Team 11 ROB 550 BotLab Report

Yifan Xu, Varun Aggarwal, Daksh Narang, and Xiaoyu Sun

***Abstract*—In this project, we developed a fully autonomous nonholonomic differential drive robot. We implemented a combination of feedforward and feedback control to control the motions of our robot. Additionally, we implemented a Simultaneous Localization and Mapping (SLAM) algorithm to perceive and process our robot's environment. Furthermore, we implemented A* search algorithm for planning paths in known environments. At large, the project encapsulates action, perception, and reasoning for a ground robot. As a result, our robot can successfully explore an unknown environment autonomously.**

I. Introduction

IN the first part, we want to be able to control the robot's motions manually. In order to accomplish this, we leveraged data from encoders to control the speed of the wheels using our control mechanism. After we are able to control the robot, we move to build a more sophisticated algorithm for mapping and localizing the environment. We first constructed an occupancy grid using known poses. Following that, we implemented Monte Carlo Localization in a known map. Finally, putting together the mapping and localization parts, we created a full SLAM system. With the knowledge of where the robot is and what is in its environment, we implemented planning with A* and a frontier exploration algorithm.

II. Methodology and Result

A. Motion and Odometry

1) Wheel Speed Characterization for an Open Loop Controller:

In Classical Control Theory, an open loop approach to control a plant, in this case the left and right motors of our robot, is implemented through a mapping function that takes a setpoint as an input and maps it to the an appropriate controller output (i.e., the signal commanding the actuator). In contrast to the closed loop control approach that monitors and reduces the error, the open loop approach does not minimize the error between the actual and the desired outputs. However, this comes with the drawback of not having a criterion to determine how the actual output compares to the desired output. In this section, we conducted experiments to collect data that helped understand the mathematical relationship between the target setpoint and the motor input signal: open-loop calibration.

We ran each motor from 0.0 m/s up to its maximum possible speed under full load conditions by varying the amount of voltage applied to the motor by varying the duty cycle (Pulse Width Modulation, PWM) of the actuation signal. Through our tests, we collected data from the encoders and processed it to estimate the actual speed the motor spinned at. Based on our data, we fitted an affine mapping function to acquire the mathematical relationship between the two variables.

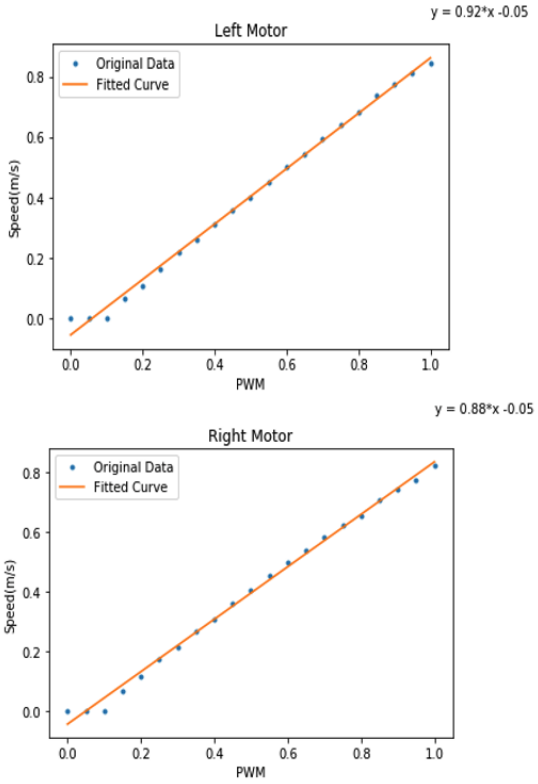

Fig. 1: Motor Calibration

As can be seen in the graphs in Figure 1, for our teams' robot, the mathematical relationship between the setpoint and the controller signal is mostly linear except at the starting and ending intervals.

2) Open and Closed Loop Wheel Speed Controllers:

In order to control the speeds of the left and right wheels of our robot, we implemented a combination

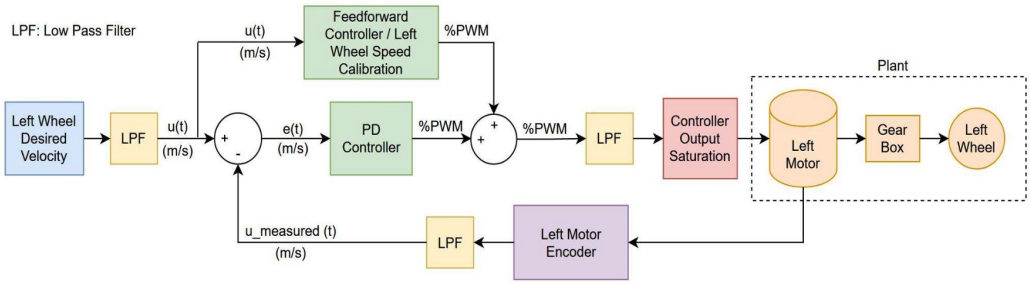

Fig. 2: Left wheel's speed controller's architecture, right wheel's speed controller is identical

of feedforward (FF) and feedback (FB) control mechanisms. The idea behind using FF control was to get the motor running close to the desired or commanded speed. Once the motor reaches close to the desired speed by virtue of the FF controller, the FB controller works to eliminate/minimize the steady state error caused by the FF controller. Together, the FF and FB controllers allow us to achieve minimal steady-state error between desired and actual speed with minimal overshoot and oscillations. Figure 2 shown above depicts the architecture of our controller for controlling the speeds of the left and right wheels of our robot.

In order to implement the FF controller, we used the calibration data described in Section 1.1. The closed-loop controller for each wheel consists of three low-pass filters (LPF), a FF controller, a proportional FB controller, a derivative FB controller, a controller output saturation block, and a sensor (encoder) to keep track of the real-time velocity estimate of the wheel. Specifically, the first LPF is used to slow down the instantaneous acceleration and deceleration of the wheel due to sudden changes in desired velocity commands (step inputs). The second LPF is used to minimize the negative effects of high frequency noise amplification caused by the derivative term in the FB controller. Finally, the third LPF is used to minimize the high frequency discretization noise caused by the encoder. The controller output saturation block protects the motors from running in their intermittent operating region for extended periods of time.

In order to tune our FB controllers, we examined the results of running our system with just FF controllers to get an idea about how much controller effort is needed from the FB controllers to minimize the steady-state error. The results of running our robot with just FF controllers are shown in Figure 3.

In Figure 3, the FF controller causes negative steady-state errors in the system. In order to minimize the steady-state error caused by the FF controllers, we used a unity gain (Kp=1) proportional FB controller. While the proportional controller helped minimize the steady-state error, it induced oscillations in the system's response to

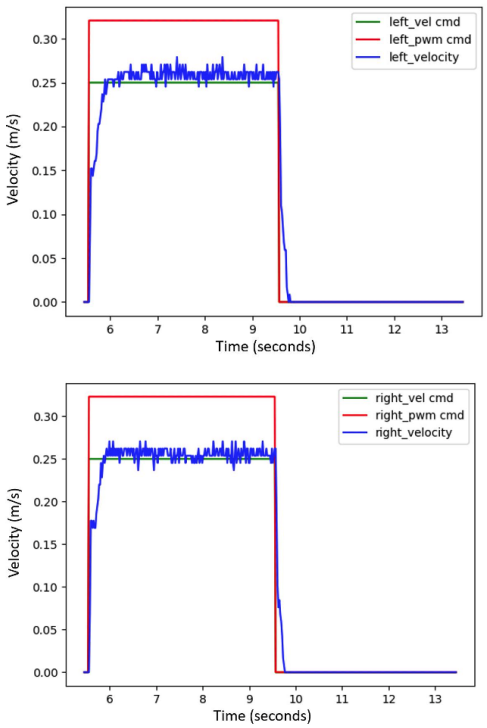

Fig. 3: Results of controlling left and right wheels' speeds with just feedforward control

step inputs. In order to damp out the oscillations, we added a derivative component to the FB controller. We experimented with the value of the gain associated with the derivative component (Kd) by steadily increasing it from 0.1 to its final value of 0.4. The results of running our robot with both FF and FB controllers are shown in Figure 4 below.

As can be seen in the graphs in Figure 4, the FB controller minimizes the steady-state error caused by the FF controller. Figure 5 contains the parameters of the controllers described above.

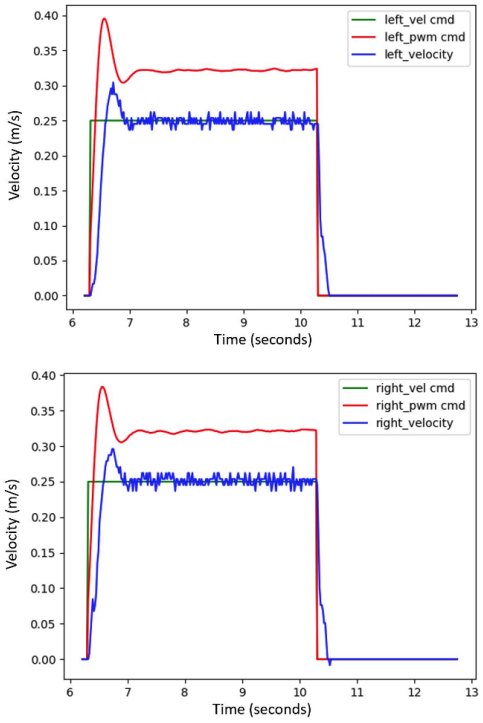

Fig. 4: Controlling left/right wheels' speeds with feed-forward and feedback (PD) controllers

Parameter	**Left Wheel**	**Right Wheel**
Calibration Slope (Speed vs. PWM) (FF)	0.9977	0.97
Calibration Intercept (Speed vs. PWM) (FF)	-0.07	-0.0633
Proportional gain (K_p) (FB)	1.0	1.0
Derivative Gain (K_d) (FB)	0.4	0.4
Setpoint LPF Frequency (Hz)	10	10
Controller Output LPF Frequency (Hz)	10	10
Sensor Output LPF Frequency (Hz)	10	10
Controller Upper Saturation Bound (PWM)	0.85	0.85
Controller Lower Saturation Bound (PWM)	-0.85	-0.85

Fig. 5: Controller parameters

3) Odometry:

In order to validate our odometry model, we manually moved our robot by known distances and rotated it by known angles. We traversed our robot along a rectangle path (0.30 m x 0.25 m). A plot of the robot's position estimate through odometry is shown in Figure 6. On a clean, even surface, the robot's pose estimate through odometry is extremely accurate and doesn't warrant any correction parameters.

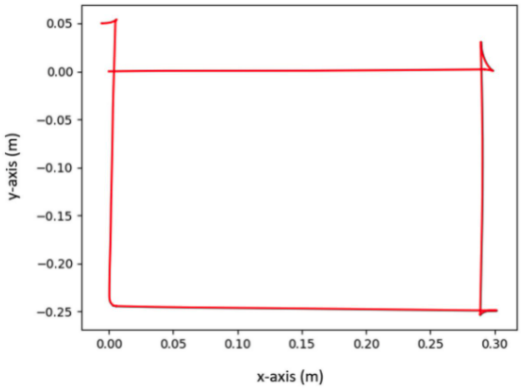

Fig. 6: Odometry Validation

4) Gyro Sensor Fusion:

In order to perform odometry, we utilized the following equations,

$$\Delta s = \frac{\Delta sl + \Delta sr}{2}$$

where, Δs is the displacement (Δd) of the center of the robot's axle, Δsl is the displacement of the left wheel of the robot, and Δsr is the displacement of the right wheel of the robot.

$$\alpha = \frac{\Delta sr - \Delta sl}{b}$$

where, α is the small change in the heading angle of the robot ($\Delta\theta, \Delta\theta_{odo}$) and b is the track width (distance between the left and right wheels) of the robot.

Finally, to ascertain how the position of our robot changes in the global/world frame,

$$\Delta x = \Delta d \times cos(\theta + \frac{\Delta\theta}{2})$$

$$\Delta y = \Delta d \times sin(\theta + \frac{\Delta\theta}{2})$$

where, Δx and Δy represent the changes in the x and y positions of the robot's axle's center in the global frame.

We obtained Δsl and Δsr by computing the difference between the current and previous encoder readings obtained from the encoders directly attached to the left and right motors respectively and by multiplying the obtained differences in encoder ticks by the constant converting it to meters.

Gyrodometry [1], [2] is a method for combining measurements from a gyroscope with measurements from wheel encoders (odometry). The method based on the idea that non-systematic odometry error sources (such as bumps) impact the vehicle only during very short periods; typically a fraction of a second for each encounter. During these short instances the readings from the gyro and from odometry differ significantly. Such that in our implementation, if the wheels are not moving, we ignore the gyro reading. If gyro readings are significantly different than odometry, then we might hit a bump and should trust the gyro over odometry.

$$\Delta_{G-O} = \Delta\theta_{gyro} - \Delta\theta_{odo}$$

If $(|\Delta_{G-O,i} > \Delta\theta_{thres}|)$, which means the Gyro readings are significantly different than odometry, then

$$\theta_i = \theta_{i-1} + \Delta\theta_{gyro,i}$$

else,

$$\theta_i = \theta_{i-1} + \Delta\theta_{odo,i}$$

We obtained $\Delta\theta gyro$ by computing the difference between the current and previous readings of the Digital Motion Processed (DMP) Tait-Bryan angle (yaw) obtained from the Inertial Measurement Unit (IMU) on board the BeagleBone Blue (BBB).

Here, by trial and error, we set the threshold to 6.25 degree, which is 0.109 in radian.

5) Robot Frame Velocity Controller:

For controlling the robot frame's velocity, we used the following equations for relating the left and right wheels' velocities with the robot forward and turning velocity commands

$$v_R = \omega(R + b/2)$$

$$v_L = \omega(R - b/2)$$

where, v_R and v_L are the velocities of right and left wheels respectively, ω is the angular/turning velocity of the robot, R is the radius of the turning circle and b is the track width. By using the relation between linear and angular velocities, the equations above can be rewritten as,

$$v_R = v + \omega b/2$$

$$v_L = v - \omega b/2$$

By breaking down the commanded robot frame forward and turning velocities into left and right wheel velocity commands, we utilized our closed-loop wheel speed controllers described in Section 1.2 to maneuver our robot with the desired forward and turning velocities. The parameters for the controller are listed in Figure 5.

Plots of robot's x-position (robot frame) vs. time for step inputs of varying magnitudes are shown in the Figures 7, 8, 9.

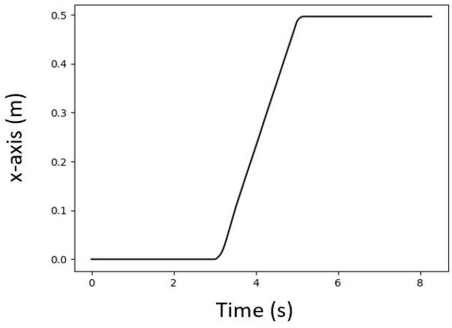

Fig. 7: x position of 0.25 m/s for 2 s

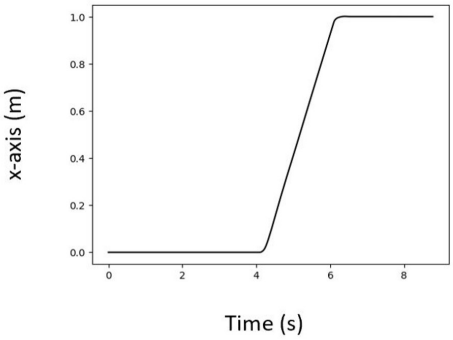

Fig. 8: x position of 0.50 m/s for 2 s

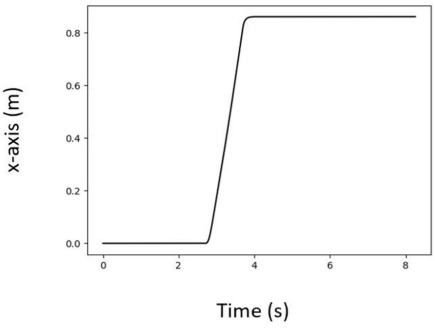

Fig. 9: x position of 1.0 m/s for 1 s

From the plots of x position vs. time above, we can see that the controller works well for 0.25 m/s and 0.50 m/s step commands, however, it doesn't work well for a step input of 1.0 m/s. In order to move at 1m/s, the

motors need very high duty cycles (>90%) very quickly. As a result, the robot often skids and becomes difficult to control. Furthermore, under full load conditions, the robot never reaches a speed of 1.0 m/s even with a 100% duty cycle input.

Plots of robot's heading (robot frame) vs. time for step inputs of varying magnitudes are shown in the Figures 10, 11, 12 below.

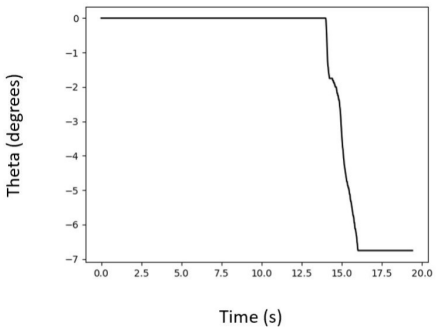

Fig. 10: heading angle of π/8 rad/s for 2s

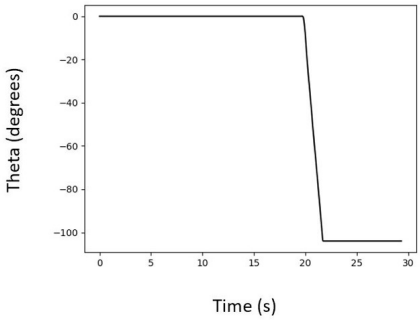

Fig. 11: heading angle of π/2 rad/s for 2s

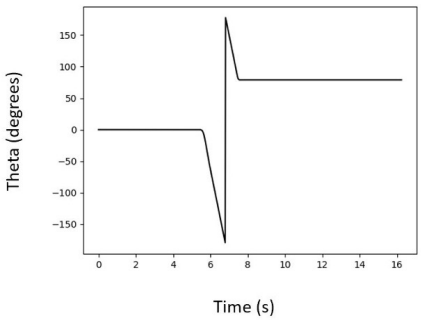

Fig. 12: heading angle of π rad/s for 2s

The plot shown in Figure 12 shows a sudden change in the value of the heading angle. The reason for the sudden change is that the heading angle of the robot is always wrapped between $[-\pi, \pi]$ radians. Therefore, the angle goes back to π when it exceeds $-\pi$ while rotating from the clockwise (CW) direction and goes to $-\pi$ when it exceeds π while rotating from the counter-clockwise (CCW) direction.

6) Motion Controller:

For the motion controller, we used a Rotate-Translate-Rotate (RTR) [3], [4] type controller to track waypoints. Specifically, our controller consists of two elements. One, a turn maneuver and two, a straight maneuver. The turn maneuver causes the robot to rotate such that it points (heads) in the direction of the target pose. It does so by controlling the rotation of the robot at its current position such that at steady state, the robot points towards the target. Since we do not care about the heading angle of our robot at its target location, we did not incorporate an element responsible for ensuring that the robot heads in a specific direction upon reaching the target position in our controller. Therefore, in essence, our controller works like a RTR controller for the first n-1 points in our trajectories consisting of n total points where n is a positive integer greater than 1.

The turn maneuver is realized by the mathematical equations shown below:

$$\omega_{sp} = K_\omega \alpha$$

$$v_{sp} = 0$$

where, ω_{sp} is the turn velocity (angular velocity), v_{sp} the translational velocity of our robot, and K_ω is the proportional gain associated with the required turning angle, α (angle between current heading and heading towards goal).

For the straight maneuver, we used both the translational and turning velocity components. Using both components, ensures that we move along a straight line and maintain by dynamically correcting our direction of motion in case of any deviations. Using only translational velocity in the straight maneuver can cause the robot to deviate from its intended path because of a multitude of reasons. Difference in the transient response of each wheel, difference in the settling time of each wheel, difference in the friction associated with the patch of ground each wheel is rotating on, and the like. Mathematical equations utilized for implementing the straight maneuver component of our controller are as follows:

$$\omega_{sp} = K_\omega \alpha$$

$$v_{sp} = K_d d$$

where, ω_{sp}, K_ω, and α are the same as above. v_{sp} is now directly proportional to the amount of distance the robot needs to traverse, d and the constant of proportionality is the proportional gain, K_d. The parameters and thresholds are shown in table below. d_threshold and α_threshold are the thresholds that serve as error tolerances about the target poses.

Parameters	Values
K_ω	0.8
K_d	0.5
d_threshold (m)	0.05
α_threshold (radians)	0.07

The plot of dead reckoning estimated pose of our robot by commanding it to drive a 1m square four times is shown below.

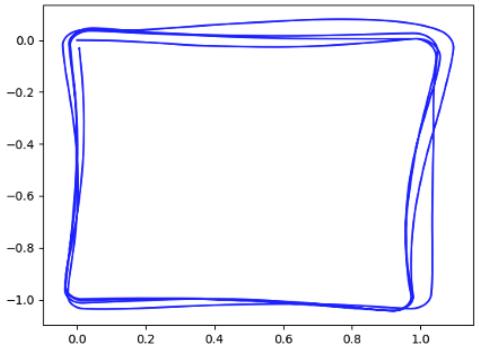

Fig. 13: Dead reckoning position estimation by driving a 1m square four times

The plots of the robots linear and rotational velocity as it drives one loop around the square are shown below.

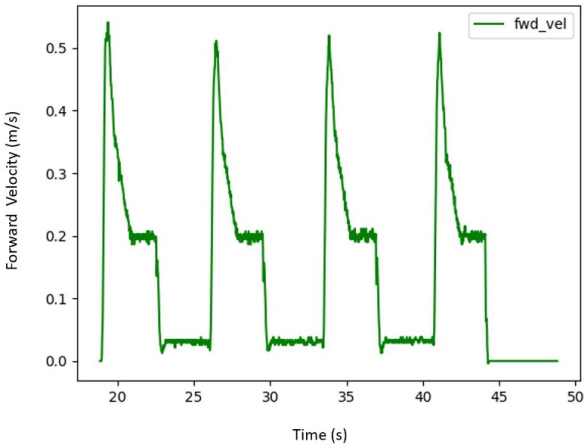

Fig. 14: Forward velocity of the robot as a function of time as it drives a square path once

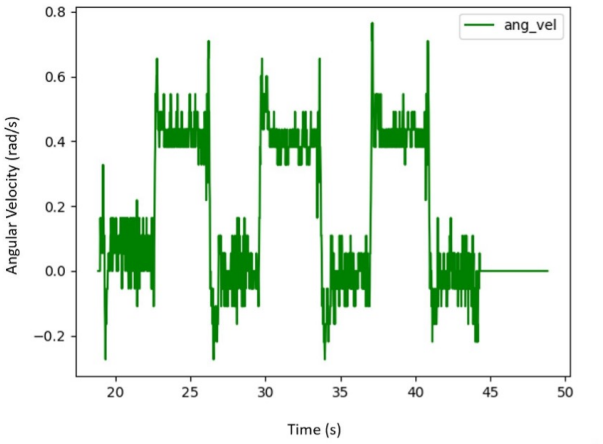

Fig. 15: Angular velocity of the robot as a function of time as it drives a square path once

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping:

Figure 16 shown below depicts a map/obstacle distance grid for a provided .log file.

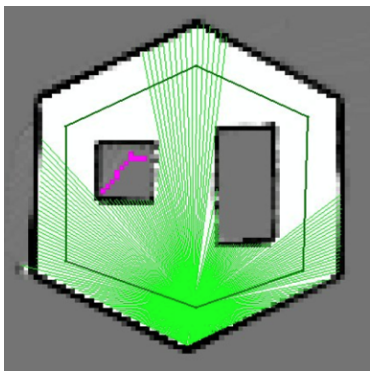

Fig. 16: Map from the log file obstacle slam 10mx10m 5cm

2) Monte Carlo Localization:

a) Action Model:

From odometry, we have previous and current poses, which we can use to calculate the changes in translation and rotation.

$$\Delta s^2 = \Delta x^2 + \Delta y^2$$

$$\alpha = atan2(\Delta y, \Delta x) - \theta_{t-1}$$

Thus, we modelled action as rotation, translation, and rotation (RTR): $u = [\alpha \;\; \Delta s \;\; \Delta\theta - \alpha]$. Figure 17 below depicts the RTR model used above.

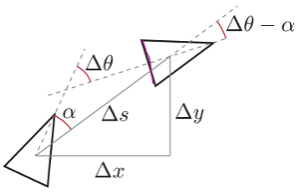

Fig. 17: Action model

Following that, we further model the action error:

- Turn$(\alpha + \varepsilon_1)$
- Travel$(\Delta s + \varepsilon_2)$
- Turn$(\Delta\theta - \alpha + \varepsilon_3)$

Using the above equations, we assigned Gaussian error ε to both translational motion and rotational motion. Since we calibrated odometry to eliminate systematic errors, we have a 0 mean error Gaussian distribution. As a consequence, we assume that there is no systematic error.

Then we use following equations to combine the error and changes together to update the new position. Given $u = [\alpha \;\; \Delta s \;\; \Delta\theta - \alpha]$:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} = \begin{bmatrix} (\Delta s + \varepsilon_2)cos(\theta_{t-1} + \alpha + \varepsilon_1) \\ (\Delta s + \varepsilon_2)sin(\theta_{t-1} + \alpha + \varepsilon_1) \\ \Delta\theta + \varepsilon_1 + \varepsilon_3 \end{bmatrix}$$

where:

- $\varepsilon_1 \sim \mathcal{N}(0, k_1|\alpha|)$
- $\varepsilon_2 \sim \mathcal{N}(0, k_2|\Delta s|)$
- $\varepsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|)$

To tune the action model [5], we performed straight line experiments and rotation experiments to determine reasonable values of k_1 and k_2.

- If both k_1 and k_2 are too small: it will track odometry position only, and take it as the true pose.
- If the dispersion is too large, particles will be everywhere.

By trial and error, we chose these two numbers because they worked well.

Parameters	Values
k_1	0.01
k_1	0.01

b) Sensor Model and Particle Filter:

Time taken to update the particle filter for 100, 300, 500 and 1000 particles is tabulated in Table below.

Particle Numbers	Time (microseconds)
100	4573.53
300	11786
500	17683.5
1000	31228.3

10Hz can be converted to 0.1 second (period) which is 100,000 microseconds. From the table above, we know that 1000 particles take 31228.3 microseconds to update, the estimated maximum number of particles can be supported at 10Hz is 3200. 300 particles at the midpoint of each 1m translation for a square and at the corners after having turned 90 degrees can be seen in Figure 18 below.

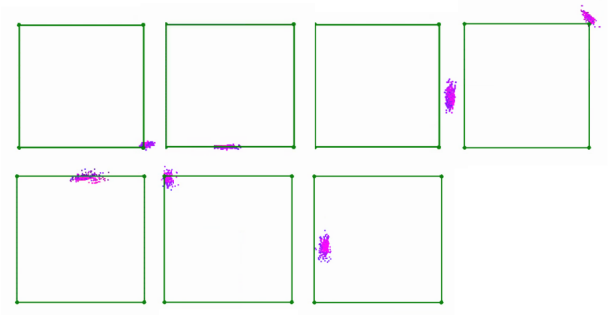

Fig. 18: 300 particles at the midpoint of each 1m translation and at the corners after having turned 90°

Plot showing how the pose error difference between the SLAM and Odometry poses evolves over time is shown in Figure 19 below.

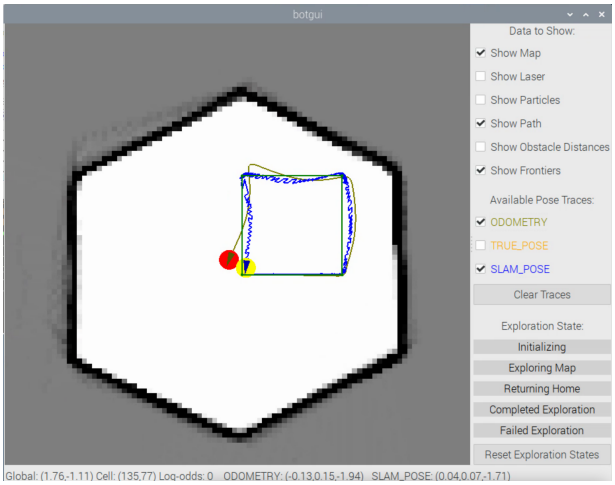

Fig. 19: pose error difference between the SLAM and Odometry poses

3) Combined Implementation:

A block diagram showing how the SLAM system components interact is shown in Figure 20 below.

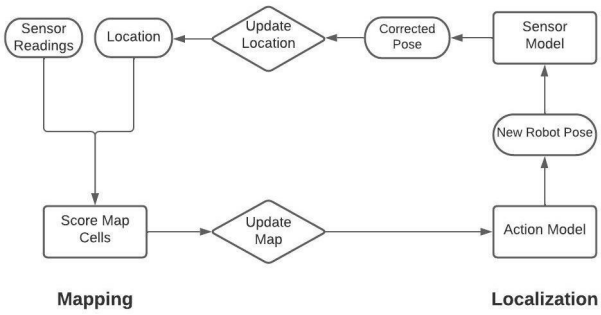

Fig. 20: SLAM system

We compared the estimated poses from our SLAM system against the ground-truth poses in obstacle_slam_10mx10m_5cm.log and obtained the RMS errors tabulated in the table below.

RMS	Values
Distance	0.07739
x	0.05560
y	0.05382
theta	0.07467

C. Planning and Exploration

1) Path Planning:

Figure 21 shown below depicts a path planned by our A* path planning algorithm in a maze environment with the actual path driven of our robot overlayed on top.

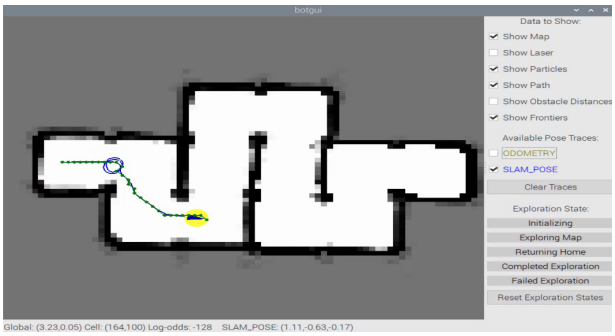

Fig. 21: Path planned through our A* algorithm and actual path traversed by our robot in a maze environment

Statistics on our path planning execution times for each of the example problems in the data/astar folder are shown in the Table below.

(μs)	Min	Mean	Max	Median	std dev
convex_grid	996	996	996	0	0
empty_grid	3464	7559.33	10127	10127	2926.8
maze_grid	881	6554.75	11261	5109	3945.07
narrow_grid	5787	567350	996	1.12891e6	0
wide_grid	10562	923133	1.72044e6	1.72044e6	702798

(μs)	Min	Mean	Max	Median	std dev
convex_grid	40	112	161	135	52.0064
empty_grid	38	90	142	0	52
filled_grid	39	62.4	89	63	19.7342
narrow_grid	90	642961	1.92866e6	134	909126
wide_grid	61	61	61	0	0

From the statistics above, we can conclude that our path planning algorithm (A*) is optimal and fast.

2) Exploration:

Using connected components search in the occupancy grid we found the frontiers. A 4-way search was carried around each free space cell to locate the frontier cells. Upon locating the frontier cells, an 8-way search grew the frontier. All the frontiers were identified using the *find_map_frontiers()* function.

Once we had a collection of frontiers, we chose which frontier to explore. Among the available frontiers, we calculated the Euclidean distance from the current pose to the start cell of each available frontier. The one with the shortest distance was returned and a path to it was planned using our A-star algorithm. Once there are no frontiers left to explore, the program ensures that all free space is surrounded by occupied cells.

3) Map Localization with Unknown Starting Position:

Particles were uniformly distributed using a uniform distribution function. The distance of each particle from the nearest frontier was calculated. The pose with the farthest distance from the frontier was chosen and the mobilebot moved along the direction towards that frontier. The pose was updated, and particles were redistributed. The same procedure was iterated until a narrow distribution was obtained.

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] ——, "Measurement and correction of systematic odometry errors in mobile robots," *IEEE Transactions on robotics and automation*, vol. 12, no. 6, pp. 869–880, 1996.

[3] S. Parsons, "Introduction to autonomous mobile robots by roland siegwart and illah r. nourbakhsh, mit press, 321 pp., $50.00, isbn 0-262-19502." *Knowledge Eng. Review*, vol. 19, pp. 379–380, 2004.

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

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