ROB 550 Report
Team 3 (Afternoon)

Anirudh Aatresh Devansh Agrawal Shreya Phirke

***Abstract*—Autonomous robot operation is very important in a stand-alone cyber-physical system. A robot must be capable of moving through complex surroundings to explore and get to its destination. In this report, we consider the task of designing a robotic system that is capable of performing control tasks, localization and mapping, and exploration. We study various scenarios where the robot needs to consider sensor feedback to achieve a particular goal. For robust velocity and position control, we build low-level and high-level closed-loop controllers that are capable of achieving this. Further, we build a framework to perform simultaneous localization and mapping using LIDAR scan data, that is further processed to perform reliable planning and exploration. We demonstrate the working of our algorithms in practice by running them in real-time on the MBot for a variety of tasks and environments.**

I. Introduction

NUMEROUS industrial applications require robots that can work autonomously. Incorporating autonomous movement into a robot's design is crucial to building systems that can operate without human intervention. Giving a robot high-level commands such as going to restaurant 'A' and returning with some food should be sufficient information for the bot to get to the restaurant safely and back, provided it knows where the target locations are. While this sounds like an interesting prospect, several unforeseen factors affect the planning and control of the bot, and how the robot performs the tasks assigned to it.

Sensors are the robot's sense organs (analogous to the sense organs of humans) and provide it crucial information about what is happening in its environment and sometimes about how the environment is reacting to its actions. This information is used as a form of feedback for the robot to understand the consequence of its actions and if additional corrections must be made to achieve its goal. This form of controlling the robot using feedback is known as closed-loop control and finds a wide variety of applications in control systems. The opposite of closed-loop control is open-loop control where it is assumed that no corrections will be needed once actuated.

A sensor that is of particular interest to us is the LIDAR (Laser IDentification and Ranging) module that is capable of producing representations of its environment with sufficient detail for a robot to use for navigation. We use the 2 dimensional version of it to build a map of the robot's surroundings, which it uses to first identify its pose within the map and further to traverse through it. This entire process can be termed as Simultaneous Localization and Mapping (SLAM).

While SLAM provides the robot information about what is present in its surroundings, it must be told what path it must take within this constructed map. This is where planning algorithms come in such as A-star and rapidly exploring random trees (RRT). Using these algorithms a robot can effectively map its surroundings and move to various locations of the map accurately at the given velocity. Moreover, it can self identify its position based on its surroundings and perform exploration tasks if need be.

In this report, we detail the construction of this system, wherein we develop a robot that is capable of performing mapping and planning while also adhering to low-level set-points such as velocity and position control. Section II covers concepts and implementation of motion and odometry, section III talks about our SLAM implementation, section IV details algorithms in planning and autonomous exploration and section V concludes this report.

II. Motion and Odometry

A. Characterizing the Wheel Speed

Characterizing the speed of the robot is critical in designing an open-loop controller. This is because the control signal to the motors from the BeagleBone Blue (BBB) is a pulse width modulation (PWM) command that is in the range of $[-1, 1]$. We have to convert PWM signals into wheel velocities for the open-loop controller. This can be represented mathematically as:

$$v_{\text{left}} = f_{\text{left}}(p_{\text{left}}) \tag{1}$$

$$v_{\text{right}} = f_{\text{right}}(p_{\text{right}}) \tag{2}$$

where $v_{\{.\}}$ represents the wheel velocity, $p_{\{.\}}$ is the PWM control signal given to a particular wheel and $f_{\{.\}}$ is the function that describes their conversion. In our experiments, we have collected data to estimate f. More specifically, for each wheel, we calculate the loaded wheel velocity for an input PWM control signal, which is varied from -1 to 1 at a resolution of 0.05. Further, we fit a piecewise linear curve to this data that is used to approximate f. The data points collected and corresponding curve fit has been shown in figure 1.

It can be seen that the linear curves for each of our robots are piecewise in nature with symmetry about the

Y-axis. Moreover, these curves exhibit some distinct and unique features. The slopes (m) and Y-intercepts (c) of each of the curves ($y = mx + c$) represent the behavior of each motor. This comparison can be made between two motors on the same robot and between two motors each mounted on a different robot. When the data is plotted as PWM values against speed, the slopes differ by ~ 0.3 and Y-intercepts by ~ 0.05. The sources of these deviations include variations in characteristics of each motor and encoder. The equations obtained from this curve fitting were further used to design an open-loop velocity controller.

Using the collected data, we created a map between desired wheel speeds and required PWM command. To account for deadband, the following relationship is used

$$p_{\text{cmd}} = f^{-1}(v_{\text{des}}) = \begin{cases} 0 & \text{if } v_{\text{des}} = 0 \\ m_1 v_{\text{des}} + c_1 & \text{if } v_{\text{des}} > 0 \\ m_2 v_{\text{des}} + c_2 & \text{if } v_{\text{des}} < 0 \end{cases} \quad (3)$$

where m_1, m_2, c_1, c_2 are parameters we calibrated by least-squares fitting.

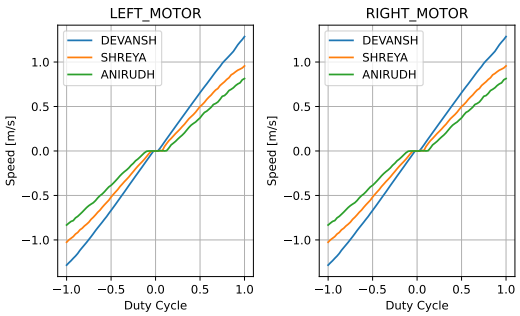

Fig. 1. Calibration curves for our robots.

TABLE I
LEFT AND RIGHT MOTOR CALIBRATIONS

User	Left Motor Calibration	Right Motor Calibration
Anirudh	$1.02867(v) + 0.09117$	$1.1124(v) + 0.08181$
Devansh	$0.72441(v) + 0.04711$	$0.76522(v) + 0.00550$
Shreya	$1.06783(v) + 0.03394$	$1.07096(v) + 0.01308$

B. Implementing the Open-Loop and Closed-Loop Control

- **Open-loop controller design:** To have our robot move at a particular velocity using an open-loop controller, we use the curve obtained from the previous task to invert the transition function f to find corresponding PWM values from setpoint velocities. It was observed that the robot completed a skewed square when it was directed through the open-loop controller to complete a perfect square. We attribute this behavior to the inherent imperfections of the surface and motors that can cause deviation in the velocities.
- **Closed-loop controller design:** For the closed-loop controller, we implemented a PID control loop that performs a proportional - integral - derivative computation on the error between the setpoint and actual velocities, with a feedforward term based on the open-loop controller. In our experiments, the PID controller was tuned by choosing a convenient setpoint and gradually varying k_p until small oscillations were observed. k_d was then increased to reduce these oscillations after which the k_i value was increased until the steady-state error approached zero. We assumed the PID controller was tuned when the steady-state value was within a particular permissible error band around the setpoint value.
  We modify the conventional PID framework by adding low pass filters to filter out noise in encoder readings and outputs from the closed-loop controller. Furthermore, we created a parallel PD controller and I controller flow for a more simple resetting mechanism of the I controller. This allows us to control the characteristics of the step response. The PID parameters have been shown in the table II.

TABLE II
PARAMETERS FOR THE PID CONTROLLER AND LOW PASS FILTERS FOR WHEEL VELOCITY CONTROL

k_p	2.0
k_i	0.05
k_d	0.05
integrator limit	1.0
time step of LP filters	0.02 seconds
time constant of LP filters	0.5 seconds

C. Odometry

Odometry can be used to localize the robot and characterize its pose using position and orientation calculations. In our case, we found the robot's position in (x, y) coordinates and its heading θ.

If we assume that the left wheel of the robot has moved a distance Δs_L and right wheel has moved a distance Δs_R such that $\Delta s_L \leq \Delta s_R$, we can express them as:

$$\Delta s = \frac{\Delta s_L + \Delta s_R}{2};\ \alpha = \frac{\Delta s_L - \Delta s_R}{b} \quad (4)$$

where the R is the radius of rotation and $\alpha = \Delta\theta$ is the angle of arc swept by the robot. For short distances of movement, we can approximate the arc length to straight line joining initial and final positions. The change in x and y coordinates can be found as:

$$\Delta x = \Delta s \cos(\theta + \frac{\Delta\theta}{2});\ \Delta y = \Delta s \sin(\theta + \frac{\Delta\theta}{2}) \quad (5)$$

Hence, the new pose can be expressed as:

$$[x',\ y',\ \theta']^T = [x,\ y,\ \theta]^T + [\Delta x,\ \Delta y,\ \Delta\theta]^T \quad (6)$$

The odometry model was validated by checking the distance traveled along X and Y directions along a ruler and heading using a protractor. Our odometry model was found to be accurate enough and no correction parameters were applied. However, this model was checked for corrections once and not repeatedly in the future.

D. Gyro Sensor Fusion

The gyrodometry algorithm proposed by Borenstein et al. in [1] proves that fusing odometry and gyro data can help alleviate non-systematic errors produced due to sudden bumps or disturbances on the surface. When such an event is encountered, there is a clear difference in the estimated angle values by the gyroscope and that through odometry, thereby producing an error in measurement. The gyrodometry algorithm used to reduce this error has been shown in algorithm 1.

Algorithm 1: Gyrodometry algorithm.

Data: Initialize θ_{thresh}
At every step of computing odometry values,
  Compute $\Delta\theta_{odo}$ from the odometry calculations.
  Compute $\Delta\theta_{gyro}$ from consecutive readings of tait-bryan angles.
  $\theta_{G-O} = \Delta\theta_{gyro} - \Delta\theta_{odo}$
  if $|\theta_{G-O}| > \theta_{thresh}$ **then**
    $\theta' = \theta + \Delta\theta_{gyro}$
  else
    $\theta' = \theta + \Delta\theta_{odo}$
  end

The parameters used in our odometry and gyrodometry computation are the wheel base distance b and the θ_{thresh}. The wheel base distance was measured using a tape measure to be 0.1584 m. θ_{thresh} was estimated by performing the experiments mentioned in [1]. We used an insulated 5 mm diameter cable as a bump to find an appropriate θ_{thresh}. Our experiments showed that the value (0.105 rad) mentioned in [1] also worked well for us.

E. Robot Frame Velocity Controller

The velocity controller in the robot frame was designed such that input commands for forward and turn velocity are used to compute the left and right wheel velocities. Given a commanded transnational velocity v, and commanded angular velocity ω, we computed the desired speeds (technically the ground speed of the wheel if the robot were travelling straight) of the left and right wheels as

$$v_R = v + w\frac{b}{2},\ v_L = v - w\frac{b}{2} \tag{7}$$

where $v_{[.]}$ is the speed of each wheel, and b is the wheel base distance.

We noticed that when the commanded translational velocity is changed rapidly, the wheel tended to slip on the ground, which meant that the encoder readings were thrown off. To avoid this, we introduced a first-order low-pass filter on the translational velocity commands, with a time constant of 0.5 seconds. We decided not to implement this for the angular speed, as wheel slip was not as significant of an issue, and for the controller to converge to the desired headings, fast response on the angular rate controller was desirable.

Furthermore, for precise positioning, it was desirable for the robot to stop rapidly when commanded to do so. As such, if the commanded speed was lower than the filtered output, we short-circuited the filter and send the smaller velocities directly. The parameters that were used for the controller has been shown in table III.

TABLE III
PARAMETERS OF THE PID ROBOT FRAME VELOCITY CONTROLLER.

k_p of turn controller	0.7
k_i of fwd and turn controller	0.0
k_d of fwd and turn controller	0.0
time step of LP filters	0.02 seconds
time constant of LP filters	0.5 seconds

F. Motion Controller

In this section, we implemented a motion controller that combined all the information received from previous sections. We designed a motion controller that will make our robot move through a series of waypoints at a given setpoint forward and turn velocity.

The motion controller is implemented using a state-machine, with three states: IDLE, TURN, STRAIGHT. The robot is initialised in IDLE state, where the commanded $v, \omega = 0$. When a new message is received, it is first parsed, and stored in memory. Then, the robot enters the TURN and orientates itself to point in the direction of the next waypoint. Once the angle is within 0.05 rad, the controller switches to the STRAIGHT state. This controller drives at ω_{max}, and then switches to a proportional controller to smoothly stop at a desired heading.

Despite many attempts at tuning it was difficult to achieve a precise turn controller, especially at high rotation speeds. Somewhat surprisingly, this was primarily due to the robots inability to turn at low-speeds - the deadband this introduced made it difficult to precisely calibrate the proportional turn controller. While better modelling of the robots turning capabilities, and smarter controllers, might be able to produce a good turning controller, we were able to achieve desired performance using the straight controller, described next.

Once the turn controller exits, the robot switches to the STRAIGHT mode, where from an current state (x, y, θ), the robot is targetting to reach the position (x_T, y_T). The

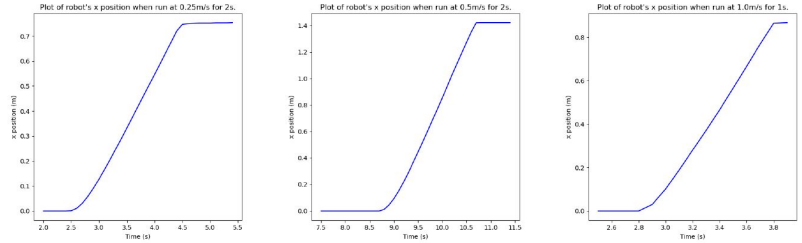

Fig. 2. Plots for 0.25m/s (left), 0.5 m/s (middle), 1m/s (right).

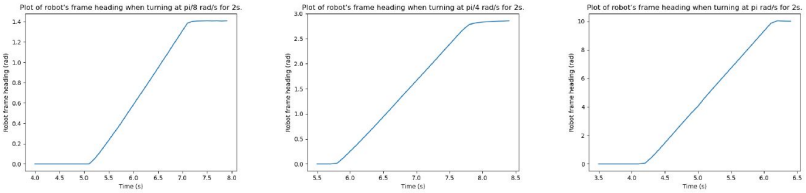

Fig. 3. Plots for $\pi/8$ rad/s (left), $\pi/4$ rad/s (middle), π rad/s (right).

velocity commands were computed as follows:

$$\theta_T = \arctan\left(\frac{y_T - y}{x_T - x}\right) \tag{8}$$

$$r = \sqrt{(x - x_T)^2 + (y - y_T)^2} \tag{9}$$

$$r_{sat} = \min(r, 1.0 \text{ m}) \tag{10}$$

$$v_{\text{cmd}} = r_{sat} v_{max} \cos^2(\theta - \theta_T) \tag{11}$$

$$\omega_{\text{cmd}} = -k_\omega(\theta - \theta_T) \tag{12}$$

In words, the translational velocity is a proportional to the distance to go, but is saturated to v_{max} and reduced (by a factor of $\cos^2$) if the angle between the current heading the target position is large. The angular velocity is proportional to the heading error. We used $k_\omega = 2.0$. It is not difficult to show that this controller is asymptotically stable to the desired position, assuming $|\theta - \theta_T| < \pi$. At the higher translational speeds, we also introduced a trapezoidal acceleration curve, to avoid initial wheel slippage. The STRAIGHT mode is exited when the position error satisfies $|x - x_T| \leq R$ and $|y - y_T| \leq R$, where we set $R = 10$ cm. If an additional waypoint is available, the TURN mode is selected, else the IDLE mode is selected.

A plot of our robot's dead reckoning estimated pose as it attempts to make a square of side 1 m is shown in figure 4. A plot of the robot's linear and rotational velocity as it drives one loop around the square can be seen in figure 5. It is clear that the robot does not return exactly to the starting point after four revolutions and is off by about 15 cm. Additional sensors or improved odometry methods would be required to improve this. In figure 4 the robot starts at (0,0). The red dashed square indicates the reference path, and the green squares denote the tolerance for the STRAIGHT controller mode.

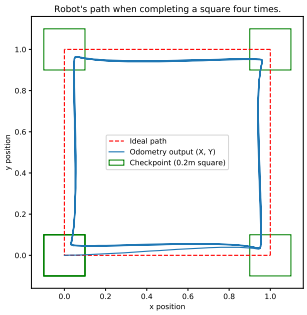

Fig. 4. Trajectory for four revolutions around the square.

III. SLAM Implementation

A. Mapping and Occupancy Grid

We divide the observable 2D space into a grid for convenience and denote the occupancy of each cell through some measurement. To convey how certain we are about the occupancy of a cell, we denote the probability that a cell is occupied by $p(occ(x, y))$ and probability that a cell is free by $p(\neg\, occ(x, y))$. The log odds of the occupancy of a cell can be expressed as:

$$\log\left(o(occ(i,j))\right) = \log\left(\frac{p(occ(i,j))}{p(\neg\, occ(i,j))}\right) \in [-\infty, \infty] \tag{13}$$

The map constructed by our implementation when run on `obstacle_slam_10mx10m_5cm.log` has been shown in figure 6.

B. Monte-Carlo Localization – Action Model

The state of our robot at any time t can be described as a 3-dimensional vector, $x_t = [x,\, y,\, \theta]^T$. This state changes as various inputs are applied to the robot. The action

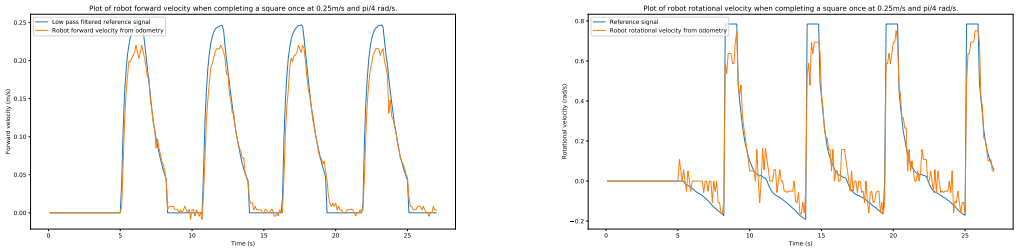

Fig. 5. Plot of the robot's linear and rotational velocity as it drives one loop around the square.

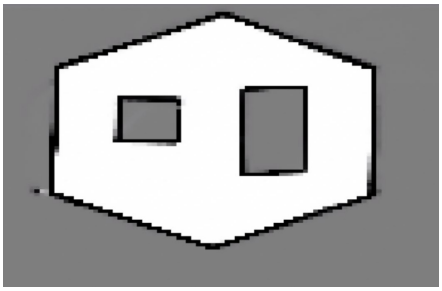

Fig. 6. Mapping on obstacle_slam_10mx10m_5cm.log.

model of the robot describes the state transition function of the system, describing how the system transitions to a new state given its current state and input signal. The probability of transitioning to state x_t from x_{t-1} and control input u_t is $p(x_t|x_{t-1}, u_t)$.

We have used the odometry action model to calculate this distribution. In this model, the odometry output is used as the input control signal. The path to be taken to the destination point is described through an initial rotation (rot1), a translation (trans) and a final rotation (rot2). Through the odometry model, we can find how much the robot must rotate and translate by calculating the error for each of these quantities. This calculation has been shown in eqs. (14) - (15).

$$\delta_{\text{rot1}} = \text{atan2}(\text{dy}, \text{dx}) - \theta,\ \delta_{\text{trans}} = \sqrt{\text{dx}^2 + \text{dy}^2} \tag{14}$$

$$\delta_{\text{rot2}} = \text{d}\theta - \delta_{\text{rot1}} \tag{15}$$

where, dx, dy and dθ are the errors in pose from initial pose $[x,\ y,\ \theta]^T$.

We then sample new data for each particle from a normal distribution centered around δ_{rot1}, δ_{trans} and δ_{rot2} and control uncertainty using parameters $k1$ and $k2$.

$$\hat{\delta_{\text{rot1}}} \sim \mathcal{N}(\delta_{\text{rot1}}, \sqrt{k1 * \text{rot1}}) \tag{16}$$

$$\hat{\delta_{\text{trans}}} \sim \mathcal{N}(\delta_{\text{trans}}, \sqrt{k2 * \text{trans}}) \tag{17}$$

$$\hat{\delta_{\text{rot2}}} \sim \mathcal{N}(\delta_{\text{rot2}}, \sqrt{k1 * \text{rot2}}) \tag{18}$$

TABLE IV
UNCERTAINTY PARAMETERS USED IN OUR ACTION MODEL.

Uncertainty parameter	Value
$k1$	0.01
$k2$	0.01

We can now find new potentially feasible poses by computing the x, y and θ by the following equation:

$$[x_{\text{new}}, y_{\text{new}}, \theta_{\text{new}}]^T = [x_{\text{old}}, y_{\text{old}}, \theta_{\text{old}}]^T + [\hat{\delta}_{\text{trans}} \cos(\theta_{\text{old}} + \hat{\delta}_{\text{rot1}}), \hat{\delta}_{\text{trans}} \sin(\theta_{\text{old}} + \hat{\delta}_{\text{rot1}}), \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}]^T$$

In essence, this amounts to sampling poses from the distribution $p(x_t|x_{t-1}, u_t)$. The uncertainty parameters $k1$ and $k2$, as shown in table IV, were chosen by looking at the spread of the particles on a plotted map. $k1$ was varied to adjust the amount of spread of particles in the heading angles. This would amount to adjusting the arc length of the spread. $k2$ was varied to adjust the the longitudinal spread of the particles.

C. Monte-Carlo Localization – Sensor Model and Particle Filters

The sensor model describes what the sensor measurement will look like from the current state. More specifically, the sensor model produces a probability distribution over various sensor measurements given a particular state and map, described by $p(z_t|x_t, m)$.

Table V shows the average time taken by our implementation to update the particle filter for different number of particles. The collected readings were averaged across multiple trials for the same number of particles to give a less noisy estimate.

TABLE V
TIME TAKEN TO UPDATE THE PARTICLE FILTER

Number of particles	Avg. Time taken (s)
100	0.0342
300	0.0722
500	0.0950
1000	0.1874

Considering practicality, the maximum number of particles that can be handled by our implementation is $20,000$.

TABLE VI
ERROR STATISTICS WHEN COMPARING ESTIMATED SLAM POSES AND GROUND TRUTH POSES FOR 50 AND 200 PARTICLES.

	50 particles	200 particles
Max Error x	0.2906	0.4526
Max Error y	0.3898	0.5815
Max Error θ	1.2127	1.2329
RMSE x	0.1192	0.1911
RMSE y	0.1743	0.2526
RMSE θ	0.1314	0.1928

Our method was able to progress through its update phase, but was too slow for all practical purposes, requiring approximately 2 seconds for each update. Considering our motion controller is updated at 50 Hz, or 20 ms, even with 100 particles, our particle filter is too slow. In our experiments, we noticed a substantial improvement in tracking performance, when we used 50 particles, instead of the default of 200.

The plot of 300 particles at each half length of the square and after each right angle turn when run on `drive_square_10mx10m_5cm.log` has been shown in figure 7.

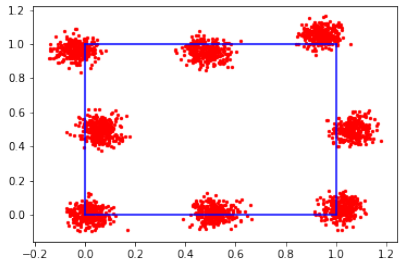

Fig. 7. Plot of 300 particles on drive_square_10mx10m_5cm.log.

D. Simultaneous Localization and Mapping (SLAM)

The plot showing the pose error difference between odometry and SLAM poses for x, y and θ components can be found in figure 8. The block diagram of our SLAM implementation has been shown in figure 9. The error statistics and comparisons between the estimated SLAM poses and ground truth poses can be found in figure 10 and table VI.

IV. PLANNING AND EXPLORATION

A. A-star Path Planning

Our A-star path planner follows the standard A-Star algorithm. In an attempt to achieve better performance (faster computation times), our implementation used the following:

- At the start of the A-Star search, the start and target grid nodes are found, and the entire search only uses the grid cells, which are stored as `int` rather than `float` or `double`. This avoids expensive floating point operations, and helps eliminate floating point rounding errors.
- The open nodes list is implemented using a C++ `std::priority_queue`, and therefore expensive sorting and searching operations are avoided.

In addition, we implemented a path simplification procedure: instead of generating individual waypoints at every node along a straight line trajectory, this procedure simplifies the path so only the corners of a trajectory are returned in the `robot_path_t` struct. This simplification was very useful, since our motion controller was tuned to driving to a distant target, rather than a neighboring cell.

That said, due to a lack of time, we were not able to implement the following features which we suspect would also increase performance: improved memory allocation and diagonal path planning.

The code's run-times are listed in the table VII. We can see that the computation times depend significantly on the test case, and the mean computation time ranges from about 4 ms to 15 ms, but were as large as 60 ms on the empty grid. I believe the main reason for this was large size of the empty grid, compared to the other cases. This makes the memory allocation much larger, slowing down the code.

TABLE VII
COMPUTATION TIMES FOR A-STAR PATH PLANNING. ALL COMPUTATION TIMES ARE ROUNDED TO THE NEAREST MICRO-SECOND, AND REPRESENT THE RESULT AFTER RUNNING THE A-STAR PATH PLANNING WITH 100 REPEATS. WHERE IT SAYS NA IS THE TESTS WHERE NO SUCH CASES OCCURRED.

Successful Cases (μs)	Min	Mean	Max	Median	Std dev
convex_grid	3371	3953	10700	3471	1271
empty_grid	8127	14747	60657	13456	8785
filled_grid	NA	NA	NA	NA	
maze_grid	3508	4193	10470	3626	1304
narrow_constriction_grid	8146	9640	19411	9447	1861
wide_constriction_grid	8074	9401	20861	9060	1958

Failed Cases (μs)	Min	Mean	Max	Median	Std dev
convex_grid	2	1108	5941	890	1421
empty_grid	0	2392	11056	3088	2662
filled_grid	1	4	177	2	11
maze_grid	NA	NA	NA	NA	NA
narrow_constriction_grid	2	9325	67565	11860	10377
wide_constriction_grid	2	4	128	2	13

B. Map Exploration

The exploration algorithm that we have used is based on frontier identification, and planning towards a selected frontier. This algorithm is iterated on repeat until no more frontiers can be detected. These steps can be described in detail as:

- Frontiers are denoted as those cells in the occupancy grid that border areas that are known to be free, and areas whose occupancy is unknown. The breadth first search (BFS) algorithm is used to iterate through all

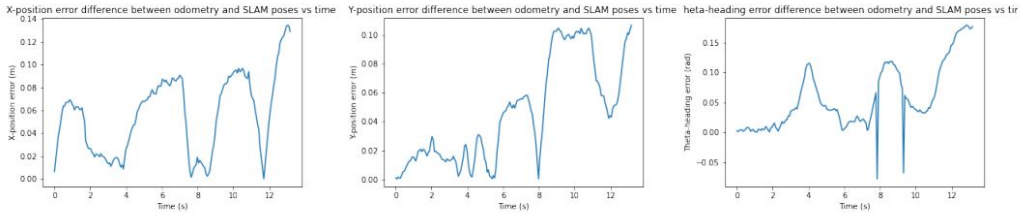

Fig. 8. Pose error between odometry and SLAM poses. The green line shows the desired path, the blue lines shows the SLAM estimate, and the light-green shows the odometry estimate.

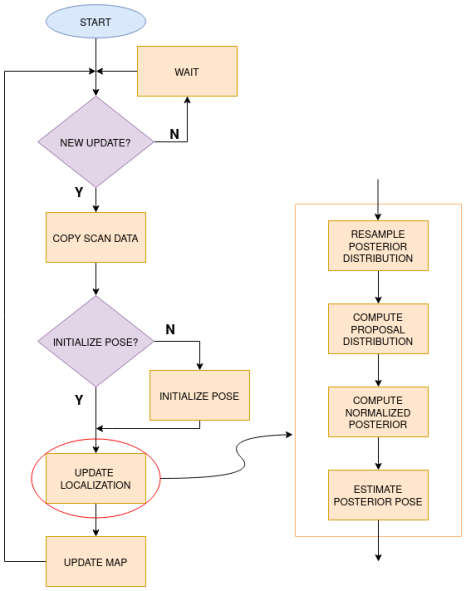

Fig. 9. SLAM block diagram.

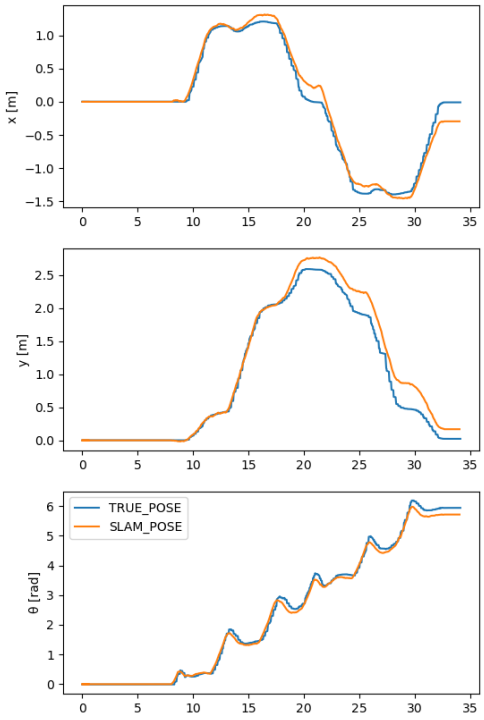

Fig. 10. Comparison of ground truth poses and slam pose estimates with respect to time (s) for 50 particles.

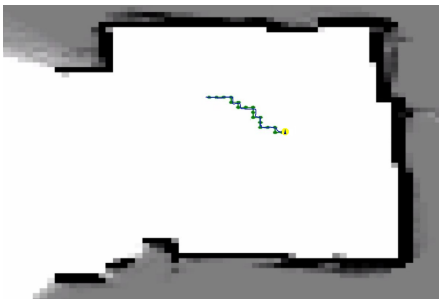

Fig. 11. Planned Path (green) and Actual Path (blue)

connected free space cells. Through 4-way connectivity, each neighbor is checked and frontiers are grown until they are of minimum length.

- Once all frontiers have been found for the current robot pose, we consider a valid target position lying on the straight line between our robot's current position and any reachable frontier cell on that frontier. We consider the target position to lie on a fraction $\alpha \sim 0.8$ of that straight line. The target heading was maintained to be the same as the robot's current heading.
- As the robot attempts to move towards this target position, new scan information may cause the frontier to either shift or get deleted (as new scan data can update the log-odds of the frontier cells). Hence, this dynamically updates the target position that must be reached as the frontier that is being considered will be constantly updated.
- At some point of time, there might come a stage when there are no new frontiers available on the map. This corresponds to the situation when the entire arena has been explored. At this stage, the robot is instructed to go back to its starting point, also known as *returning home*. This algorithm was implemented in `frontiers.cpp`

and `exploration.cpp`. The heart of the implementation consists of a state machine that goes through states such as `STATE_INITIALIZING`, `STATE_EXPLORING_MAP`, `STATE_RETURNING_HOME` and `STATE_COMPLETED_EXPLORATION`. Our method worked well for certain cases, but faced difficulty when there were frontiers lying in parts of the occupancy grid which were unknown (0 log-odds). This could be solved by modifying our algorithm to calculate the target position to lie in the nearest free cell that the robot can reach. Moreover, the algorithm can be made more efficient by selecting the frontier that is closest to the robot's current position instead of choosing the first frontier with a valid path to it.

C. Map Exploration with Unknown Starting Pose

We considered a number of methods to perform self-localization on a map, considering factors including computation cost, capability, edge cases, and robustness against slightly incorrect maps. In the end we decided on a fairly simple algorithm:

1) We create $4n^2$ particles, where n is an integer.
2) We determine the limits of the free space in the provided map (instead of the full grid) using a simple min/max search. This gives us x_{min}, x_{max} and y_{min}, y_{max}.
3) The rectangular domain $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ is uniformly gridded, and four particles are placed at each grid point. Each of these four particles are initialised with headings that are 90 degrees apart. I.e., at each grid point, we initialise four particles, facing North, South, East, West
4) Now the standard particle-filter based update is started, and the robot starts acting with the following sequence of actions, each lasting a set duration.
    a) Spin clockwise for T_1 seconds
    b) If it is safe to do so, move forwards forwards for T_2 seconds, else continue spinning until it is safe to move forwards.
    c) Repeat until all particles are within a small region of each other, i.e., the particle filter has converged.
5) Exit.

Here "safe to move forwards" is defined as requiring the LIDAR to read distances greater than L meters within the 45 degrees of the straight direction.

The main benefits of this approach are:

- The motion controller is decoupled from the map that is being generated, and more importantly, is decoupled from the individual particle's poses: as such, even if the particles are all wrong, the robot will not crash into walls (at least in theory)
- By placing four particles in each grid point, and using a uniform grid of points, we are distributing our initial guess of the robot pose very widely, and uniformly. As opposed to random sampling, this avoids situations where the samples do not cover a region of the map sufficiently.
- The robots path will never get stuck at one spot, since it can always turn around enough to return on the path that it came from. It can get caught in a loop/limit cycle, (for instance bouncing between walls) but by randomising T_1, T_2, this can be avoided.

We had some challenges in our implementation, and while the above algorithm sometimes worked, it was not very reliable. By inspecting the behaviour of the particles, it seems the particles would very quickly (in about 3 or 4 iterations) down select to a single sub-region as the most likely state of the robot. As such, the true state of the robot is missed, and the algorithm fails to localise the robot correctly.

This probably could be corrected by (A) increasing n, (B) reducing the rate of convergence towards the best estimate particle, and (C) improving the LIDAR-match scoring algorithm to be faster and to use more of the information that is available in the LIDAR scans, (D) increasing the variance when re-sampling the particles or (E) intentionally adding outliers when re-sampling, which would help move the particle filters out of the local-minima.

Furthermore, at the moment we place particles at a uniform grid. One alternative could be to use a triangulation of the free space, and place guesses at the center-points of the triangles. This would avoid placing initial particles inside obstacles, which are obviously false placements, and simply are a form of wasted particle. Another potential extension is to use a wall-following type controller, instead of the predefined sequence of movements.

V. Conclusion

In this report, we study controller design, SLAM, and planning algorithms for the MBot and show them working successfully in real-time and in different environments. Through our experiments, we observe how open-loop control can be inferior to closed-loop control and use this observation to create controllers for our robot. Further, we use our implementation of the simultaneous localization and mapping (SLAM) algorithm to plan a path and guide our robot through an environment. Our experiments and observations have given us valuable insight into how these algorithms can be implemented on embedded devices to accomplish a particular task.

Although our implementation is capable of completing the task at hand, we believe that there is further scope to improve these algorithms. Future work on this project would include improving the efficiency of our methods and making them applicable for a wider variety of test cases.

Link to requirements.csv file: here

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, pp. 423–428 vol.1, 1996.

[2] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

APPENDIX

A. Problem Statement

In an open-loop control system, a robot or a plant operates without any real-time update of the motion of the plant. Since there is no form of feedback, the robot is unable to correct for disturbances, miscalibrations, errors introduced by discretization and various nonlinearities. There can be several uncertainties and disturbances in a real-world application that can cause errors in the operation, even when the logic might be correct. These make open-loop control impractical to operate anything other than simple applications or projects.That said, open-loop control has been used in some incredible applications: the Space Shuttle's launch trajectory was an open-loop controller: nonlinearities introduced by atmospheric drag made closed-loop control mathematically and computationally challenging with the technology available at the time. The main flaw of this type of control is that there is no update in the algorithm if there are errors in the motion of the robot.

When we use some kind of feedback to understand how well the robot is doing in its environment, it's called a closed-loop control system. A closed-loop controller can take corrective action if the plant's actual output deviates from its setpoint. This makes the objectives more achievable by taking into account the systematic errors and feedback from the sensors.

In our experiments, we have compared the performance of the two control systems. The task aimed to maintain a constant velocity while traversing a square trajectory. We have shown the results of these experiments in the following section. Furthermore, we have used closed-loop controllers for position control to preemptively change velocity as the target pose is reached.

B. Motor Calibration

The following relationship is used

$$p_{\text{cmd}} = f^{-1}(v_{\text{des}}) = \begin{cases} 0 & \text{if } v_{\text{des}} = 0 \\ m_1 v_{\text{des}} + c_1 & \text{if } v_{\text{des}} > 0 \\ m_2 v_{\text{des}} + c_2 & \text{if } v_{\text{des}} < 0 \end{cases} \tag{19}$$

where m_1, m_2, c_1, c_2 are parameters we calibrated by least-squares fitting. These parameters were saved into user-specific config files. Since these four parameters were needed for each wheel, each user had unique 8 parameters. The PWM commands were also saturated to lie between -1 and 1. The calibration was performed for both left and right motors. The positive and negative slope values were found to be very similar to each other.

While a better fit would be possible using quadratics or higher order polynomials, with the range of uncertainty and other errors, we felt that the additional complexity would not significantly improve performance.

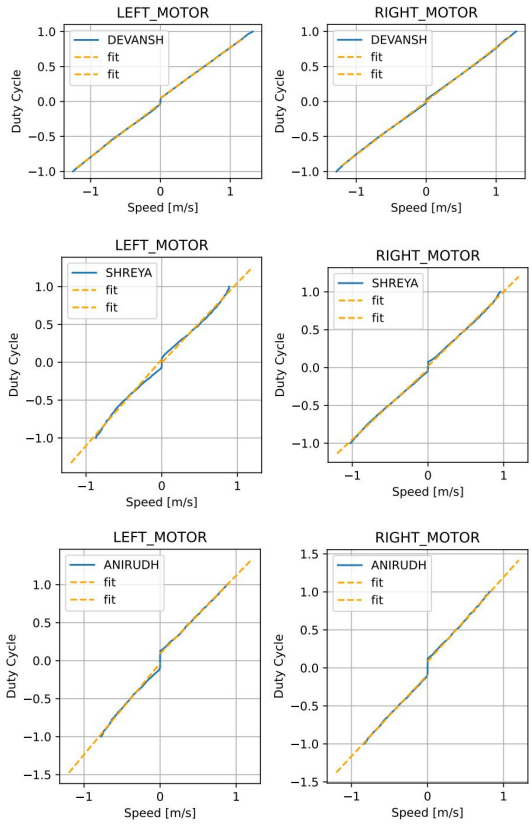

Fig. 12. Data points and respective curve fits for each of our robots. Devansh's (left), Shreya's (middle) and Anirudh's (right).

C. PID Control

The proportional controller gives the robot the kick it needs to approach the setpoint, and its extent is controlled using coefficient k_p. The integral controller is highly effective in reducing steady-state errors that often occur during PID operation. The differential controller helps avoid transients in the error signal, thereby reducing the oscillations.

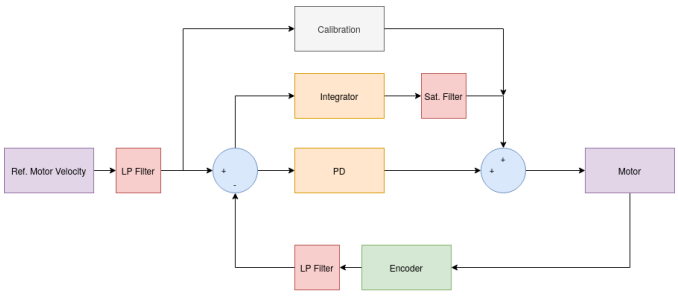

Fig. 13. Block diagram of PID controller implemented.

D. Mapping

In our experiments, we have followed the standard flow of building a preliminary map of the surroundings and

subsequent cycles of localizing within the map and updating the map. These steps lead us to an estimation of the MBOT's surroundings and its location based on LIDAR scans using SLAM. To convey how certain we are about the occupancy of a cell, we denote the probability that a cell is occupied by $p(occ(x, y)) \in [0, 1]$ and probability that a cell is free by $p(\neg\, occ(x, y)) \in [0, 1]$. The occupancy of a cell can be expressed as:

$$o(occ(i, j)) = \frac{p(occ(i, j))}{p(\neg\, occ(i, j))} \in [0, \infty] \tag{20}$$

These probability values can at times become really small and border on the precision of the system. To avoid this problem, we consider the logarithm of this value that also simplifies potential multiplicative operations. This formulation can be expressed as $\log o(occ(i, j)) \in [-\infty, \infty]$. To build a map of the surroundings, we construct an occupancy grid in which each cell represents the log-odds of that cell being occupied. Each scan provides crucial data about the distance traversed by each emitted ray. We look how far each ray has travelled and compute its end point in world frame with the knowledge of its originating point and range. This gives us the input required to compute the cells through which the ray has passed in our occupancy grid. We have used Breshenham's algorithm [2] for this purpose. This algorithm's popularity can be attributed to the fact that it requires only integer arithmetic operations. Breshenham's algorithm [2] gives us information about which cells the ray might have passed through. As the ray travels from its origin to its end point (where an obstacle may be located), we consider each of the cells in between to be free and we update their log-odds accordingly. Finally, we update the log-odds of the cell in which the obstacle is supposedly located.

Algorithm 2: Occupancy Grid Mapping Algorithm.

Input: Current occupancy grid map `(map)`
Input: LIDAR scan `(scan)`
if *map not initialized* **then**
  previousPose = pose
end
create a moving LIDAR scan
adjscan ← (scan, previouspose, pose, 1)
for *ray in adjscan* **do**
  if *end_is_obstacle = False* **then**
    Use Breshanham's Algorithm to find cells
     along ray and insert them as free
  else
    increase log- odds to insert last point as hit
  end
end
previousPose = pose

E. Resampling the Distribution

The following algorithm was used to resample the posterior distribution

Algorithm 3: Resampling Posterior Distribution

Input: `posterior_`
Input: `sampleWeight`
vector prior_ = posterior_
for *p in prior_* **do**
  p = posterior + rand_normal_dist($\mu = 0, \sigma = 0.04$)
  parent_pose = posterior
  weight = sampleWeight
end
return prior