Team S2T6 ROB 550 BotLab Report

Jonathan Heidegger, Ke Liu, Sibo Wang

***Abstract*—In the field of robotics, there is a growing need for advanced technologies that enable robots to navigate and interact with their environment in a more intelligent and autonomous manner. This lab reports focus on the development of algorithms and techniques for motion and odometry, simultaneous localization and mapping (SLAM), and path planning. These technologies are essential for enabling robots to perform tasks such as localization, navigation, and obstacle avoidance, which are critical for a wide range of applications, including autonomous vehicles, manufacturing and logistics, and search and rescue operations.**

I. Introduction

BOTLAB discusses various applications of robotics technology, including motion and odometry, simultaneous localization and mapping (SLAM), and path planning.

The platform of the MBot 1 is an open hardware platform for robotics education and research developed at the University of Michigan. It is a skid steer robot using a Raspberry Pico co-processor and Raspberry Pi as primary processor. This platform allows for small mobile robotics education and research among multiple groups for an affordable price. This paper uses the Mbot platform.

Motion and odometry involve the measurement and analysis of the movement of a robot, which is important for tasks such as localization and navigation. SLAM refers to the process of creating a map of an unknown environment while simultaneously determining the location of the robot within that environment. Path planning involves the development of algorithms and strategies for determining the best path for a robot to follow in order to reach a specific goal.

These applications of robotics technology are important for a wide range of applications, including autonomous vehicles, manufacturing and logistics, and search and rescue operations. This report explores the various techniques and technologies used to support these applications, as well as the challenges and opportunities presented by this rapidly evolving field. We will first discuss the methodology we used for motion and odometry, simultaneous localization and mapping (SLAM), and path planning, then we will evaluate and analyse our results, and finally we will provide our reflections as well as suggestions for further improvement.

Fig. 1: The Mbot Platform

A. Motion and Odometry

1) Wheel Speed Controllers: The foundation of the Mbot control is in the wheel speed controllers. A PF loop is implemented to govern the motor controllers PWM duty command for each wheel independently. The wheel position is measured using magnetic quadrature encoders and velocity measured as the number of encoder tics over a set time period. Finally encoder measurements are converted to meters and meters/second respective using the gear ratio (78.0), wheel radius (.042m), and encoder resolution (20 tics/rev). The feed forward term K_F is generated experimentally with the robot driving on the ground at various PWM duties with values being recorded once steady state velocity had been achieved. A linear regression is fit to the gathered data points giving the feed forward K_F constant as well as the intercept K_{F0} representing the PWM duty needed to overcome the static friction of the system. A Proportional loop is then used to account for any other system perturbations or errors.

2) Robot Base Velocity: Control for the robot often desired to done in the base frame not at a wheel level. The conversion is done simply by

$$V_{left} = V_{linear} - \omega * wheel\_base/2.0$$
$$V_{right} = V_{linear} + \omega * wheel\_base/2.0$$

Thus a linear and angular velocity of the robot base frame is converted to wheel speeds. It was determined that an additional controller for the vehicle frame was not needed because of the accuracy of the wheel controllers. Instead further development was invested in the high level motion controller.

3) Odometry: The pose of the robot is tracked as a process of odometry. Odometry combines sensor data from the encoders and IMU to synthesis the robot pose relative to an initial pose. The encoder data from the wheels gives a distance delta for each wheel. This can be converted into a pose delta according to the following

$$\delta_d = (\delta_l + \delta_r)/2$$
$$\delta_\theta = (\delta_l - \delta_r)/wheel\_base$$
$$\delta_x = \delta_d * cos(\theta + \delta_\theta/2)$$
$$\delta_y = \delta_d * sin(\theta + \delta_\theta/2)$$

This gives a delta pose that is accumulated resulting in odometry. To improve the accuracy of the heading measurement the IMU is used. IMUs offer a higher accuracy for heading with the drawback of zero rate drift. "gyrodometry" solves this drift problem by trusting the IMU for an accurate theta any time δ_d is larger than a threshold indicating the robot is moving. When δ_d is less than this threshold an accumulator captures the drift of the IMU heading measurement that is applied for every subsequent measurement. Thus the robot is able to benefit from an increased accuracy to heading and mitigate the zero rate drift when not moving.

4) Motion Controller: The Motion Controller takes in a series of points and uses an RTR (Rotate Translate Rotate) controller to interpolate between these points. First, the initial turn is accomplished by a proportional controller in place.

The translation section of the path is implemented with a trapezoid profile controller. A linear 1 dimensional trapezoid motion profile is computed that takes the robot from the current position to the next target point bounded by constants `max_accel` and `max_velocity`. The feed forward position along this trajectory is calculated by projected the 1 dimensional trapezoid profile back into Cartesian space with start and end points of current pose and goal pose. For each time step along the trajectory there is a feed forward velocity and expected pose. The translation error is computed from the current pose and the expected pose at the time step. This error is divided into components of those parallel to the linear trajectory perpendicular to the trajectory. The parallel component is used in a proportional controller for the translation velocity of the robot. A δ_θ is calculated as the difference of the current theta and the line of sight angle to the goal target. The controller is finished when the translation error is less than $.02m$.

$$V = V_{ff} + K_p * \delta_{parallel}$$
$$\omega = K_p * \delta_\theta$$

A final turn controller is executed to bring the heading of the robot to the goal heading once it has reached the goal point from the translation controller. This again is a proportional controller with a high gain and a lower error tolerance for completion.

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: Firstly, the map is discretized into grid cells by the provided `OccupancyGrid` class, where each cell has an assigned log odds. The log odds of a cell, $O(x, y)$, is defined as

$$O = \log\Big(P(A)/\neg P(A)\Big)$$

where $P(A)$ is the probability that a cell is occupied. $O = 0$ when $P(A) = 0.5$. O then increases as $P(A)$ increases and vise versa. In this project, for computation efficiency, O is truncated as a integer $\in [-128, 128]$.

Then, the mapping is updated based on the LIDAR laser scan. For each ray in a scan, the endpoint (x_e, y_e), $O(x_e, y_e)$ is incremented by the default value 3, increasing the belief that the cell is an obstacle.

Meanwhile, for all cells $(x^{(i)}, y^{(i)})$ in between current pose and endpoint, $O(x^{(i)}, y^{(i)})$ is decremented by the default value 2, increasing the belief that the cells are free.

The mapping is updated whenever the robot is determined to be in motion, which is checked using the default conditions,

$$d\theta > 0.002 \cup dS > 0.002$$

where $d\theta$ is the change in heading and dS is the distance traveled between poses.

2) Monte Carlo Localization: The Monte Carlo localization estimates the position and orientation, which are together referred to as the pose, of the robot as it moves and senses the environment [1]. Specifically, a particle filter is used to represent an empirical distribution of likely poses, with each particle representing a hypothetical belief of the robot pose [2]. The details of how the particle filter fuses odometry and laser data to output localized pose will be discussed next.

a) Measurement Model: Odometry-based measurement model is adopted as odometry data is readily available. For $t = k$, the model firstly dissembles the update in odometry in the last time step into a RTR process [2],

$$\delta_{rot1} = \arctan(y_k - y_{k-1}, x_k - x_{k-1}) - \theta_{k-1}$$
$$\delta_{trans} = \sqrt{(x_k - x_{k-1})^2 + (y_k - y_{k-1})^2}$$
$$\delta_{rot2} = \theta_k - \theta_{k-1} - \delta_{rot1}$$

where only when any of $\{\delta_{rot1}, \delta_{tran}, \delta_{rot2}\}$ are greater than given thresholds would the robot be considered to have moved. If moved, all the particle poses would be propagated by the odometry with tunable Gaussian noise.

Specifically, let $\{x^{(i)}, y^{(i)}, \theta^{(i)}\}$ be the pose of the i-th particle,

$$x_k^{(i)} = x_{k-1}^{(i)} + \hat{\delta}_{trans} \cos(\theta_{k-1}^{(i)} + \hat{\delta}_{rot1})$$
$$y_k^{(i)} = y_{k-1}^{(i)} + \hat{\delta}_{trans} \sin(\theta_{k-1}^{(i)} + \hat{\delta}_{rot1})$$
$$\theta_k^{(i)} = \theta_{k-1}^{(i)} + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$$

where

$$\hat{\delta}_{rot1} = \mathcal{N}(\delta_{rot1}, \alpha_1 \delta_{rot1}^2 + \alpha_2 \delta_{trans}^2)$$
$$\hat{\delta}_{trans} = \mathcal{N}(\delta_{trans}, \alpha_3 \delta_{trans}^2 + \alpha_4 \delta_{rot1}^2 + \alpha_4 \delta_{rot2}^2)$$
$$\hat{\delta}_{rot2} = \mathcal{N}(\delta_{rot2}, \alpha_1 \delta_{rot2}^2 + \alpha_2 \delta_{trans}^2)$$

where $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ are tunable parameters respectively representing the noise of rotation on rotation, translation on rotation, translation on translation and rotation on translation. The variances of the Gaussian noises should be inversely proportional to the confidence in the accuracy of odometry data. More details for the tuning of parameters will be discussed in Results.

b) Sensor Model: The sensor model used is based on the simplified likelihood model introduced in ROB550 lecture slides. The model superposes the current LIDAR scan onto the pose of each particle and assigns a weight to the particle based on how likely is the LIDAR scan given the particular pose. The likelihood is calculated based on if the end of each ray is near a hit. Algorithm 1 outlines the steps, where O_T, a_1, a_2 are tunable parameters. Modifications are made compared to the original model introduced in class. Firstly, stricter occupied condition is imposed with $O_T = 110$, which has experimentally proved to have better performance, following more closely to the actual pose. The consideration of diagonal neighbors has also improved performance as they follow the ray's path more accurately. The coefficients are tuned to $a_1 = 0.6, a_2 = 0.4$.

Algorithm 1 Simplified likelihood sensor model

Input: particle pose $p^{(i)}$, lidar scan, map
Because robot is moving while taking scan, interpolate rays between $p^{(i)}$ and its parent
for all $ray \in movingScan$ **do**
  Find ray's end cell (x, y)
  if $O(x, y) > O_T$ **then**
    $L(p^{(i)}) += O(x, y)$
  else
    if ray $\in \pm$10deg of X/Y axis **then**
      take horizontal/vertical cells as neighbors
    else
      take diagonal cells as neighbors
    if O(after)$> O_T$ **then**
      $L(p^{(i)}) += a_1 O$(after)
    else if O(before)$> O_T$ **then**
      $L(p^{(i)}) += a_2 O$(before)
Return $L(p^{(i)})$

c) Particle Filter: The particle filter maintains an empirical distribution of N weighted particles, each representing a belief of the pose of the robot. The filter fuses the probabilistic action and sensor models and recursively localizes the robot.

The filter initializes with particles $p_0^{(i)} = (x_0^{(i)}, y_0^{(i)})$ with $w_0^{(i)}$ where $i = 1, 2, \ldots, N$. If the initial position of the robot is known, then $p_0^{(i)}$ are initialized by sampling an Gaussian distribution centered at the known position with $\sigma = 5$mm, accounting alignment error when placing the robot. Meanwhile, if the initial position is unknown (kidnapped), then $p_0^{(i)}$ are sampled uniformly across the map. For both cases, the initial weights $w_0^{(i)}$ are all set to be equal.

Then, for $t = k$, the filter recursively updates the poses $p_k^{(i)}$ and weights $w_k^{(i)}$ based on Algorithm 2. where the re-sampling utilizes low-variance sampling, with its details found in [2]. Meanwhile, after trials and errors, it was determined that keeping N to be relatively low and using all particles for weighted average is computationally more efficient and stable than using large N and only using sorted upper percentiles for weighted average. Logically, the former encompasses a more holistic representation of a possibly narrower current belief, which has demonstrated better performance. $N = 200$ was chosen for the final implementation.

Algorithm 2 Particle filter for localization

Input: odometry pose (x, y, θ), laser scan, map
Determine if robot has moved
if has moved **then**
  Re-sample posterior distribution
  Propagate $p_k^{(i)}$ from $p_{k-1}^{(i)}$ using action model
  Update $w_k^{(i)} = L(p^{(i)})$ using sensor model
  Normalize $w_k^{(i)}$ such that $\sum w_k^{(i)} = 1$
Return weighted average of $p_k^{(i)}$ as the localized pose

3) Combined Implementation: The combined implementation involves simultaneously localizing the robot while updating the map. With the above features implemented, the combined implementation is simply to initialize grid map and particle filter first, then running the particle filter and mapping algorithm sequentially in a loop, which is shown in Figure 2.

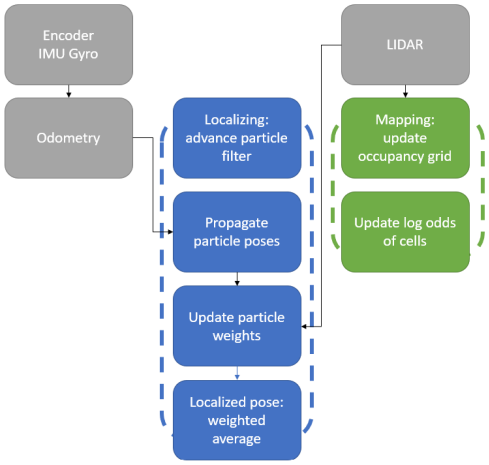

Fig. 2: Block diagram of SLAM system

C. Planning and Exploration

After implementing SLAM algorithms as above, a map of an environment can be constructed using the MBot. The Path Planning and Map Exploration will enable the MBot to autonomous recognize the surrounding environment in an efficient way.

Algorithm 3 A* Algorithm

Input: $start$, $goal(n)$, $h(n)$, $expand(n)$
Output: path
if $goal(start) = true$ **then**
  Return $makePath(start)$
$open \leftarrow start.$
$closed \leftarrow \varnothing.$
while $open \neq \varnothing$ **do**
  $sort(open).$
  $n \leftarrow open.pop().$
  $kids \leftarrow expand(n).$
  for all $kid \in kids$ **do**
    $kid.f \leftarrow (n.g + 1) + h(kid).$
    if $goal(kid) = true$ **then**
      Return $makePath(kid)$
    if $kid \cap closed = \varnothing$ **then**
      $open \leftarrow kid$
    $closed \leftarrow n$
Return $\varnothing$

1) Path Planning: Path Planning uses A* Algorithm. In A* Algorithm [3], the open list is implemented as a priority queue which stores all accessible neighborhood cell using breadth-first search. the closed list is also implemented using a priority queue to store all the cell being passed through.

Firstly, the open list will contain the cell in the start position (it could be interpreted as the start node, while each node represents a cell in the grid map), and the algorithm will start searching map from this position, and stop until the open list is empty.

While the open list is not empty, the open list will pop out the node with highest g cost (cost of movement in grid) plus h cost(heuristic cost to move from current position to goal position), and expand to other surrounding cells that available for MBot to reach. For all current node, the maximum quantity of kid nodes are 8, as there are at most 8 cells in its neighborhood. To assess whether the kid node is available, the kid cell's distance towards obstacles must be larger than 0.2m (larger than the radius of MBot), and the cell must be in the current map. If both requirements are satisfied, the kid node is available. Those available nodes' parent is the current node, and they will be push back to the open list if they are not in the closed list. Finally, the current node which being pop out will be added to the closed list.

The g cost is calculated by adding up the parent cell's g cost and the Diagonal Distance from parent cell to current cell. H cost is also calculating the `Diagonal Distance` from current cell to goal cell [4].

When processed in each while loop, each kid node's cell position is compared with the goal position. If they are the same, A* will stop searching. Tracing back from each node's parent node, a linked node path from start to goal is generated. maximum iteration time of the for loop to 10000 times is set in the case that A* cannot find the map such as when goal position is not reachable by obstacles. Thus, when it reaches the loop limit while not reaching the goal, the algorithm will stop and return a prompt showing search failure.

The path, if generated, is pruned such that only changes in heading are sent to motion controller. This pruned path then can be more efficiently followed by the motion controller such that it does not attempt to reach each point of the un-pruned path.

2) Exploration: The Map Exploration consists of several states, including `Initializing`, `ExploringMap`, `ReturningHome`, `Completed`, `Failed`. First it will initialize its information, including current pose (also being copied as home pose), goal pose(preset to current pose), and start publish the status message to lcm channel. After initialization, it will automatically switch to `ExploringMap` stage. It will find all exists frontiers and place them in a vector. Then web will process throughout all frontiers to select one that has the nearest midpoint to current position. Then, it will use the above Partial A* Algorithm to plan a path to approach this frontier. While the `ExploringMap` states is still in progress, the algorithm updates the path after 10 iteration with the `ExploringMap` state is called. So it will update the path to the new nearest frontiers as its exploring the map.

Simultaneously, the number of unreachable fron-

Algorithm 4 Partial A* Algorithm

Input: $start$, $goal(n)$, $h(n)$, $expand(n)$, $isclose$
Output: path
if $goal(start) = true$ **then**
  Return $makePath(start)$
$open \leftarrow start$
$closed \leftarrow \varnothing$
$close\_node \leftarrow start$
while $open \neq \varnothing$ **do**
  $sort(open)$.
  $n \leftarrow open.pop()$.
  $kids \leftarrow expand(n)$.
  for all $kid \in kids$ **do**
    $kid.f \leftarrow (n.g + 1) + h(kid)$.
    if $h(close\_node) > h(kid)$ **then**
      $close\_node \leftarrow kid$
    if $goal(kid) = true$ **then**
      Return $makePath(kid)$
    if $kid \cap closed = \varnothing$ **then**
      $open \leftarrow kid$
    $closed \leftarrow n$
if $isclose$ **then**
  Return $makePath(close\_node)$
Return $\varnothing$

tiers is recorded. Thus, the state will switch to `ReturningHome` if frontiers number is equal to the number of unreachable frontiers, and changes to `Completed` state when it successfully return to the home position. If all frontiers' quantity is higher than unreachable frontiers', while Partial A* Algorithm cannot find any available path, the state will switch to `Failed` and the program will be ended.

The vehicles automation is achieved through map exploration using a strategy similar to A* algorithm, that allows MBot to explore the map itself and return home when all the frontiers has been explored. A frontier is detected and defined as unexplored cells bordering an open cell in the occupancy grid.

Once MBot starts exploration, it will seek all frontiers in the map, and try to calculate a path from start position to the frontier whose midpoint is nearest.

Most of the case, the midpoint of the frontier is not within the map, so setting the midpoint directly as the goal position will failed the original A* search. Two options to mitigate this are breadth-first search from the midpoint and find an available cell to be the goal position, or create another unique strategy to reach some available cells closer to the midpoint.

A unique strategy was chosen that relies on adjusting A* search. It was named the Partial A* Algorithm. This algorithm will continue after the open list when empty

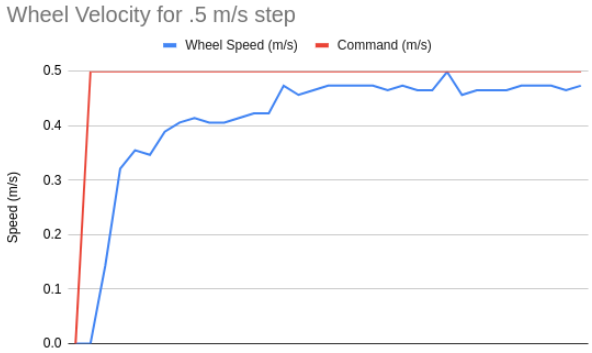

Fig. 3: Step response of wheel PK controller for .5 m/s step

or the loop reaches the maximum loop times. The node with the lowest h cost is recorded in $close\_node$. Then by tracing the parent node from the $close\_node$, it will return a path from start node to $close\_node$ as next exploration path (Detailed in Algorithm 3). This satisfies that all points are within the configuration space of the robot and the goal pose is the closest to the goal pose that sits outside the configuration space.

SLAM is used to localize the MBot while automation steps are running. The slam pose will be updated as the odometry information is continuously changing. Therefore, the Mbot can tell its location in the grid map.

II. Results

A. Motion and Odometry

1) Wheel Speed Controllers: The results of the wheel speed controller step function response was 3. Accepted levels of accuracy were achieved using the same loop constants for both the left and right wheels which was determined experimentally with the robots ability to drive in a straight line for 1 meter deviating less than 5 cm from the center line. These values were I. The wheel controllers can accept values $\pm 0.8 m/s$ as the maximum with best results achieved when commanding between $\pm 0.1 - 0.2 m/s$. This range was prioritized in the feed forward and tuning of the wheel speed controller. The wheel speed data was combined to create the velocity and angular speed of the robot driving an example 1 meter out and back path. 4

K_P	2.0
K_F	1.176
K_{F0}	0.0737

TABLE I: Wheel Controller Values

2) Odometry: The odometry accuracy was measured using a test pattern of a 1m square with goal points at each corner. Multiple iterations of the path then can show

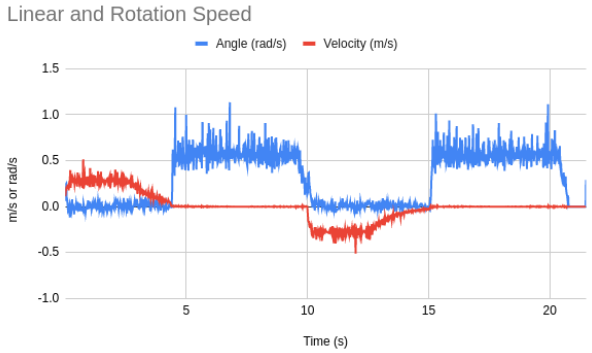

Fig. 4: Linear and Angular speeds on an example path from Wheel Speed

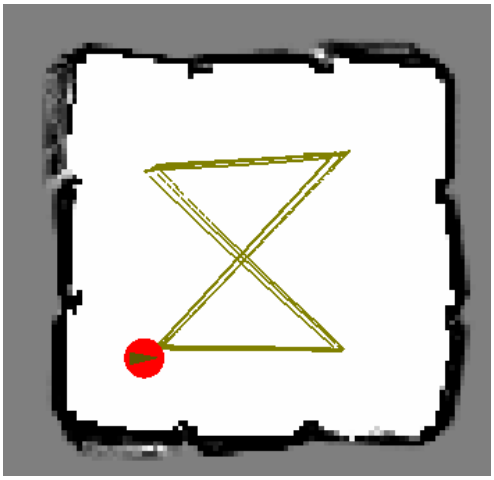

Fig. 5: Drift of odometry over 1m square test. 4 repetitions

the drift of the odometry over time 5. The odometry drifted less than 5cm over the 4 repetitions from the actual position of the robot.

3) Motion Controller: The Motion Controller led to a high level of accuracy in combination with the optometry. This is primarily because the controller took into account the inability of the wheels to instantly accelerate and thus the errors from wheel velocity where much smaller than seen in the step response 3. This results in a smooth achievable pose as can be seen in the overlap of the feed forward pose with the measured pose in 15. These low errors combined for an accurate path following such as for an example maze run at .2 m/s 14.

B. Simultaneous Localization and Mapping (SLAM)

1) Mapping: Figure 6 demonstrates the performance of the mapping algorithm.

2) Monte Carlo Localization:

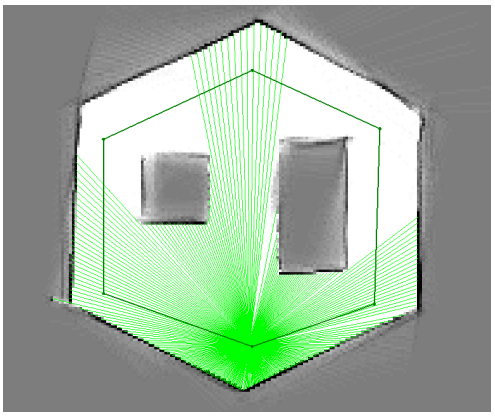

Fig. 6: Mapping-only on `obstacle_slam`

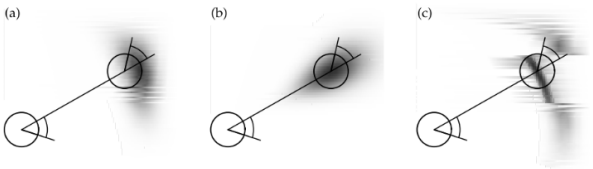

Fig. 7: The odometry action model for different noise parameter setting [2]. (a) similar confidence in translation and rotation; (b, c) less confidence in translation / rotation respectively

a) Measurement Model: Ideally, when localizing using only the action model, the particles should disperse steadily due to the accumulated noises. Specifically, sudden increase in the particle spread for edge cases are eliminated, such as when turning-in-place with large magnitudes. After tuning, the parameters used are listed in Table II.

Parameter	Value
α_1	0.015
α_2	0.025
α_3	0.04
α_4	0.0001

TABLE II: Tuned uncertainty parameters for the action model

where the main sources of uncertainty are from translation on translation and rotation on rotation. α_4 is extremely small as it also inherently incorporates the unit conversion from rad/s to m/s. Since the odometry was well constructed for both translation and heading direction, the parameters were tuned to established (a) in the patterns shown in Figure 7.

b) Particle Filter: Table V shows how the run time linearly increases as the number of particles increases. Based on the linear relationship, the estimated run time for $10Hz = 0.1s$ is $N = 2500$ particles.

	test_convex_grid	test_empty_grid	test_maze_grid	test_narrow_constriction_grid	test_wide_constriction_grid
Min	137	4267	2600	3006	3297
Mean	182.5	6105.33	15322.2	4301.5	4831
Max	228	9243	31538	5597	6090
Median	0	9243	8512	0	6090
Std dev	45.5	2229.55	10979.4	1295.5	1156.7

TABLE III: The timing information for successful planning attempts

	test_convex_grid	test_empty_grid	test_filled_grid	test_narrow_constriction_grid	test_wide_constriction_grid
Min	48	50	28	45	50
Mean	143	92.5	38.4	1.75911e+06	50
Max	238	135	49	5.27715e+06	50
Median	0	0	43	127	0
Std dev	95	42.5	8.73155	2.48763e+06	0

TABLE IV: The timing information for failed planning attempts

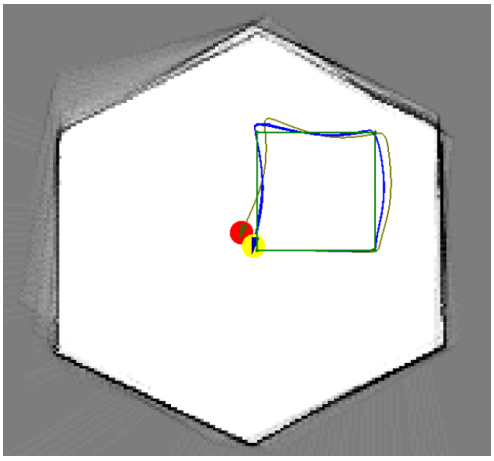

Fig. 8: Error difference between SLAM and odometry for `drive_square`

Particles (N)	Time (s)
100	0.0045
200	0.0081
300	0.0118
500	0.0210
1000	0.0399

TABLE V: Time taken to update particle filter with different number of particles N

Please refer to the appendix for a series of plots for the performance of 300 particles on `drive_square_10mx10m_5cm.log`. Figure 8 shows the pose difference between SLAM and odometry poses overtime.

3) Combined Implementation: Figure 9 shows the comparisons of SLAM with true poses. As seen, the SLAM pose has made corrections to the odometry pose towards the true pose by propagating noise to odometry and fusing sensor model data.

Figure 24 in Appendix further analyzes the error in the above plot numerically. With these numerical errors, the RMS is calculated to be 0.467.

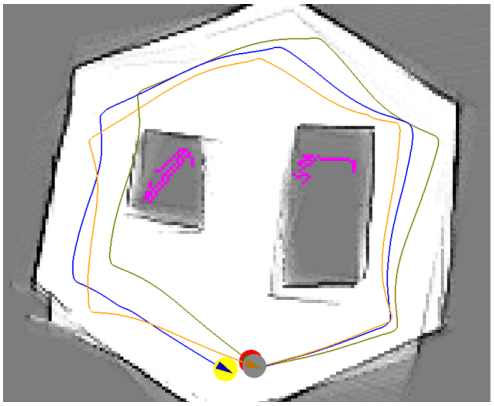

Fig. 9: Error difference between SLAM and true pose for `obstacle_slam`

C. Planning and Exploration

1) Path Planning: One of the requirement is to show the planned path in the environment mapped and the actual path driven by robots. The figure 10 is the robot executing the map exploration in a maze, and most of the time the odometry (yellow line) and slam pose (blue line) is aligned with the planned path (green line).

To test the A* algorithm, the `astar_test.cpp` was run to see the statics output. The statistics on Table III and Table IV are the successful attempt and failure attempt on path planning execution for each example problems provided in `data/astar/` directory. The results shows passing 5/6 of the A* tests, the test for the `test_empty_grid`, `test_filled_grid`, `test_narrow_constriction_grid`, `test_wide_constriction_grid`, `test_maze_grid` is passed, while the test for `test_convex_grid` is failed. The timing information is concluded in Table III and IV

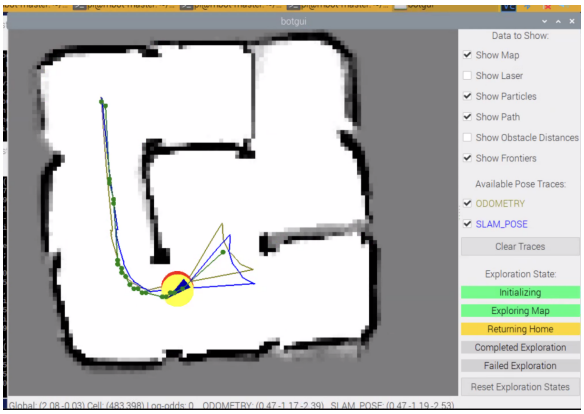

Fig. 10: Figure of planned path and actual path driven by robots in an maze

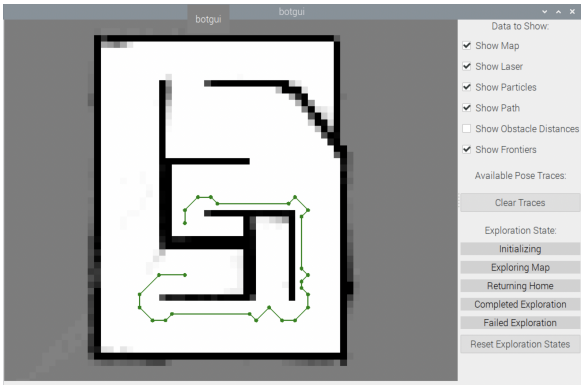

Fig. 11: Figure of test_maze_grid with path planning using A* algorithm

The test information provided by `test_maze_grid` is most useful as it is simulating the actual maze that were explored during the competition. Figure 11 shows one path successfully predicted in `test_maze_grid` from specified start and goal location, which is conform to the shortest path that avoids the obstacles.

2) Exploration: One example of Map Exploration is depicted in Figure 10, which shows the MBot returning home following the green planned path, after finishing exploring the maze.

III. Discussion and Conclusion

The non-SLAM odometry and motion controller had good accuracy that improved the robustness and ease of implementing the higher levels of control. By commanding achievable goals to the wheel controllers a higher fidelity of odometry which led to more success with SLAM. Better performance was observed when operating on hardware with this implementation than logs from previous years.

Path planning and map exploration produced a good results. The implementation of A* algorithm allows MBot to run a path efficiently leading it to the goal position, as long as the obstacles leaves a proper space for MBot to pass through. It might be necessary for us to smooth the path so that it can avoid a sharp turn when it is not needed.

The map exploration part has highly robustness as the path is updated at a fixed frequency while marching towards the nearest frontier. Still, more can be implemented on map localization with an unknown starting position.

A. Competition

1) Task 1: Task 1 was a test of the basic path following and map smearing of a consistent small map. The results can be seen in 5. Here the bot performed well in pose and was able to return to within 3cm and less than 5 degrees of error for heading. The map did experience an amount of smearing which resulted in thicker walls than expected because of the increased iterations through the map. Such smearing may be an indication for drifts in SLAM poses. The drift can also be deduced based on the increasing error shown in Figure 24. Thus, the parameters in SLAM could be further fine-tuned.

2) Task 2: This task involved the robot autonomously navigating a small maze. The robot was able to clearly navigate through the maze and explore all frontiers as well as return the the start. Seen in 12 A* was able to plan a long path back to the beginning section. The robot also returned to the starting pose within 3cm and less than 5 degrees of error for heading. the map quality was improved from task 1 and had much less smearing.

3) Task 3: This task was large maze for the robot to navigate through and return the beginning. A feature that caused some issues was the curved wall in a portion of the maze. This caused the SLAM pose to be unstable which resulted in the motion controller also being unstable. The robot was able to successfully explore all frontiers however the robot was unable to return to the start because of an error in the exploration program. The error may be again due to drift in SLAM pose and thus deviations from the actual robot location, causing the robot to drive into an obstacle. The map that was generated was 13. The curved section can be seen with the most smearing of that wall. This task did however demonstrate the robustness of the exploration and frontier searching program.

References

[1] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *Proceedings of (ICRA) International Conference on Robotics and Automation*, vol. 2, May 1999, pp. 1322 – 1328.

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

[3] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. [Online]. Available: https://doi.org/10.1109/tssc.1968.300136

[4] "theory.stanford.edu," 1968. [Online]. Available: http://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html

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

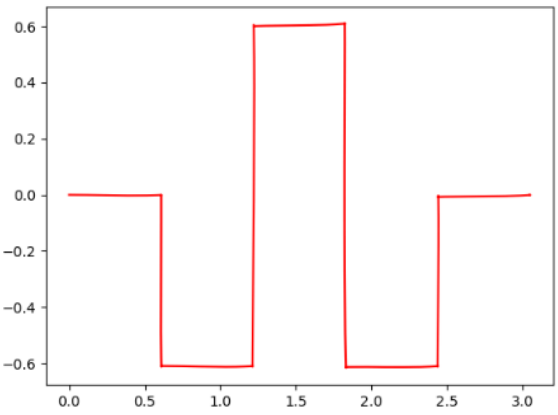

Fig. 14: Example of motion controller achieving maze run

APPENDIX

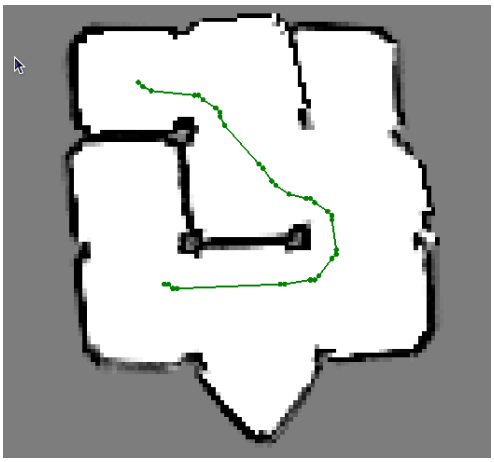

Fig. 12: Competition Task 2

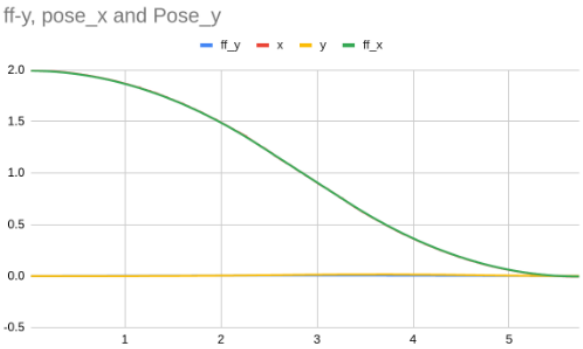

Fig. 15: Pose over time with a trapezoid controller

Please refer to below for a series of plots for the performance of 300 particles on `drive_square_10mx10m_5cm.log`.

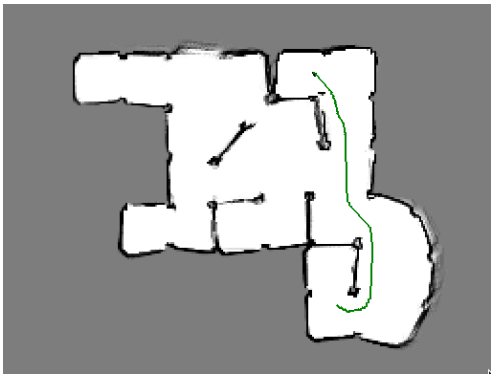

Fig. 13: Competition Task 3

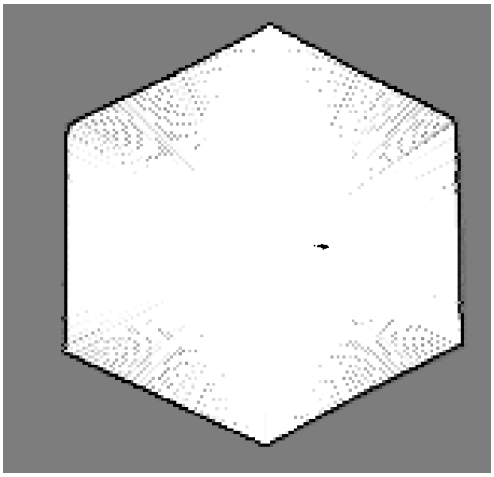

Fig. 16: Demonstration of slam particles (1)

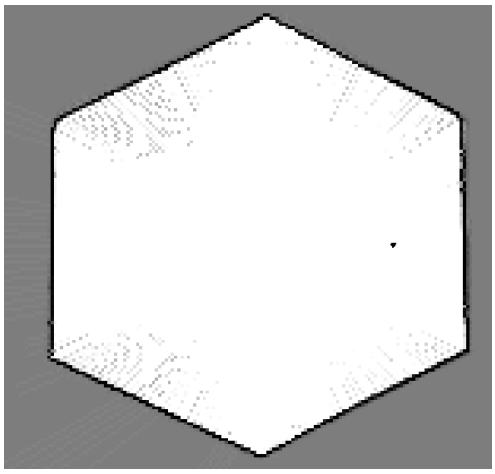

Fig. 17: Demonstration of slam particles (2)

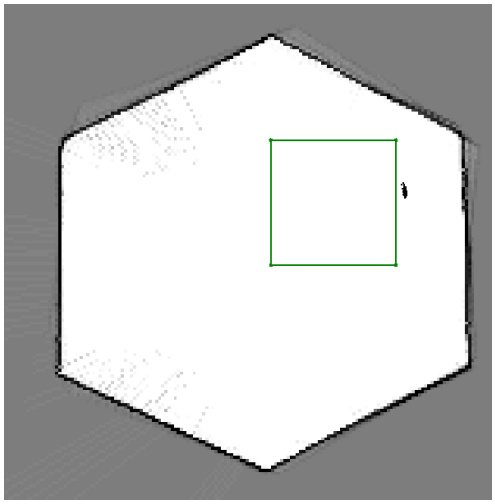

Fig. 18: Demonstration of slam particles (3)

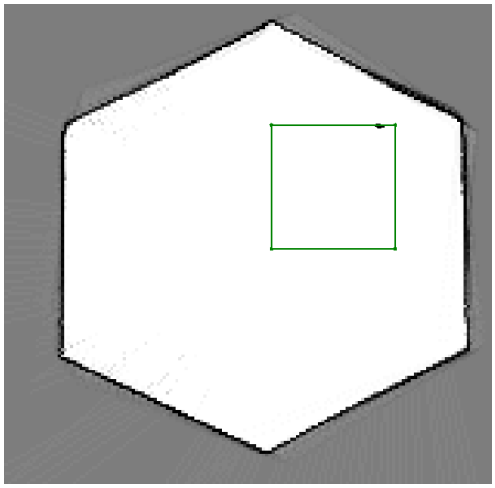

Fig. 19: Demonstration of slam particles (4)

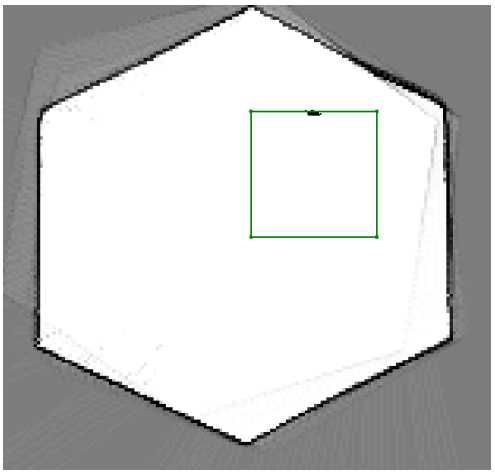

Fig. 20: Demonstration of slam particles (5)

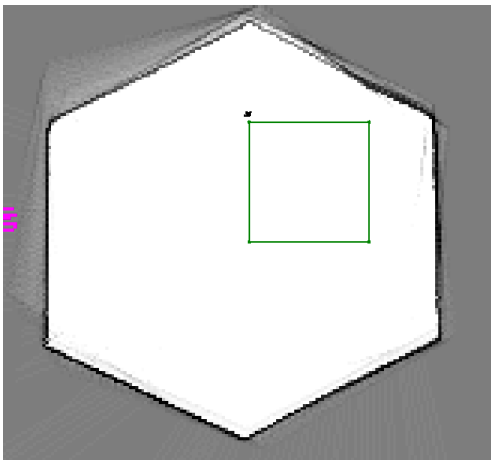

Fig. 21: Demonstration of slam particles (6

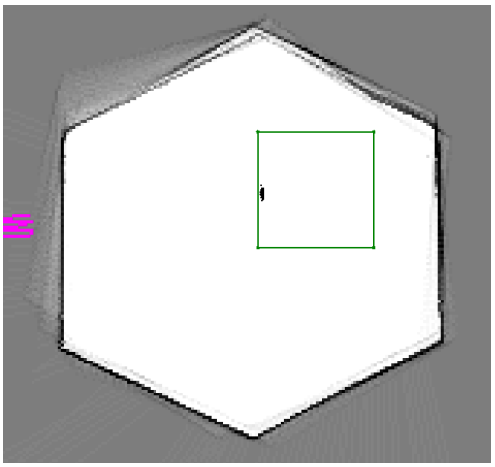

Fig. 22: Demonstration of slam particles (7)

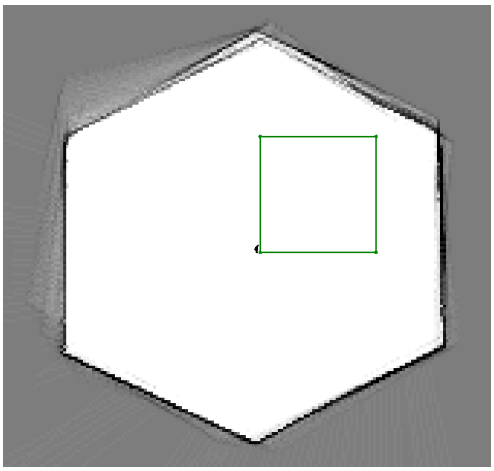

Fig. 23: Demonstration of slam particles (8)

Figure of numerical SLAM pose error compared to truth

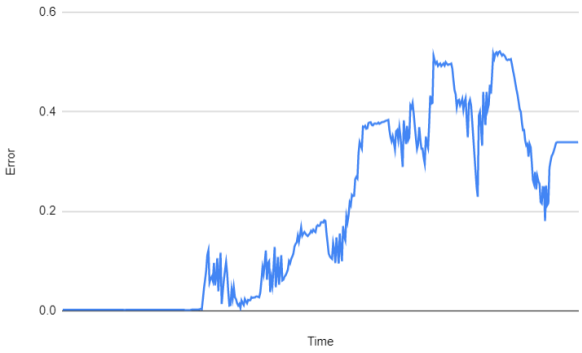

Fig. 24: SLAM pose error compared to truth