning and and at fast speeds, it is still prone to error propagation. On the other hand, we can see from the RMSE values shown in Figure 20 that the error for x remained below 0.05 m, below 0.10 m for y, and below 0.10 radians for theta. It can be observed that for the first 10 seconds, the error is extremely close to zero , indicating the error is very low for short periods of time.

Our final A* algorithm worked well, passing the test cases as well as navigating a custom path around an obstacle. Our initial implementation only used 4-neighbor node expansion for each cell coupled with Manhattan distances to calculate h and g costs. When testing this implementation, we noticed that the path created around corners were step/stair like, which the RTR controller had a difficult time maneuvering. This was solved by instead using an 8-neighbor node expansion for each cell and using diagonal distance instead of Manhattan distance. In addition to this, we decreased the distance and angular thresholds on the RTR controller. This made the paths created much smoother and easier for the robot to maneuver around.

V. References

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