Bot Lab: Autonomous Ground Vehicle from Low-level Control, SLAM to Planning and Exploration

Zheyuan Zhang, Yu Zhu, Manu Aatitya Raajan Priyadharshini, Thirumalaesh Ashokkumar
{zheyuan, jyuzhu, rpmanu, thiruchl}@umich.edu

***Abstract*—The MBot mobile robotics project aims to develop an autonomous ground vehicle to navigate in the unknown environment. There are four key components of the project. Low-level control executes commands from high-level system to drive the robot based on velocity models and kinematics with a PID controller. Simultaneous Localization and Mapping (SLAM) is at the core of MBot project which allows the robot to use LiDAR to build a map of the environment and localize in that map at the same time. We developed mapping module, particle filter with action model and sensor model. Additionally, we implemented AStar (A*) heuristic search with pruning algorithm for path planning and frontier-guided algorithm for exploration. We present the theory and detailed implementation with experimental results for the project in this report.**

I. Introduction

MBot would be an autonomous ground vehicle, that would have the ability to explore locations in which it is deployed while avoiding obstacles. In this process, it would have capabilities to generate and keep track of maps of the surrounding environment, and also be capable of locating itself in the map. In essence, the bot has the following functionalities:

A. Low-level Control

The low-level control system is based on velocity models and kinematics of the bot running in control loops, that can handle the commands arising from the high level planner. While the robustness of higher level systems plays a significant role in overall accuracy and success, the importance of the low level controllers cannot be understated as the accuracy of the actuators and it's corresponding actuation significantly influence the errors that are accrued by the system, and it's compounding effects over time. Thus, to achieve high accuracy in our low level control, we will implement a feedforward with PID controller as shown in Fig II-B2 . The hardware of our system with respect to actuation and control consists of a differential drive system with two wheels and motors supported by a castor wheel at the rear. We use the Pololu 12V brushed DC gear motors with 20 count/revolution resolution and a 78:1 gear ratio. The motor is driven by a motor driver. We use an MPU9250 9DOF IMU, for our gyro measurements. The control is handled by a Pico board, which acts as the interface for control between a Raspberry Pi(RPi) and actuators. In other words, the communication between the motors and the sensors go through the Pico Board, but the inherent processes are processed on the RPi. Also, the system is powered by a 12V battery pack.

B. SLAM

On top of the control system, we have the 'SLAM' - Simultaneous Localization and Mapping architecture, that can build maps of the environment using data from the LiDAR; concurrently 'localizing' our bot on the generated map. The SLAM layer primarily defines the 'intelligence' of the bot on being able to able to execute a well defined traversal task. For any deployed mobile bot, the ability to 'know' it's environment and to identity it's own location in said environment is a foundational requirement since any conceivable high level task is dependent on this requirement. To achieve this, informative data of the environment is necessary. We use a LiDAR to generate maps of the environment and locate obstacles in the vicinity of the bot. Using this information and the Particle Filter localization algorithm, we implement a solution to the SLAM problem.

C. Path Planning

Once we have low level control and SLAM functioning, we need to provide solutions to how the bot would handle user defined tasks like moving from one location to another while avoiding obstacles. That requires a path planning algorithm like AStar, that would generate efficient path for the bot to follow. This level takes in the map generated as input along with current location and goal as input, to generate the path. These points are taken as inputs by the lower level controller.

D. Exploration

We would also need an algorithm that can iteratively generate the 'goal position' described in the previous section, if the bot in an 'unknown(to the bot and the user) environment', implying no possibility for explicit set of commands from the user. In these scenarios, we will have an exploration algorithm at the highest level, that can aid in understanding an unknown environment by moving through it deterministically, with the goal of reaching all 'corners/regions' of the environment, to generate a complete map.

The above defined features are drawn from the motivations for this work, which is to build the foundations for a robot that have major and vast applications across multiple domains including public/human safety(going into areas with explosion threats, nuclear reactor inspection), space exploration(assessing/exploring environment of any non terrestrial region), autonomous vehicles(a scaled Mbot with seats), service sector(cleaning robots), etc., In general, the any bot with the features of a Mbot can be used in applications where the environment is dangerous or impractical for humans to be in, or to eliminate the requirement of man power and man time for mundane tasks.

II. Methodology

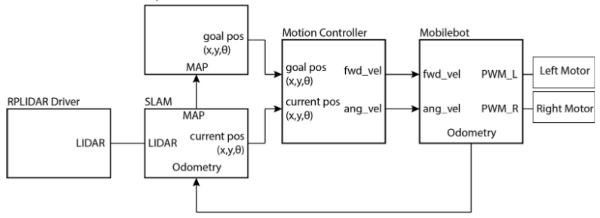
A. Overview of the development

This section describes in detail the development of the MBot. In terms of hardware, we construct and integrate 3 layers.

1) Hardware: At the base layer, we will have the wheels, motors, castor wheel, motor driver and the Pico board mounted onto a chassis. At the second layer, we will have the relevant sensors for perception relevant electronics. Thus, this layer we will include the RPi, a PiCamera mounted on a chassis plate. And at the top, we will mount the LIDAR, specifically the RP-LIDAR. Once the three layers of the hardware are assembled and integrated, we can begin to power up the bot and begin the software development.

2) Software: At the lowest level, commands to related to the mobility of the robot, are routed through the Pico board. The Pico runs the firmware for the control of the actuators. The RPi is the 'computer', that handles SLAM, Path planning operations and processing Camera and LIDAR information. The RPi also interfaces with the Camera and LIDAR. Fig 1, shows the flow of information between the software modules on the MBot. And the flow of information is parsed through the LCM (Lightweight Communications and Marshalling) protocol. Given this structure, we begin by developing low level control following which we discuss SLAM, path planning and exploration.

Fig. 1. Software - Information Flow (Credits: lab manual)



B. Kinematics and Control

1) Kinematics: We update the position estimate using the encoders and the heading sensor. However using them as standalone setup will lead to accumulation of errors and thus, we use this as an initial estimate before it's updated with information from other sensors. We will use the wheel velocity model to update odometry information. For the wheel velocity model, if v_R, v_L represented the velocities of the right and left wheel respectively, then for an angular velocity ω we have,

$$\begin{cases} R &= \frac{b}{2}\left(\frac{v_R+v_L}{v_R-v_L}\right) \\ v_R &= \omega\left(R+\frac{b}{2}\right) \\ v_L &= \omega\left(R-\frac{b}{2}\right) \end{cases} \tag{1}$$

where b is the base length of the robot. The odometry pose is updated by

$$\begin{cases} \Delta\theta_{odo} &= \frac{1}{b}(\Delta s_R - \Delta s_L) \\ \Delta d &= \frac{1}{2}(\Delta s_R + \Delta s_L) \\ \Delta x &= \Delta d\cos(\theta_{t-1} + \Delta\theta_{odo}/2) \\ \Delta y &= \Delta d\sin(\theta_{t-1} + \Delta\theta_{odo}/2) \end{cases} \tag{2}$$

where θ is the previous heading of the robot, Δs_R and Δs_L are the distance traveled by left and right wheels which are obtained by the measurement of encoders in time interval Δt. Note that here we only use the $\Delta\theta_{odo}$ obtained from odometry. This term can be replaced by $\Delta\theta_{gyro}$ obtained by gyroscope, which will be mentioned later. In our case, the value of $b = 0.162$ m, $\Delta t = 0.02$ (50Hz).

We try to validate the odometry model by moving the robot for known distances and turning angles. For instance, when the robot moved along a known square, the orientation of its trajectory was offset by an angle, while a small arc was also observed between each of the four line segments. These are systematic errors whose effects can be reduced. The arc in moving through a straight line could be explained by the differences in wheel diameters (one wheel moves faster). The offset in the square angle could be explained by the uncertainty in the wheel base. The UMBmark procedure [1], which accounts for difference in wheel diameters and uncertainty in wheel base, was performed to correct systematic errors. After trials, the ratio of the wheel base, $E_d = D_R/D_L$ was set to 1.1, while $E_b = b_{actual}/b_{nomial}$ was set to 0.95. Here, D_R, D_L represent the right and left wheel diameters. b_{actual} and b_{nomial} represent the actual and nominal wheel base.

As uncertainty in orientation is a primary contributor to odometry errors, the gyro was used to help find more precise orientation. However, gyro itself drifts and is calibrated every few time steps, $\Delta T = 0.02$. We fuse odometry with a heading estimate from gyro, as is shown in Alg. 1. We ignore the gyro if we aren't moving since gyro's measurement is based on varying speeds across timesteps. At the same time, we rely on the gyro more than odometry if the difference is significant. After multiple experiments and measured the Δ_{GO} values, we found that the error of $\Delta\theta_{gyro}$ was almost always at the order of 10^{-3} rad/sec while $\Delta\theta_{odo}$ might have abnormal values when the robot turns. Since the robot traveling at relatively low speeds, we decided to use the gyro value predominantly for $\Delta\theta$.

Algorithm 1 Gyrodometry

```
Input: $\theta_{t-1}$, $\Delta\theta_{gyro,t}$ and $\Delta\theta_{odo,t}$
Output: $\theta_t$
1: $\Delta_{G0,t} = \Delta\theta_{gyro,t} - \Delta\theta_{odo,t}$
2: if $|\Delta_{GO,t}| > \Delta\theta_{thres}$ then
3:     $\theta_t = \theta_{t-1} + \Delta\theta_{gyro,t}\Delta T$
4: else
5:     $\theta_t = \theta_{t-1} + \Delta\theta_{odo,t}\Delta T$
6: return $\theta_t$
```

2) PID Control: The control system we developed is a feedforward with PID. The feedforward takes the desired motor speed as input and outputs the corresponding PWM duty cycle. For this, we use our motor calibration calibration model, whose results are shown in Fig. 2 and Table I.

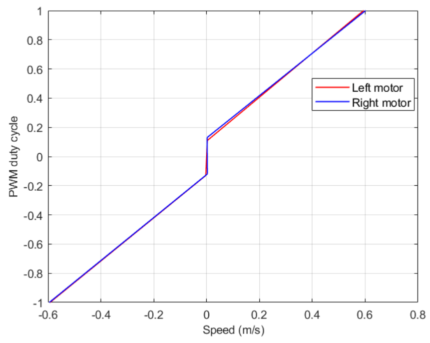


Fig. 2. Fitted motor calibration results

TABLE I
FITTED MOTOR CALIBRATION PARAMETERS

	Left motor	Right motor
Slope (positive)	1.506	1.452
Intercept (positive)	0.106	0.127
Slope (negative)	1.476	1.472
Intercept (negative)	-0.125	-0.124

The feedback component is derived from the encoder readings which is then compared to the reference velocity. The encoder feedback is run through a low pass filter before being used. However, the effects of using a low pass filter was found to be minimal and thus was ignored in later stages. The error between reference and the filtered feedback is the input for the PID. Also, the output of the integrator was also passed through a saturation function, which clamped the output to the range [-1, 1], before it was propagated. To tune the values of k_p, k_i, k_d, we observe the values of the steady state error, rise time and overshoot and visual performance of the robot over a square trajectory. After tuning, the parameter values are shown in Table II.

TABLE II
PARAMETER VALUES FOR PID

Parameters	k_p	k_i	k_d
Values	1	0.15	0.05

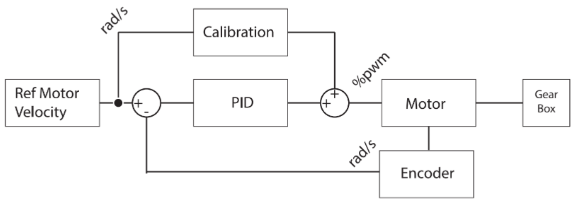


Fig. 3. PID with Feedforward control (Credits: lecture slides)

Also, since we were satisfied with the results of the wheel control PID, we decided not to implement a Frame velocity controller. However, if one were to implement the frame velocity controller, they would use the following equations to implement the PID loop:

$$\begin{aligned} v_R &= v_{fw} + \tfrac{1}{2}\omega b \\ v_L &= v_{fw} - \tfrac{1}{2}\omega b \end{aligned} \tag{3}$$

where v_{fw} is the forward velocity.

3) Motion Controller: Now that we have a functioning control loop, we will implement the higher level motion controller that can take waypoints as input and generate real-time velocities based on the difference between waypoints and current odometry pose (actually the odometry pose will be replaced by SLAM pose after SLAM is implemented). We use the "SmartManeuverController", which decouples rotation and translation movements. The way to generate forward velocity v and angular velocity ω is shown in Eq. (4)

$$\begin{cases} \Delta x &= x_{\text{target}} - x_{\text{pose}} \\ \Delta y &= y_{\text{target}} - y_{\text{pose}} \\ \Delta s &= \sqrt{\Delta x^2 + \Delta y^2} \\ \alpha &= \arctan \frac{\Delta y}{\Delta x} - \theta_{\text{pose}} \\ \beta &= \theta_{\text{target}} - \alpha - \theta_{\text{pose}} \\ v &= k_s \cdot \Delta s \\ \omega &= k_a \cdot \alpha + k_b \cdot \beta \end{cases} \tag{4}$$

where the subscript "target" represents the target waypoint, the subscript "pose" represents the current pose.

k_s, k_a, and k_b are hyperparameters. The values of these parameters in our case are given in Table III. We observed that the performance of the robot was unsatisfactory if we have the term k_b, so we set it to be zero. The forward velocity and angular velocity are also clamped by some preset values. We also clamped the forward acceleration, i.e. the change in forward velocity between two adjacent time steps, to be within [-0.25, 0.25], to prevent the robot from toppling if it moves at high speeds. The threshold for reaching a target pose is set to $|\Delta x| and |\Delta y| < 0.02m$.

TABLE III
PARAMETER VALUES IN MOTION CONTROLLER

Parameters	k_s	k_a	k_b
Values	1	2	0

C. SLAM

SLAM is the integration of a mapping algorithm and a localization algorithm. The algorithm used for localization is the Monte Carlo Localization (MCL) using particle filters. The interaction of SLAM system components is shown in Fig. 4.

1) Localization: MCL algorithm is a non parametric version of the Bayes' filter and requires action model and sensor model to be defined to update particle states and weights. In essence, the MCL can be represented as,

$$P(x_{0:t}|z_{1:t}, u_{1:t}) = \eta P(z_t|x_t)P(x_t|x_{t-1}, u_t) \\ P(x_{0:t-1}|z_{1:t-1}, u_{1:t-1})$$

Here, $P(z_t|x_t)$ is the likelihood obtained from sensor model. $P(x_t|x_{t-1}, u_t)$ is the action model and η is a normalizing factor used since the "evidence" of the Baye's filter is not practically computable. Each of the particles serve as an estimate of the robot pose. In the particle filter, new poses are proposed by the action model. Then sensor model will determine the weight of each particle based on the current map.The weights imply reliability of each particle. Finally, the particles are resampled based on their normalized weights. The slam pose is obtained by the weighted sum of resampled particles.

Action model:

The odometry action model [2] with two uncertainty parameters is used to update the pose of particles. Suppose the current time step is t and the previous step is $t-1$. Denote Δx, Δy, and $\Delta\theta$ as the change in robot pose from the pervious time step to current one. Define $\Delta s = \sqrt{\Delta x^2 + \Delta y^2}$ and $\alpha = \arctan(\Delta y, \Delta x) - \theta_{t-1}$. We assume that the model errors, which include ε_1, ε_2, and ε_3, follow Gaussian distribution: $\varepsilon_1 \sim \mathcal{N}(0, k_1|\alpha|)$, $\varepsilon_2 \sim \mathcal{N}(0, kbaye_2|\Delta s|)$, and $\varepsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|)$, where k_1 and k_2 are hyperparameters. Given the previous

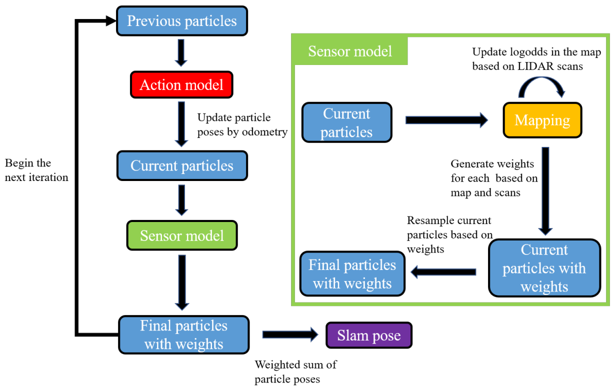

Previous particles
Action model
Update particle poses by odometry
Begin the next iteration
Current particles
Sensor model
Final particles with weights
Slam pose
Weighted sum of particle poses
Sensor model
Update logodds in the map based on LIDAR scans
Current particles
Mapping
Generate weights for each based on map and scans
Resample current particles based on weights
Final particles with weights
Current particles with weights

Fig. 4. Block diagram for the SLAM system

particle pose $[x_{t-1}, y_{t-1}, \theta_{t-1}]$, the new $[x_t, y_t, \theta_t]$ can be obtained by

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

From the equation above, we can conclude that the magnitude of k_1 will affect the distance the particle translates and k_2 is related to the angle the particle rotates. If they are too small, the particles will follow the same relative translation and rotation as the odometry pose. On the contrary, if they are too large, the particles will be extremely sparse and the slam pose (action only) will be a mess. When tuning these two parameters, we first set relatively large values which are around 0.1. We decrease k_1 if the translation part of slam pose differs a lot from odometry pose. Similarly, if the rotation angle of slam pose differs a lot from odometry pose, k_2 will be decreased. After experiments, the final values of uncertainty parameters are shown in Table IV.

TABLE IV
UNCERTAINTY PARAMETERS OF ACTION MODEL

Parameters	k_1	k_2
Values	0.005	0.025

Sensor model:

In this part, the weight of each particle is updated at each time step. The weight of the m-th particle at time t can be expressed as

$$w_t^m = P(z_t|x_t^m) = \prod_{i=1}^{k} P(z_t^i|x_t^m)$$

where z_t^k represents the length of k-th scan at time t. The simplified likelihood field model [2] is used to update the weights, which is shown in Alg. 2 where θ_k is the angle of the k-th scan with respect to the robot frame. We only take LIDAR scans that are blocked by obstacles, which means their logodds are positive. Otherwise, the scan does not hit anything and we do not take them into account for the likelihood. After the weight of all the particles are obtained, these weights are normalized and the method of low variance sampling [2] is used to resample particles based on their weights.

Algorithm 2 Sensor model

```
Input: Particle pose $x_t^m$ and LIDAR scans $z_t$
Output: The weight for the input particle
1: $w_t^m = 0$
2: for all k do
3:     if $z_t^k < z_{\max}$ then
4:         $x_{z_t^k} = x_t + z_t^k \cos(\theta + \theta_k)$
5:         $y_{z_t^k} = y_t + z_t^k \sin(\theta + \theta_k)$
6:         if logodds$(x_{z_t^k}, y_{z_t^k}) > 0$ then
7:             $w_t^m = w_{t-1}^m +$ logodds$(x_{z_t^k}, y_{z_t^k})$
8: return $w_t^m$
```

2) Mapping: An occupancy-grid-based map is used to generate a map of the environment for the MBot. The occupancy grid-based approach abstracts the map as MxN cells, with each cell storing the probability of it being occupied. The map is obtained using Baye's filter update of the cell occupancy using Lidar ray rasterization for the grid cells using Breshenham's algorithm. Log Odds are used instead of probabilities as the log function converts the posterior update to addition and subtraction, making it easier with dealing small numbers. A beam sensor model is used, which updates the posterior belief of the cell being occupied based on whether the ray terminates the cell. The mapping function is done in two parts. First, if the cells are at the ray's endpoints, it implies the possibility of it being an obstacle and thus, the log odds of the cell are increased. Following this, we also update the values of the log odd of cells between the endpoint and the current robot pose by ray rasterization using Breshenham's line algorithm. The hit log odds value is set to +3, and the miss log odds value is set to -2. The reasoning is based on the fact that the lidar has a high accuracy and hence we weight it more and a higher increase in log odds on obstacle detection. But, to eliminate noise, a two-time decrease in the log Odds is required to nullify the effect of a ray falsely identifying an unoccupied cell as an obstacle.

D. Planning

With a controller and SLAM algorithm in place, we can now perform mobility tasks in a given environment. To do so, the bot has to be able to generate a path between any two points. There are many ways of viewing this problem. However, in this context, we will identify algorithms that solves the grid/graph search problem. This is because, it is more convenient to consider the ground of the environment as a grid space, discretized into some n * n space. We can denote any such grid as some a * b * s where 'a' denotes the length of the grid, 'b' denotes the width of the grid, and 's' is the length of each grid.
In order to do path planning, we need to have a working obstacle distance grid. That is, we need a grid where each cell will contain within itself, the distance to it's nearest obstacle. Any cell that contains LogOdds> 0 is considered an obstacle as defined in SLAM. And we only travel towards a particular cell, if the value of the cell in the obstacle distance grid is more than the robot radius. And we use the Brushfire algorithm, that uses a priority queue.

1) A Star: We consider the AStar (A*) search is the algorithm we consider for planning. It is a heuristic search algorithm which can find an optimal path between two locations in a graph. The map can be considered as a graph where the occupancy grid cells represent nodes and connections between grid cells represent edges. The A* algorithm requires a g-cost function and a h-cost function (heuristic). The g-cost function computes the distance from start node to current node. The h-cost function computes the distance from current node to goal node. If the heuristic function is admissible which the actual path cost is always greater than or equal to the heuristic cost at the current node, A* algorithm will always return an optimal path. We use a $cost_g$ of 1 if travelling vertically or horizontally, and use a $cost_g$ of 1.4 if travelling diagonally. For heuristic, we use the Manhattan distance. Finally, the path is returned by using parent node to traverse back from the goal node to the start node and then reverse it.

An A* generated path has consecutive nodes along the path. However, given the discretization, the generated path will have many way points to follow which would significantly lower the speed of robotic motion. Therefore we designed a path pruning algorithm in 3.

E. Exploration:

Exploration is another key algorithm for robot to enable movement in unknown environment, namely, without a map. In real world scenario, there doesn't exist a preset path for mobile robot to follow. Therefore, robot has to explore the unknown environment based on LiDAR information. Inherit LiDAR algorithm generates frontier which is the boundary of explored area and unexplored area. The basic idea for exploration is to follow frontiers until there's no reachable frontier left, which indicates the exploration has already done. However, simply following an arbitrary point on the frontier is not robust for complex environment. Therefore, we designed an algorithm to perform robust exploration in unknown environment which illustrated in 4.

Algorithm 3 Path pruning

```
Input: A* path $nodes$, obstacle distance grid $grid$
Output: Pruned path consisting of robot poses
 1: initialize $path$ to an empty list
 2: $\epsilon \leftarrow 0.005$
 3: for $node$ in $nodes$ do
 4:     $pose \leftarrow$ convert $node$ to global pose
 5:     if $node$ is start node then
 6:         $\theta_{pose} \leftarrow 0$
 7:         push $pose$ to $path$
 8:         $pose^{prev} \leftarrow pose$
 9:     else if $node$ is goal node then
10:         $\theta_{pose} \leftarrow$ atan2$(y_{pose} - y^{prev}_{pose}, x_{pose} - x^{prev}_{pose})$
11:         push $pose$ to $path$
12:     else
13:         $\theta_{pose} \leftarrow$ atan2$(y_{pose} - y^{prev}_{pose}, x_{pose} - x^{prev}_{pose})$
14:         $pose^{prev} \leftarrow pose$
15:         if $|\theta_{pose} - \theta^{prev}_{pose}| > \epsilon$ then
16:             push $pose^{prev}$ to $path$
17: return $path$
```

In the algorithm, we first find a goal position from frontiers. Notice that it will be the mid point of first cells of two frontiers if the total number of frontiers is 2 or 3. Secondly, the diffusion stage iterates from one point gradually expand to further surrounding points to find a point that satisfies the conditions. Thirdly, it generates a "fake path" for robot to follow to avoid termination because we found out that the exploration will fail (terminate) if A* planner cannot find a solution from current robot position to goal position which leads to output a path of length 0. After investigation into this issue, it is very likely that it is caused by system asynchronicity (out-sync). Therefore, by generating a "fake path", it allows the system to synchronize and then move towards the frontier. Additionally, notice that $pose_{robot} + 0.01$ means adding 0.01 to x, y and theta of the robot. This small offset will not change the current robot position too much because in most cases, the robot system will be able to follow a frontier for a generated path with length greater than 0. Therefore, this design makes the robot has an almost 100% success rate on exploring and returning home in small-size to medium-size maze.

III. Results

A. Odometry and PID control

1) Tracking Velocity Setpoint: Figure 5 provides the wheel speeds as a function of time when a given setpoint of 0.5 m/s is provided. The rise time is satisfying, but the oscillations at steady state are quite observable. However, when increasing the Kd gains the system becomes unstable and hence left at the current stage.

Algorithm 4 Exploration

```
Input: $frontiers$, $pose_{robot}$, occupancy grid $map$, obstacle distance grid $grid$, A* planner $planner$
Output: Path consisting of robot poses
 1: initialize $path$ to an empty list
 2: $r_{robot} \leftarrow$ radius of the robot
 3: $\epsilon \leftarrow 0.1$
 4: if number of $frontiers$ == 1 then
 5:     $point^{goal} \leftarrow$ centroid of $frontiers[0]$
 6: else if number of $frontiers$ == 2 or number of $frontiers$ == 3 then
 7:     $point^{goal} \leftarrow$ mid point of $frontiers[0]$.cells[0] and $frontiers[1]$.cells[0]
 8: else
 9:     $point^{goal} \leftarrow$ centroid of $frontiers[1]$
10: $node^{goal} \leftarrow$ convert $point^{goal}$ to grid cell
11: for $i \leftarrow 0$ to 24 step 3 do
12:     $update_x = \{0, 0, i, i, i, -i, -i, -i\}$
13:     $update_y = \{i, -i, 0, i, -i, 0, i, -i\}$
14:     for $j \leftarrow 0$ to 7 do
15:         $node^{goal}_{srnd}.x \leftarrow x^{goal}_{node} + update_x[j]$
16:         $node^{goal}_{srnd}.y \leftarrow y^{goal}_{node} + update_y[j]$
17:         $odds \leftarrow map$.logOdds$(node^{goal}_{srnd})$
18:         $distance \leftarrow grid(node^{goal}_{srnd})$
19:         $\Delta^{robot}_{node} \leftarrow$ Manhattan distance between node and robot
20:         if $odds < 0$ and $distance > r_{robot}$ and $\Delta^{robot}_{node} > \epsilon$ then
21:             $pose_{goal} \leftarrow$ convert $node^{goal}_{srnd}$ to global point

22:             $path \leftarrow planner(pose_{robot}, pose_{goal})$
23:             if length of $path$ == 0 then
24:                 initialize $path$ to an empty list
25:                 push $pose_{robot}$ to $path$
26:                 push $pose_{robot} + 0.01$ to $path$
27: return $path$
```

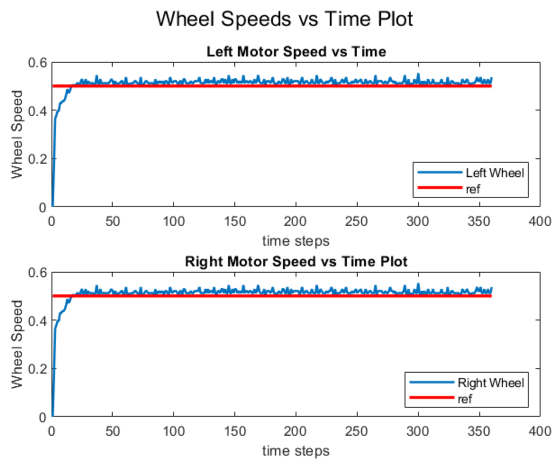


Fig. 5. Wheel Speeds vs Time plot

2) Moving forward and returning to the same location: Figure 6 provides the raw linear and angular velocities on driving forward and returning back to the same location. It can be observed that the velocities have a lot of noise that must be filtered especially in the case of angular velocities. A low pass filter has been used to tackle this problem.

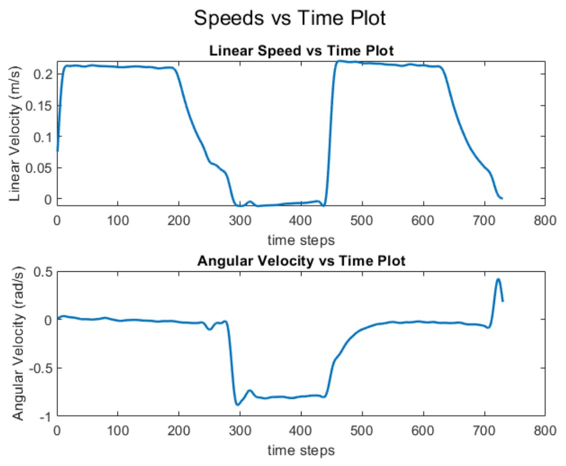


Fig. 6. Move forward and return

3) Looping square path: Figure 7 provides the position and angular velocity plots for driving the bot around a square path 4 times. The odometry measurements drift over time and hence a calibration procedure like UMBMark has been performed to address this issue and the calculated values and procedure have been explained in the Kinematics and Control Section.

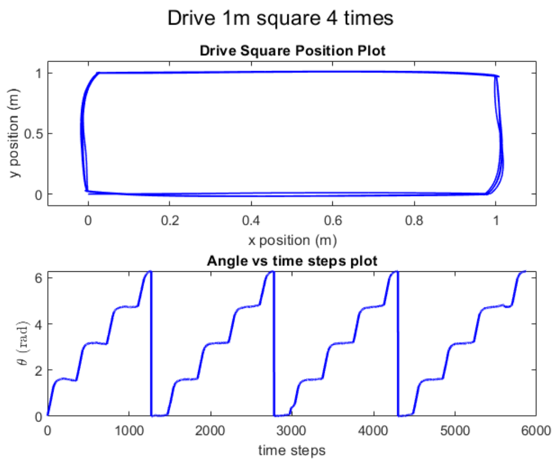


Fig. 7. Drive Square 4 times

4) Position plots for various set velocities: Figure 8 gives the (x,y) position and heading vs time for different set velocities. When $v = 0.25$ m/s and $v = 0.5$ m/s, the time is set to be 2s. For $v = 1$ m/s, the time is set to be 1s. From the plots, it can be observed that the noise in the heading increases as the linear velocity of the bot increases. The measured path deviates to either left or right side of the desired path. The error may be caused by wheel slipping under high acceleration and limited resolution of wheel sensors. Finally, we can also find that the robot only moves about 0.65 m when $v = 1$ m/s, much smaller than the theoretical value 1m. This can be explained by the fastest speed of left and right wheels shown in Table V.

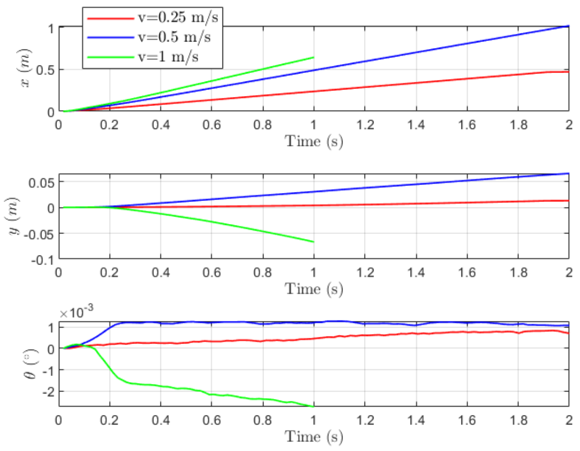


Fig. 8. Pose plots for different velocities

TABLE V
SLOWEST AND FASTEST SPEED

	Angular speed (rad/s)	Translation speed (m/s)
Min	0	0
Max	8.591	0.661

B. Mapping:

Figure 9 represents the map obtained using SLAM when simulated using the obstacle slam log file. The map is of good quality, as seen in Figure 9.

C. SLAM

1) Particle Filters: The following table gives the median time taken to execute particle filter computations on running through the "obstacle grid maze" log file 5 times. The time taken increases almost linearly.

From Table VI, it can be inferred that the maximum number of particles that can be used to run on a Pi running at 10 Hz (at 0.1 s for each update) is between 15500 - 16000 particles.

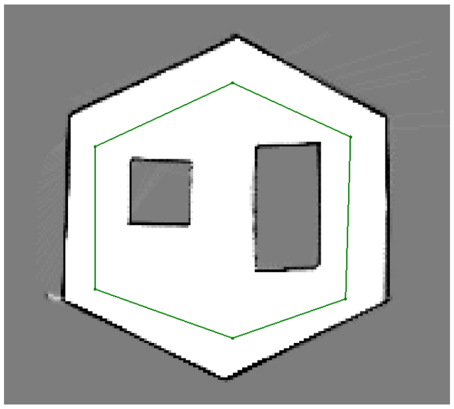


Fig. 9. Obstacle SLAM 10mx10mx5cm Map

TABLE VI
PARTICLE FILTERS TIME

Number of Particle	Median Time taken (ms)
100	6.735
200	13.500
300	20.582
500	33.776
1000	63.100
10000	658.135
15000	957.860
16000	1010.580

2) Drive Square - SLAM: Figure 10 below gives the spread of particles at the specified locations when driving around a square using the drive square log file.

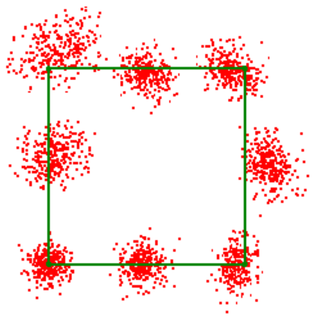


Fig. 10. Particle Spread driving square with 300 particles

Figure 11 gives the slam vs the odometry pose when driving along a square using the drive square log file. The error statistics from the poses are as follows, RMS Error in x = 3.7 cm, RMS Error in y is 2.7 cm and RMS Error in θ is 1.3 deg. Generally, we can conclude that the slam pose is quite close to the odometry pose with some small errors. Since the errors oscillate around 0, they may be caused by randomness of updating the particle poses in SLAM. The accuracy could be improved by properly increasing the number of particles or reducing the standard deviation in action model.

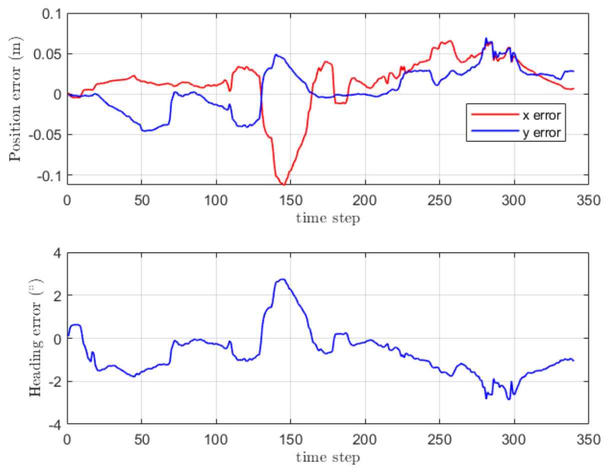


Fig. 11. Slam Pose vs Odometry Pose when driving around square

3) Convex Path - SLAM: Figure 12 gives the SLAM vs the odometry pose when driving along a convex maze using the obstacle grid log file. We can conclude that the slam pose we obtained is better than odometry pose. In addition, table VII indicates that the overall RMS is acceptable. But the slam pose still deviates a bit from the true pose. The error mainly occurs during the turning process. Further tuning k_1 value in the action model may help cope with this problem.

4) Checkpoint 2 - SLAM: The MBot follows the path published by the Python script in checkpoint 1. In Appendix A, Figure 13 shows the maze we used for demonstrating SLAM and Figure 14 shows the SLAM mapping result with SLAM path and odometry path.

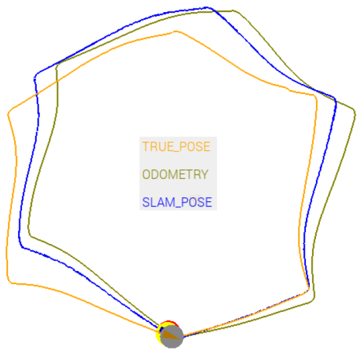


Fig. 12. Comparison between different poses: driving through convex world

TABLE VII
SLAM Performance Statistics for a Convex Maze

RMS Error x	7 cm
RMS Error y	4 cm
RMS Error θ	2.31 deg

D. Planning

1) A search:* The A* test statistics are as follows in Table VIII

TABLE VIII
Timing Information for successful Planning Attempts (us)

Parameters	Value
Min	353
Mean	4843.25
Max	10301
Median	2969
Std dev	3689.77

2) Exploration: For competition task 2, we are aimed to let the MBot automatically explore the unknown environment (maze) without giving a designated path or odometry command. The robot will keep exploring until the environment has been fully explored. After that, the robot will return to the home position (starting position) automatically to finish the task. In Appendix B, Figure 15 shows the maze we used for demonstrating competition task 2 (exploration and returning home) and Figure 16 shows the SLAM mapping result with SLAM path and odometry path.

IV. Discussion and Conclusion

In this project, we developed an autonomous ground vehicle which capable of automatically exploring and navigating in the unknown environment. However, there are still some improvements we can make to let the robot generally perform better as an unmanned vehicle. From the results presented above, we can see that, after bug fixes and modification to the algorithms, the robot has successfully accomplished competition task 2 (exploration and returning home) which we failed to accomplish in the competition day because of bad allocation error. After investigation, we found out that it is caused by using a potentially infinite while loop that costs too much memory but other parts of the system create new objects in the meantime which eventually lead to bad allocation. The reason of being potentially infinite loop is because we used an algorithm to randomly sample points within a L1 square range of the goal point computed from the frontiers information. Thus, it is very likely to run into the case where we have a bad luck of failure to randomly sampled a point which satisfy the while loop exit conditions. Due to this issue, we change the algorithm to a for loop which guarantees to stop after all the iterations. This eliminates the bad allocation error. We also modified some parts of the exploration algorithm to make it more robust.

Additionally, the SLAM mapping result for competition task 2 is not very clear which caused by overlapping deviations of the map. This is because our localization algorithm is not very accurate, i.e. the robot on the same pose (x, y, theta) on the physical map for different time step are actually different for the SLAM pose. Then the new laser scan updates on a deviated map which overlaps upon the original map. Therefore, we may need to tune hyper-parameters involved in the particle filter including action model and sensor model. Moreover, new algorithms may need to be implemented to address this issue.

References

[1] L. F. J. Borenstein, "Umbmark — a method for measuring, comparing, and correcting dead-reckoning errors in mobile robots." [Online]. Available: http://www-personal.umich.edu/~johannb/Papers/umbmark.pdf

[2] S. Thrun, "Probabilistic robotics," *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

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

[4] H. Choset, "A-star lecture notes." [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf

[5] D. L. Poole and A. K. Mackworth, "Artificial intelligence: Foundations of computational agents, poole & mackworth - multiple path pruning." [Online]. Available: https://artint.info/2e/html/ArtInt2e.Ch3.S7.SS2.html

[6] A. Patel, "Introduction to astar." [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html

[7] S2T5, "task2 - competition." [Online]. Available: https://drive.google.com/file/d/1J_zVYvnbyBrE59yy_oLomuH5xp3FldVF/view?usp=sharing

V. Appendix

A. Checkpoint 2 maze - SLAM

Results from checkpoint 2:

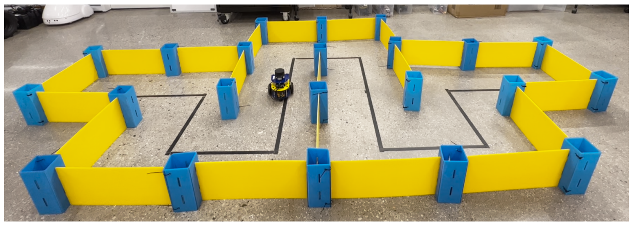


Fig. 13. MBot following the path in real environment - checkpoint 2 maze

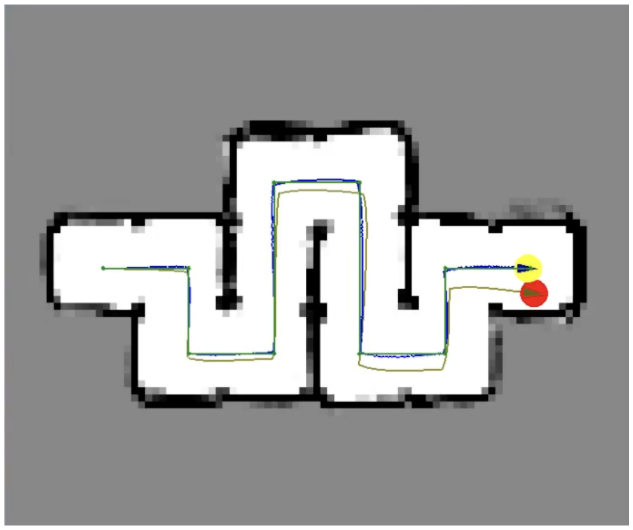


Fig. 14. SLAM mapping result with SLAM path and odometry path

B. Competition task 2 maze - SLAM

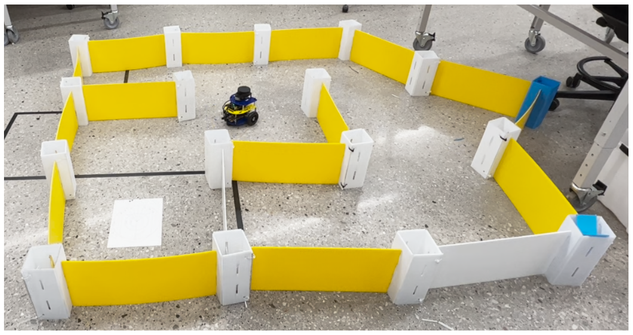


Fig. 15. Maze of competition task 2 - explore and return home

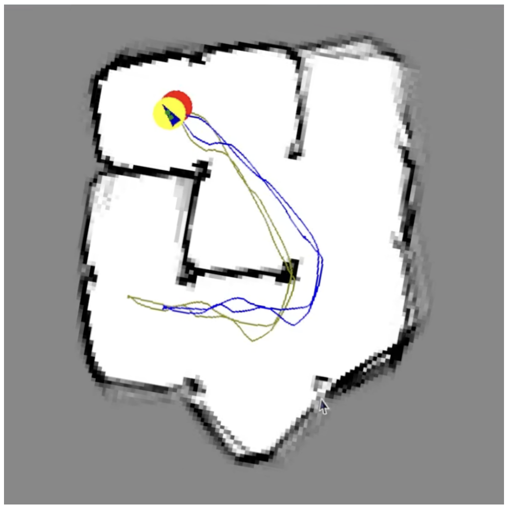


Fig. 16. SLAM mapping result of competition task 2 - exploration and returning home with SLAM path and odometry path