Team 3 ROB 550 BotLab Report

Zahraa Bazzi, Rahul Agrawal, Michelle Ji

***Abstract*—This report outlines the algorithmic design of an autonomous exploration robot that operates within a 2D environment. The program is designed for a 2-wheel differential drive robot that uses the Raspberry Pi 4B and Beaglebone Blue to perform calculations and commands. The robot uses encoders and 2D LiDAR to perform rotation-translation-rotation based motion control, Monte Carlo SLAM, A* path planning, and frontier exploration. The current implementation of the robot is capable of autonomous travel through an unknown 2D environment.**

I. Introduction

The purpose of BotLab is to implement a motion control system that uses simultaneous localization and mapping (SLAM) to explore unknown environments. The lab explores major topics within acting, perception, and reasoning. Topics in acting include PID controller design, trajectory planning, and motion control. Perception tools include encoders, gyroscopes, and LiDARs to measure speed and pose. Reasoning algorithms are implemented for path planning. This report covers the methods to achieve such functionalities in section II, the performance in section III, and summarizes the issues and future improvements in section IV.

II. Methodology

A. Motion and Odometry

1) Motor Calibration: In order to measure the loaded steady state wheel speed, we defined the gear ratio, wheel base, and wheel diameter. To calculate the gear ratio, we marked the wheel and measured the encoder reading for one wheel revolution. We then marked the magnet on the inside of the motor and measured the encoder reading for one revolution. The first encoder reading divided by the second is the gear ratio. We measured the rotational velocity of a single wheel by using the encoder count for the wheel along with the parameters of the robot previously determined. To reliably use such a method, we first calibrated the loaded robot's angular speed against the duty cycle input.

The calibration is done by first setting the desired duty cycle to the motors. After giving the motors one second to speed up, the encoders were reset and the resultant value is read after a specified amount of time Δt (1s). Afterwhich, the rotational velocity of each motor is calculated using the equation below. This process was repeated for duty cycles from 0 to 1 in 0.05 increments.

$$\omega = \frac{2\pi * ticks}{GearRatio * EncoderResolution * \Delta t} \quad (1)$$

Each motor was calibrated individually by receiving a PWM command while the other motor on the bot did not receive any inputs. This resulted in the bot spinning around the axis of the non-rotating wheel.

2) PID Controller: To develop an accurate PID controller, we experimented with features such as Feed Forward control, PID control, conditional reset of integrator, and low pass filters. To evaluate performance after each controller update, we generated a plot for PWM and measured velocity (step test plot) as well as a plot of the bot driving in a square path (square plot).

Feed Forward control takes forward and turn velocity as input to calculate the motor PWM commands. This is shown in the following equations where m and b are the slope and y-intercept of the motor calibration section.

$$pwm_l = m*(fwdVel - \frac{wheelBase}{2} * turnVel) + b \quad (2)$$

$$pwm_r = m*(fwdVel + \frac{wheelBase}{2} * turnVel) + b \quad (3)$$

After running the Feed Forward control, the step test plots were used to determine the time constant for the PID control, which represents the time it takes to reach 63% of settling velocity after the step input is sent.

To further tune the PID control with the new time constants, the integral and derivative terms were set to zero, and the proportional term was varied until the plots showed that the actual velocity and PWM command steady state values were overlapping.

Next, the Ziegler-Nichols method was used to find PID values that would reduce the noise in the plots. In the formulas in Table I below, K_u represents the P value used in the previous step, Pu represents the measure of one oscillation in the step test plot, while K_p, K_i, and K_d represent the new PID values, respectively.

Control Type	K_p	K_i	K_d
PID	0.60 K_u	2 K_p/P_u	$K_pP_u/8$

TABLE I: PID using Ziegler-Nichols method[1]

The calibrated Feed Forward control and tuned PID control were combined into one controller. To reduce inaccurate motions due to noise, the parameters were tuned to improve the stability and steady-state error. Optimal values were selected by compromising overshooting and

settling time to improve stability[1]. The square plot was used to judge the performance of the bot based on how well it achieved 90° turns and straight lines when driving in a square.

Regularly checking the integrator value at every control loop for a conditional reset did not significantly improve the output. As shown in Fig. 1 below, the final controller is a combination of the calibrated feed forward control and the tuned PID control, with a low pass filter on the forward and turning velocities as well as the left and right motor velocities. The low pass filters were implemented to reduce the noise and improve the motion of the bot when it is driving forward or turning.

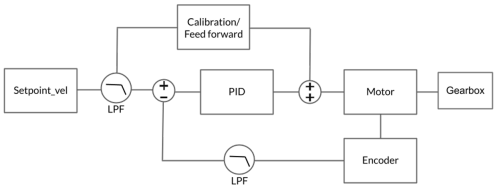

Fig. 1: Controller setup

3) Odometry: Encoders are used to update the robot position estimate (x, y, and θ), which are initialized to zero. Each motor shaft of the Mbot has an encoder sensor and the angular displacement is given by

$$\Delta\phi = \frac{2\pi * \Delta EncoderReading}{GearRatio * EncoderResolution} \tag{4}$$

The distance each wheel traveled is given by

$$\Delta s_w = r * \Delta\phi \tag{5}$$

where r is the radius of the wheel.
The change in x, y, and θ are computed as

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b} \tag{6}$$

$$\Delta d = \frac{\Delta s_r + \Delta s_l}{b} \tag{7}$$

$$\Delta x = \Delta d * cos(\theta + \Delta\theta/2) \tag{8}$$

$$\Delta y = \Delta d * sin(\theta + \Delta\theta/2) \tag{9}$$

where b is the wheelbase.
Finally, x, y, and θ are updated by adding Δx, Δy, and $\Delta\theta$ to the original values respectively.

Using performance evaluation from the competition as reference, if odometry accuracy test results fall below 5% error, the model is sufficiently accurate for an initial estimate. As this initial estimate of the robot's position would be further refined using other sensors (Gyroscope, LIDAR) and methods like SLAM, no calibration procedure such as UMBark calibration was performed.

4) Gyroscope and Odometry: To increase the accuracy of the odometer, we fused the angular data from the gyroscope to calculate the angles reported by the odometer. In the case of slip, the encoder generated odometry could over-report compared to the true angular displacement. The gyroscope, although not affected by slip, is susceptible to bias, which builds up the error over time. Therefore, we select a threshold that is larger than the observed gyroscope bias. If a reasonable discrepancy exists between the two sensors, then the gyroscope will be used to update the odometry.

$$|\Delta\theta_{gyro} - \Delta\theta_{odom}| \geqslant threshold \tag{10}$$

To determine an optimal angular threshold, the robot will be tested in two conditions where slippage may commonly occur: travel over different surfaces and turning. As such, the first test condition will be to have the robot travel straight from one surface into another, which we chose to be from carpet to tiled floors, as the two surfaces were shown to drastically alter performance during the introductory assignment. The second test has the robot traveling straight for 1m at 0.55m/s before making a rapid 90° turn at πrad/s. In both tests, the angular displacement of the odometer and gyrodometer will be compared to the true angular displacement.

5) Robot Frame Velocity Controller: A final velocity controller was implemented in the robot's frame that takes inputs of forward velocity (m/s) and turning velocity (rad/s). As shown in Fig. 2, the input velocities are converted to PWM commands using a PID in addition to the controller architecture of Fig. 1.

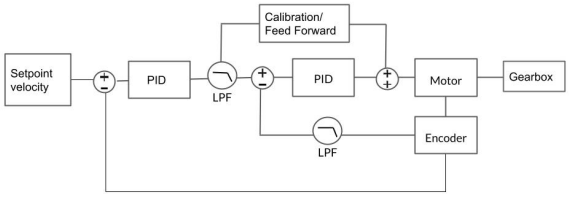

Fig. 2: Robot frame velocity controller

The actual forward ($v_{f(actual)}$) and turn velocity ($v_{t(actual)}$) of the robot are computed as

$$v_{f(actual)} = \frac{v_L + v_R}{2} \tag{11}$$

$$v_{t(actual)} = \frac{v_R - v_L}{b} \tag{12}$$

where b is the wheel base, and v_L and v_R are the actual left and right wheel velocities of the robot respectively.

The error between the input and actual velocities are calculated as

$$error = v_{input} - v_{actual} \tag{13}$$

The errors are sent through a PID controller to get the setpoint velocities of the left and right wheels of

the robot. $v_{f(pid)}$ and $v_{t(pid)}$ are the output velocities of the PID, which is tuned as described in section A.2 The setpoint velocities are used to compute the PWM commands in the manner described in section A.2.

$$v_{L/R(setpoint)} = v_{f(pid)} \pm (v_{t(pid)} * b/2) \tag{14}$$

6) Motion Controller: The motion controller was originally designed using a simple rotate, translate, rotate (RTR) method that stopped at each target and rotated before proceeding to the following target. This method worked well when driving through the maze at a low speed of 0.25m/s, however, it did not work well at a high speed of 0.8m/s unless the speed was drastically ramped down before stopping. In order to accommodate for a variation of speeds, the controller was optimized using Feedback Control to a Pose with a discretized path using the Carrot Following method. This method works by defining the current x, y, and θ coordinates and determining the translation and rotation motions to reach the target coordinates. The translation and rotation motions are synchronized such that when the bot drives forward, it drives at an angle tangent to each discretized point to achieve the target turn defined by theta at the end of the translation. Since the rotation is synchronized with the translation, a separate state was added to the code to achieve an independent turn that returned the bot to its original pose at the end of a cycle. The formulas in Fig. 3 were used to determine the path required to go from the current to the goal position.

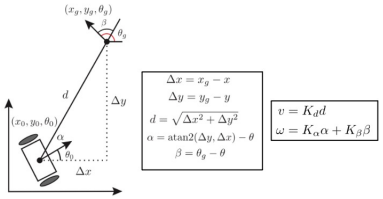

Fig. 3: RTR controller and Feedback Control to a Pose equations (ROB550 Lec 6 - Slide 14-15)

The gains on the distance (K_d), alpha (K_α), and beta (K_β) ensure the bot stability at higher speeds, keep the bot centered within its path, and control the turns to avoid obstacles. The stabilizing conditions for robust position control are provided in the equations below [2].

$$k_\rho > 0;\ k_\beta < 0;\ k_\alpha + \frac{5}{3}k_\beta - \frac{2}{\pi}k_\rho > 0 \tag{15}$$

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: Our mapping is designed using the occupancy grid method. The bot uses a LiDAR to evaluate its surroundings, which sends laser beams out and the beam returns the distance traveled after it hits an obstacle (endpoint). It then increments the weighting of the obstacle by 3 and uses Breshenham's algorithm to characterize all of the points between the bot and the endpoint as free space. When it identifies the free space, it decrements the weighting of each space by 1. The bot depicts this free space in the map as negative numbers, the obstacles as positive numbers, and unexplored territory as 0.

2) Monte Carlo Localization:

a) Measurement/Action Model: An action model is implemented to reflect the changes of position and model the increase of possible positional distribution when the bot is in motion. The action model assumes a rotation (rot1), translation (trans), rotation (rot2) motion controller and takes values from the odometer to update the posterior distribution. A Gaussian distribution is used to model the error in the resultant position, and the standard deviation (σ) of this error is proportional to the magnitude of the action at hand in the form of four error coefficients (k).

$$\sigma_{rot1} = \sqrt{k_1 * rot1^2 + k_2 * trans^2} \tag{16}$$

$$\sigma_{trans} = \sqrt{k_3 * trans^2 + k_4 * (rot1^2 + rot2^2)} \tag{17}$$

$$\sigma_{rot2} = \sqrt{k_1 * rot2^2 + k_2 * trans^2} \tag{18}$$

Coefficients are chosen by performing linear and rotational tests that result in a distribution that reflects the real world positional experiences of the robot. Coefficient k_1 affects the dispersion of the rotational uncertainty during rotation, k_2 is for the translational uncertainty during rotation, k_3 is for the translational uncertainty during translation, and k_4 affects the rotational uncertainty during translation. The standard deviation produces a sample particle location on a gaussian distribution, and is applied to the existing particle samples.

$$\theta 1_{sample} = sampleGaussian(rot1, \sigma_{rot1}) \tag{19}$$

$$d_{sample} = sampleGaussian(trans, \sigma_{trans}) \tag{20}$$

$$\theta 2_{sample} = sampleGaussian(rot2, \sigma_{rot2}) \tag{21}$$

$$x_{new} = x_{old} + d_{sample} * \cos(\theta_{old} + \theta 1_{sample}) \tag{22}$$

$$y_{new} = y_{old} + d_{sample} * \sin(\theta_{old} + \theta 1_{sample}) \tag{23}$$

$$\theta_{new} = \theta_{old} + \theta 1_{sample} + \theta 2_{sample} \tag{24}$$

b) Sensor Model: The objective of the sensor model is to determine the probability that the hypothesis pose matches the LiDAR scan for each particle in the posterior, and assign weights accordingly. The sensor model uses the Simplified Likelihood Field Model to give a score to each ray in a scan and computes the total scan score. For each ray in a laser scan, the coordinates of the endpoint are computed. If the endpoint is an obstacle (hit), then its log odds are added to the total

scan score. If it is not an obstacle, the cells before and after the endpoint are checked for an obstacle, and 1/3 of the log odds are added to the total scan score if positive. This fraction of 1/3 was determined through trial and error by observing the accuracy of the SLAM system. This total scan score is used to get the weights of all the particles in the posterior by normalizing the scores.

c) Particle Filter: The SLAM system performs localization using Monte Carlo Localization also known as particle filter. In this implementation, a sample of particles (posterior) are initialized as a Gaussian distribution around zero mean. These particles are input into the action and sensor models as described in the above sections. As a result, every particle is assigned a weight. The posterior pose of the Mbot is estimated by taking a weighted average of all the poses. Finally, the particles in posterior are resampled into prior based on their weights using Low Variance Resampling (LVR). LVR ensures that the resampling is done with replacement and minimizes the risk of losing diversity of particles. Our implementation of LVR also includes a check on the iterator of the posterior to ensure it remains within the bounds of the number of particles, hence making it robust against segmentation faults.

3) Combined Implementation: Fig. 4 depicts a block diagram of how the different components of the SLAM system interact. The SLAM system comprises of two major elements, mapping and localization (particle filter). At every iteration, first mapping is performed, which uses the moving laser scans to compute the log odds and updates the occupancy grid. The updated grid is used by the localization and specifically the sensor model. As shown in the block diagram, the particles are resampled and input into the action model. The sensor model then assigns them the weights by evaluating the probability of the poses matching the LiDAR scan. Finally, the posterior pose is computed using the weighted average of all the poses and the estimated pose is used by mapping.

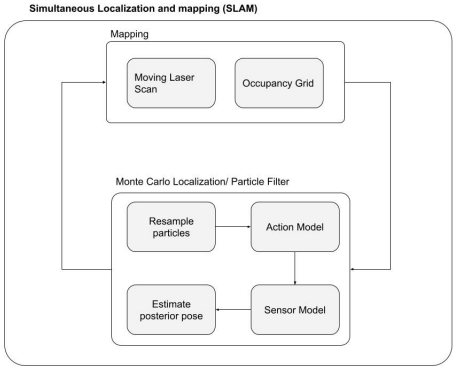

Fig. 4: Block Diagram of SLAM system

C. Planning and Exploration

1) Path Planning: A* is used to find the shortest path between the current position of the robot and the goal position. Each cell in the map is interpreted as a node in a graph. An 'open' and 'closed' list is maintained to track whether the cell has been visited or not. A priority queue is used for the open list to avoid sorting at each iteration. Our implementation of A* supports both 4-way and 8-way neighbors and we decide on what to use based on the task. We use Manhattan distance for the h-cost of 4-way implementation and the diagonal distance for 8-way. We do not consider the neighbors that are obstacles or in the region of the obstacle distance grid. This allows the algorithm to make safer paths which are not close to the obstacles. The obstacle distance is also incorporated into g-cost using the following equation.

$$gcost = gcost + (md - od)^{dce} \tag{25}$$

where *md* is a scaled value of the robot's radius, *od* is the obstacle distance of the current cell, and *dce* is the exponent to apply to the distance cost.

2) Exploration:

a) Frontier Exploration: Exploration in an unexplored map occurs in chronological stages. The algorithm is first initialized to check for system status; if all is good, the robot will begin exploring the map to travel to frontiers, which are unexplored regions of the map. This continues until all frontiers have been explored. At this stage, the robot will return to its home position and the exploration is considered complete. If there are regions left to explore but the robot is unable to find a safe path to reach the goal, or the robot cannot find a safe way to return to the home position then the exploration is considered to have failed.

During the exploration of the map, a list of frontiers within the map is found, and the closest frontier to the robot's current location is selected as a goal. For a region of a map to be considered a frontier, we require that the region spans a length of at least 30cm. The pose is determined to be at the center of the frontier. Once a goal pose is decided, the A* path finding algorithm developed earlier is used to find a path to the pose, and if it is safe, then the path is sent to the motion controller for the travel to be executed. If it is not safe, then the program increments the eight-way neighbors of this point until either a safe path is found, or neighbors of up to 20 cells away have all been checked. This process is repeated until no frontiers are found in the map.

b) Map Localization with Unknown Starting Point: When a robot is placed in an unknown position within a mapped environment, the SLAM system will initialize the sample of particles randomly all over the provided map with a high variance. Using the LiDAR scans, the

laser with the highest range is chosen and the robot is commanded to turn in the direction of the laser and move towards the endpoint to a distance of the range of the laser minus the radius of the robot and a safe distance threshold. At this point, the system checks if the farthest endpoint is too close to the robot, in which case it is determined that movement is not possible and the program is terminated. If the bot is limited to a specific area in the map, random nearby lasers are also evaluated to ensure that the robot is able to explore further.

As the robot moves towards the endpoint, the sensor model evaluates the likelihood of the hypothesis pose of the particles and assigns weights. Since the majority of the particles are outliers, we sort the posterior particles based on the weights and use the top 20% to estimate the mean posterior pose. This along with resampling of the particles using LVR significantly reduces the variance as the robot moves. If the robot reaches the endpoint but is unable to localize, the system checks for the laser with the highest range again and the algorithm continues as explained above. The variance of the particle distribution is evaluated at each SLAM iteration after the initial. If the variance is found to be less than 0.1 (which is the original variance we use in our initialization of the particles for the normal system), it is determined that the robot is localized and the normal motion planner and SLAM system is executed.

III. Results

A. Motion and Odometry

1) Motor Calibration: The gear ratio was calculated for three robots. All six motors had an encoder count of about 1520 per wheel revolution, and 20 per magnet revolution, which resulted in a gear ratio of 78:1.

The loaded steady state wheel speed vs. input PWM calibration program saves the resultant values, which are plotted to obtain the best fit equation, variance, and cut off values. This calibration equation is used later on in the controller. The equations are shown in Table II below and are represented in the plot in Fig. 5.

MBOT	Left Motor	Right Motor
24	y = 0.9557x + 0.0515	y = 0.9681x + 0.0545
08	y = 0.9428x + 0.0476	y = 0.9229x + 0.0556
95	y = 1.0327x + 0.0662	y = 1.0179x + 0.0664

TABLE II: Calibration equations for right and left motors of three different robots

A max standard deviation of 0.064 is observed, which is significant enough to require each bot to be tuned separately. Some of the variation is a result of the low-cost hardware (i.e. motors, batteries) used for the bots and the ambiguity in the robot assembly process (i.e. wheel base measurement).

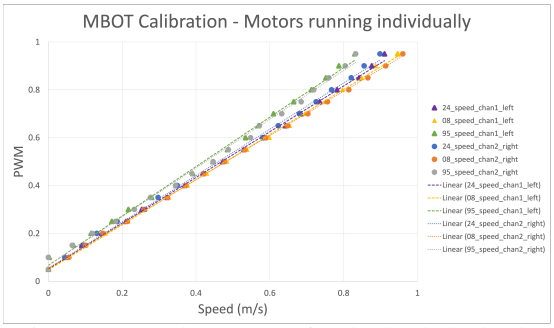

Fig. 5: Motor speed vs. PWM for the loaded wheels and fitted curve for both motors of three different robots

2) PID Controller: Feed Forward is a good starting point that helps take into consideration any discrepancy in the outputs, however, it is not sufficient as a standalone controller. The PID controller does really well on its own, especially because there are many parameters that could be tuned to improve its performance. Since this controller would be carried over through the rest of the project, it was critical to fully optimize its performance by combining the Feed Forward and PID controllers and adding additional features. The tuned values for the controller are listed in Table III below, and the performance of the bot using this controller is represented in Fig. 6.

Wheel Speed Open Loop & PID Controller	
P	5
I	3
D	0.0175
Left time constant	0.067
Right time constant	0.098
Low pass filter (Hz)	50

TABLE III: Final controller parameters and gains

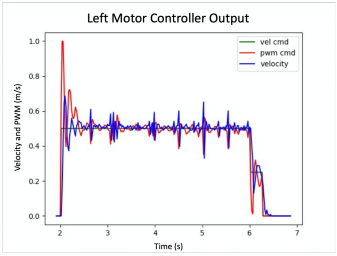

Fig. 6: MBOT24 actual velocity and PWM command

3) Odometry: The accuracy of the odometry model was determined with three manual tests that move the Mbot by known distances and turning by known angles, and the percentage error was calculated.

We manually moved the robot forward 0.50m and computed the updated x value. The robot was manually rotated by 90° and the updated θ was calculated. We manually rotated the robot by 90° and moved it forward by 0.40m to compute the change in y.

i) Move the robot forward 0.50 m

Parameter	Actual(m)	Computed(m)	% error
x	0.5	0.493	1.4%

TABLE IV: Test for accuracy of x position

ii) Rotate the robot by 1.5708 rad (90°)

Parameter	Actual(rad)	Computed(rad)	% error
θ	1.5708	1.556	0.94%

TABLE V: Test for accuracy of heading

iii) Rotate the robot 90°, then forward 0.40 m

Parameter	Actual(m)	Computed(m)	% error
y	0.4	0.394	1.5%

TABLE VI: Test for accuracy of y position

All three tests had low percentage errors (<2%), so the odometry model was determined to be satisfactorily accurate and no further calibration was performed.

4) Gyroscope and Odometry: The test over two surfaces was used to determine an optimal threshold for the gyrodometer, and the turn test was used to confirm performance. Testing was done for different thresholds in the direction that improves the gyrodometer reading, and end once the performance decreases. Testing began with a low threshold of 0.0005, and ended at 0.005. The optimal threshold is found to be 0.003; the process is shown in Tables VII and VIII.

Gyrodometer Threshold (rads)	Actual Rotation (rads)	Odometer Reading (rads)	Gyrodometer Readings (rads)
0.0005	0	0.013	0.040
0.001	0	0.015	0.016
0.002	0	0.013	0.003
0.003	0	0.011	0.001
0.005	0	0.016	0.03

TABLE VII: Multi-surface Testing

Method	Actual Rotation (rads)	Computed Rotation (rads)	Percentage Error (%)
Odometry	1.452	1.561	7.5%
Gyrodometry	1.452	1.426	1.8%

TABLE VIII: Rapid turn testing with Gyrodometry threshold of 0.003

5) Robot Frame Velocity Controller: The robot was tuned for the new feedback controller and the PID values are reported in Table IX. The result of the tuned left motor is provided as an example in Fig. 7.

To test the robot frame velocity controller, the Mbot was provided with a step input for forward velocity of 0.25m/s for 2s, 0.5m/s for 2s, and 1m/s for 1s. The robot's x position vs. time is plotted in Fig. 8.

The bot received a step input for turn velocity of $\frac{\pi}{8}$rad/s for 2s, $\frac{\pi}{2}$rad/s for 2s, and πrad/s for 2s. The robot frame heading vs. time is plotted in Fig. 9.

The Mbot's plotted performance matches the step input, so the robot frame velocity controller was validated.

Parameter	Value
P	0.75
I	25
D	0.0056
Time Constant	0.0825

TABLE IX: Gains for robot frame velocity controller

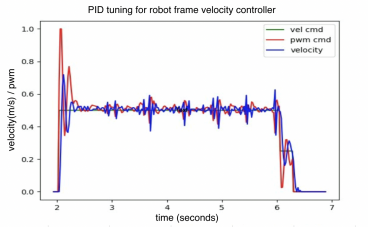

Fig. 7: PID tuning for robot frame velocity controller

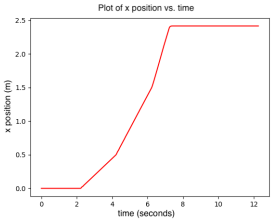

Fig. 8: Plot of x position vs. time

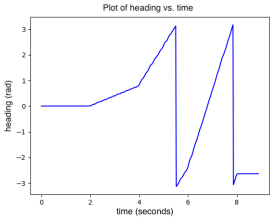

Fig. 9: Plot of θ vs. time

6) Motion Controller: Table X lists the gains used to run the bot through a path. The gains were determined using the stability conditions from (15) and differed for low speed (0.25m/s) vs. high speed (0.8m/s). The trajectory and velocity plots of the bot driving in a square path at a low speed are shown in Fig. 10 and 11.

Parameter/Gain	Low Speed	High Speed
Final-target-threshhold (m)	0.1	
Interval length (m)	0.01	
Velocity (m/s)	0.25	0.8
Target-reached-threshhold (m)	0.15	0.2
K_d	1.67	4.00
K_β	-0.60	-0.2
K_α	2.40	3.00

TABLE X: Motion controller parameters and gains

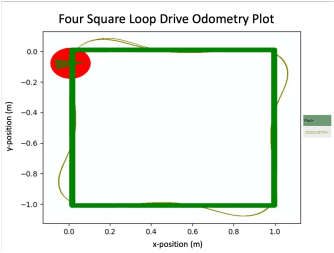

Fig. 10: Odometry path of robot driving around a square 4x and ending in its initial pose

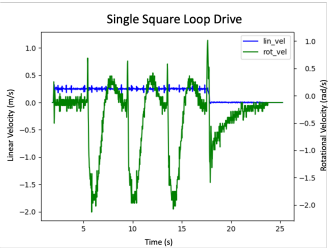

Fig. 11: Linear and rotational velocity as the robot drives one loop around a square

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: Fig. 12 shows the result of our updated mapping function.

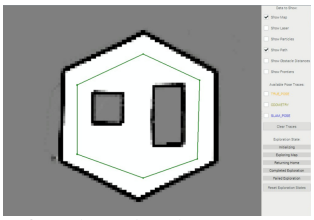

Fig. 12: Map for obstacle_slam_10mx10m_5cm.log.

2) Monte Carlo Localization:

a) Measurement Model: The values of the error coefficients were determined so that the distribution consistently overlays the true pose in the sample program provided, but not significantly more dispersed. The final values chosen allowed the solution and slam poses to be nearly identical in the sample trajectory provided when also running the sensor model. Further tuning was done when running tests in the real world environment, and the result is shown in Table XI.

Coefficient	Value
k_1	0.01
k_2	0.005
k_3	0.01
k_4	0.001

TABLE XI: SLAM Error Coefficients

No. of particles	Time(μs)
100	24583
300	40287
500	66212
1000	132647

TABLE XII: Time taken to update particle filter

b) Sensor Model and Particle Filter: Table XII shows the evaluation of time taken to update the particle filter for 100, 300, 500, and 1000 particles.

The total execution time of resampling particles, applying action and sensor model, and estimating the posterior pose was evaluated for different number of particles. This data was fit into a linear model to get:

$$y = 122.96x + 7524.63 \tag{26}$$

where y is time in μ s and x is the number of particles. At 10 Hz, the raspberry pi runs 1 cycle in 0.1s. Using 0.1s in equation x, the number of particles our SLAM system can support was evaluated to be 752.

The slam system was used on the log file drive_square_10mx10m_5cm.log with 300 particles. Seven different plots for the midpoint of every 1m translation and after the bot turns 90° were plotted and superimposed into Fig. 13.

Fig. 14 shows the SLAM vs odometry pose on drive square log with our SLAM system.

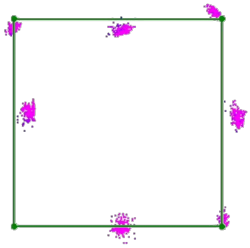

Fig. 13: Plot of particles on drive square log

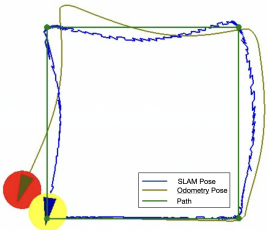

Fig. 14: SLAM vs. odometry pose on drive square log

3) Combined Implementation: To evaluate the accuracy of the SLAM system, it was used on obstacle_slam_10mx10m_5cm.log and the estimated SLAM pose was plotted against the ground-truth true pose as shown in Fig. 15. It can be observed that the SLAM

pose follows the true pose very accurately. The estimated SLAM pose was logged and the Root Mean Square (RMS) errors in x, y, and θ is reported in Table XIII. The mean RMS error is 0.149 which shows the SLAM system is highly accurate.

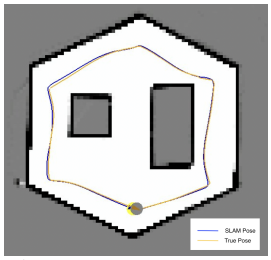

Fig. 15: SLAM Pose vs. true pose

Evaluation metric	Value
RMS error in x	0.073
RMS error in y	0.067
RMS error in θ	0.309
Mean RMS error	0.149

TABLE XIII: RMS error of SLAM pose vs. True pose

C. Planning and Exploration

1) Path Planning: After mapping the maze in the lab, we used BotGUI to make a path to the opposite end of the maze using A*, which the bot uses to drive home. The planned path vs. the actual path driven is shown in Fig. 16. In this case, A* uses 4-way neighbors to make the path and the planned path successfully takes into account the obstacle distance grids as the path avoids being close to the obstacles. The path is discretized in intervals of 1cm as specified in the motion controller designed for this Mbot.

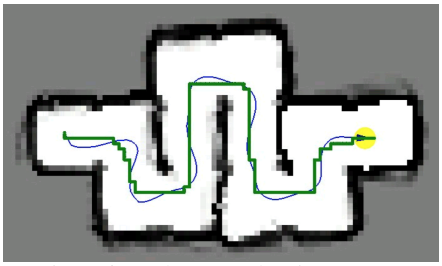

Fig. 16: Path created using A-star

We evaluated the execution time of the A* algorithm and report the statistics in Table XIV. During exploration, the Mbot frequently uses A* to create paths to the closest frontiers to ensure fast execution. In general, A* is able to make paths in less than 1s and hence determined to be fast. In the case of narrow constriction grid, there is no path possible in one of the test cases due to the narrow gap between the obstacles. In our implementation of A*, we iterate for a maximum of 10000 iterations before concluding that no path is possible and hence the execution time for narrow constriction grid is high. This can be improved in future implementations of A*.

Test grid	Mean (μs)	Max (μs)	Median (μs)	Std. dev (μs)
convex	2891	5554	2663	2663
empty	8524.3	15843	15843	5179.7
maze	54177.5	68465	37380	11070.9
narrow constrict	1.9e6	5.9e6	5.9e6	2.8e6
wide constrict	163378	481339	481339	224833

TABLE XIV: Path planning execution time

2) Exploration: When testing the algorithm using the same 2D map as the SLAM section, it was observed that the robot is capable of successful exploration, and returns home on multiple occasions. However, exploration failed after partial exploration for half of the tests conducted. When exploration fails, the robot typically crash into walls near the end of the unexplored map, and it was observed that the map would skew largely the further into the map the robot traveled. It was concluded that the SLAM results were not accurate enough for the smooth operation of the frontier exploration, causing the robot to falsely believe the path it planned was in the safe space of the map. To improve the functionality of exploration, localization must be improved. In addition, the bot can check in real time that the path planned is still safe with the current map generated, and if the path is no longer safe, then to stop travel, and create a new path.

IV. Discussion

In conclusion, this robot design adequately achieves all goals outlined in the report on concrete floors. Our gyrodometry based controller is able to successfully complete a path. Using the on-board LiDAR, the robot is able to map and localize any 2D map, but displays noticeable skewing. The robot is able to path plan quickly, and traverse an unexplored maze autonomously. Due to inaccuracy built up from localization and mapping, the robot encounters issues such as hitting obstacles during autonomous exploration.

As the overall performance is affected by the accuracy of every section, the algorithm can be improved by strengthening each of its features. Loop closure can improve localization inaccuracies, where the point cloud of the map is adjusted to fit with measurements from a previously visited location. Another solution for static environment only is to terminate mapping once a full enclosure is found.

References

[1] THORLABS. Driver pid settings. [Online]. Available: https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=9013

[2] D. Scaramuzza, R. Siegwart, and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*. The MIT Press, 2011.