Team 6 ROB 550 BotLab Report

Bo Fu, Tor Shepherd, Nikko Van Crey, Xuran Zhao
{bofu, tors, nikkovc, xuranzh}@umich.edu

***Abstract*—Autonomous robots capable of localizing, mapping, and navigating within unknown environments are in demand to bring research outside structured laboratory environments and into relatively unstructured environments such as city streets, hospitals, and homes. This report describes the algorithms we implemented on a mobile robot to address these challenges. The system competed in events which required simultaneous localization and mapping, navigation, exploration, and block detection/manipulation. The occupancy grid mapper was able to generate grid maps at a resolution of 0.05 m, the Monte Carlo localization algorithm was able to reach a mean accuracy of 0.05 m in translation and 0.08 rad in rotation, and the system placed 3^{rd} in the BotLab Competition.**

I. Introduction

LOCALIZATION and mapping within an unknown and unstructured environment remains a challenge in robotics. This problem is relevant to planetary rovers, unmanned aerial vehicles, autonomous underwater vehicles, legged robots, self-driving cars, disaster relief robots (DRC), etc. and is often addressed with simultaneous localization and mapping (SLAM) algorithms. These algorithms leverage hardware models and information from on-board sensors to form approximate yet sufficient solutions in feasible time. Popular algorithms include the extended Kalman filter, the particle filter and FastSLAM.

The problem becomes more difficult when an agent must also navigate within the environment. This is evident in the push for autonomy in the automotive industry for which robust algorithms and on-board sensors are needed to replace the decision making and sensory feedback of a human driver. This proves difficult as drivers continuously consider a broad range of local feedback such as road conditions, street signs, obstacles, behavior of nearby agents and an internal model of vehicle limits, dynamics, response to driver input, footprint, etc. in planning navigation.

This report describes our approach to implementing SLAM, exploring and navigating within unknown environments, and capturing blocks on a mobile robot with on-board sensors. This system competed in fours events which included: 1) completing four circuits around a convex arena, 2) detection and relocation of six blocks within convex area, 3) exploration and mapping of an unknown environment, and 4) detection and relocation of 8 blocks scattered across a map.

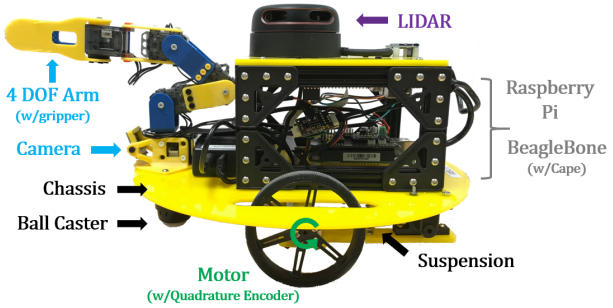


Fig. 1: SLAM, exploration, navigation, and manipulation algorithms were implemented on a mobile robot. The on-board sensors, microprocessors, motors, and chassis components are shown.

II. Methodology

A. Hardware

The hardware used throughout this report is a mobile robot equipped with two microprocessors (Raspberry Pi 3, Beaglebone Green), 2D Light Detection and Ranging (LIDAR), Inertial Measurement Unit (IMU), 4 degree of freedom (DOF) arm, and 20.4:1 Pololu metal gearmotors with integrated quadrature encoders (Fig. 4). The two wheels are mounted to each motor with shaft hubs. The chassis has two casters to balance the robot and a suspension to ensure wheels remain in contact with the ground. The stiffness of the suspension also dictates the relative contributions of the casters and wheels to supporting the robots weight. Additional springs were added to our robots suspension to increase normal force at the wheels, increase surface friction, and prevent systematic error in odometry from wheel slippage. The SLAM, navigation, exploration, and manipulation algorithms implemented on the robot are described in the following subsections.

B. Simultaneous Localization and Mapping (SLAM)

The SLAM Algorithm we chose to implement was the Monte Carlo localization (particle filter) and occupancy grid mapping. The primary steps of the particle filter include 1) Initialize n particles (x_{t-1}) with uniform weights, 2) propagate particles in time with action model, 3) update sensor measurements, 4) re-sample the distribution with non-uniform weights based on the likelihood of each particle corresponding to the sensor measurements given the current map, and 5) perform a weighted average of particles to estimate posterior pose (x_t). Steps

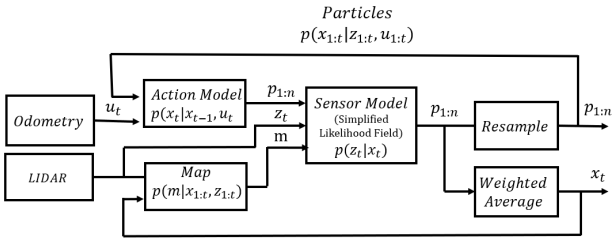


Fig. 2: Placeholder

2-5 are repeated continuously to estimate the robot pose. The n particles approximate a probability distribution of the current pose given the sensor measurement. Based on the pose estimation from the particle filter and the sensor information, the occupancy grid map is updated. A block diagram of this SLAM process is shown in Fig. 2.

1) Mapping: The mapping function $P(m|z_{1:t}, x_{1:t})$ updates the occupancy grid m representing current free space and obstacles with the knowledge of the robot pose $x_{1:t}$ and laser rangefinder measurement $z_{1:t}$. Assuming each cell is independent, the update process is conducted according to Eqn(1-3). Each cell in the occupancy grid stores the log odds $l(m_i|z_{1:t}, x_{1:t})$ of the probability that this cell is occupied, which is the recursive term in Eqn.(2); $l(m_i|z_t, x_t)$ is the inverse sensor model and $l(m_i)$ is the prior, which is set to 0, indicating no prior knowledge about a cell. In practice, the occupancy grid is presented as a grey-scale image with log odds range from -127 to 127. Each grid represents a 5 cm $\times$ 5 cm space in the world.

For each laser beam z_t^k denoting the beam k at time t, we use Breshenham's algorithm to process all the cells m_i on the beam line. Suppose r_i is the distance of cell m_i to the current robot pose. Then the inverse sensor model output is calculated according to Eqn.(4).

$$P(m|z_{1:t}, x_{1:t}) = \prod_i P(m_i|z_{1:t}, x_{1:t}) \tag{1}$$

$$\begin{aligned} l(m_i|z_{1:t}, x_{1:t}) = l(m_i|z_{1:t-1}, x_{1:t-1}) \\ + l(m_i|z_t, x_t) - l(m_i) \end{aligned} \tag{2}$$

$$l(A) = \log(\frac{p(A)}{p(\neg A)}) = \log(\frac{p(A)}{1 - p(A)}) \tag{3}$$

$$l(m_i|z_t, x_t) = \begin{cases} o_{\text{miss}}, & r_i < z_t^k \\ o_{\text{hit}}, & z_t^k \leq r_i < z_t^k + \text{cell size} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

TABLE I: Mapping parameters

parameters	o_{hit}	o_{miss}
value	1	3

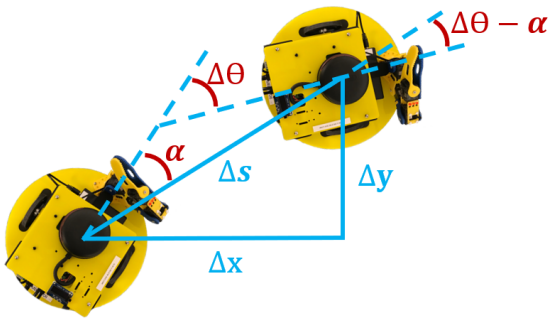


Fig. 3: Motion of the mobile robot from one pose to another is represented by an initial rotation α, translation Δs, and final rotation $\theta - \alpha$. The distance from one pose to another is Δs with vertical and horizontal displacements of Δy and Δx respectively. The angle from the robots current heading to the next pose is α and the angle between the heading the initial and final pose is $\Delta\theta$.

2) Monte Carlo Localization (MCL): The MCL is conducted according to Eqn.(5), where $P(x_t|x_{t-1}, u_t)$ is the action model, $P(z_t|x_t)$ is the sensor model, and η is the normalization factor.

$$\begin{aligned} P(x_{0:t}|z_{1:t}, u_{1:t}) = &\eta P(z_t|x_t) P(x_t|x_{t-1}, u_t) \\ &\cdot P(x_{0:t-1}|z_{1:t-1}, u_{1:t-1}) \end{aligned} \tag{5}$$

a) Action Model: The action model [1] propagates particles in time with a model of hardware motion; estimating the distribution for the next state x_t given the previous state x_{t-1} and odometry u_t. Motion of the mobile robot from one pose to another is represented by an initial rotation α, translation Δs, and final rotation $\theta - \alpha$. This method (Eq. 6-12) is computationally inexpensive and simplifies the motion planning.

$$\Delta s = \sqrt{\Delta x^2 + \Delta y^2} \tag{6}$$

$$\alpha = \text{atan2}(\Delta y, \Delta x) - \theta_{t-1} \tag{7}$$

$$u_t = [\alpha \quad \Delta s \quad \theta - \alpha] \tag{8}$$

$$\epsilon_1 \sim N(0, \min(k_1|\alpha| + k_2\Delta s, 2|\Delta\theta|)) \tag{9}$$

$$\epsilon_2 \sim N(0, k_3\Delta s + k_4|\theta|) \tag{10}$$

$$\epsilon_3 \sim N(0, \min(k_1|\theta - \alpha| + k_2\Delta s, 2|\Delta\theta|)) \tag{11}$$

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} (\Delta s + \epsilon_2)\cos(\theta_{t-1} + \alpha + \epsilon_1) \\ (\Delta s + \epsilon_2)\sin(\theta_{t-1} + \alpha + \epsilon_1) \\ \Delta\theta + \epsilon_1 + \epsilon_3 \end{bmatrix} \tag{12}$$

The model errors $\epsilon_1, \epsilon_2, \epsilon_3$ are represented as Gaussian distributions, where the constants k_1, k_2, k_3, k_4 represent model uncertainty. k_2 and k_4 represent cross-correlation. To choose these uncertainty parameters, the localization algorithm is run on the log file of drive square (drive_square_10mx10m_5cm.log) and obstacle SLAM (obstacle_slam_10mx10m_5cm.log), and tune the parameters such that the error between the SLAM pose and the ground truth is minimized. The fixed parameters

are shown in TABLE II. Notice that in Eqn.(9, 11), there is a min function; the importance of this is discussed in Sec.III-A2a. In short, this allows the robot to move in reverse.

TABLE II: Action Model Uncertainty Parameters

Parameter	k_1	k_2	k_3	k_4
Value	0.3	0.3 rad/m	0.2	0.05 m/rad

b) Sensor Model: The sensor model adds weights $P(z_t|x_t)$ to each particle after the action model is applied. This weight is added according to the current location x_t of a particle and the sensor reading z_t. The "likelihood model" is selected over the "beam model" due to the lower computational cost. The weights are calculated according to Eqn.(13 - 15). For particle pose x_t, for each laser beam $z_t^k \in z_t$, we draw the endpoint of the beam based on the particle pose on the map. In the likely case that the endpoint is at an obstacle, the weight is larger. If the endpoint is at a free space, this is less likely to happen. The probability of the endpoint after a obstacle is the lowest. Therefore the $s_{\text{hit}} = 3.0$, $s_{\text{miss}} = 1.5$, and $s_{\text{behind}} = 0.5$.

$$P(z_t|x_t) = \prod_k P(z_t^k|x_t) \tag{13}$$

$$log(P(z_t|x_t)) = \sum_k log(P(z_t^k|x_t)) \tag{14}$$

$$log(P(z_t^k|x_t)) = \begin{cases} s_{\text{hit}}, \text{endpoint is on an obstacle} \\ s_{\text{miss}}, \text{endpoint is a free space} \\ s_{\text{behind}}, \text{endpoint is behind an obstacle} \end{cases} \tag{15}$$

Part of the laser beam readings are bad sensing data. If a beam reading is larger than the maximum laser range, smaller than the robot radius, or the endpoint is outside the map, we regard it as a bad data. Instead of $log(P(z_t^k|x_t)) = 0$, as this means even lower weights than s_{behind}, we simply ignore this data and use Eqn.(16) as the final weight. N is the number of usable data.

$$log(P(z_t|x_t)) = 1/N \cdot \sum_k log(P(z_t^k|x_t)) \tag{16}$$

c) Low Variance Resampling: Due to its numerical stability, we choose $log(P(z_t|x_t))$ instead of $P(z_t|x_t)$ as the weight of each particle; the low variance resampling method is implemented using this weight.

d) Pose Estimation: For pose estimation, we take the average of the best 2% particle poses, instead of choosing the best particle. This reduce the noise in pose estimation, and end up with a smoother trajectory.

e) Lidar Odometry: The u_t in Sec.II-B2a, from the IMU and wheel odometry system is occasionally far from the ground truth value due to non-systematic errors such as wheel slippage. When the wheel speed is abnormally large or the best particle weight is too small (as usually it is close to s_{hit}), we assume non-systematic error is introduced and the "lidar odometry" is triggered. We perform an exhaustive search around the previous pose and the pose with best score $\log(P(z_t|x_t))$ is recorded as the initial current pose. This pose is used instead of the odometry pose from the action model update. The region of lidar odometry is ± 0.07 m for x and y, and ± 0.25 rad for θ. And we evaluate 1000 points in this region to decide the "lidar odometry" value. According to experiment results, this method supports the MCL even without odometry input.

f) Global Localization: For the kidnapped robot problem, where a robot loses its pose during the SLAM process, the SLAM algorithm first terminates the mapping branch and does localization only. Secondly, the particles are distributed evenly in the free space region of the current occupancy grid map. Then MCL is used to update the particle distribution as the robot moves around the map. The particle distribution will converge as the robot keeps moving (5 seconds according to experiment). When there's only one cluster of particles, the global re-localization is completed, and the mapping stream is turned on. The number of particles needed for this global localization is much larger than the local localization used in the previous chapter, and depends on the the map size. To reduce the computation cost, we implemented an adaptive particle number which decreases with time. The result for this part is not in Sec.III, please refer to the video we provided.

C. Planning

A* algorithm was implemented to design optimal trajectories within the work space because it has the lowest time-complexity possible while guaranteeing an optimal path. A discretized map of the workspace and robot pose are estimated by SLAM and considered in the A* algorithm. Our algorithm is a 3D A* with robot angle θ as a third search dimension in computing costs h and g Eqn.(17,19). Turning is not penalized in 2D A*, so paths that have many turns and take a long time to execute in practice will have same cost as long-straight paths with minimal turning, as long as the distance travelled are the same. Considering θ allows A* to penalize trajectories with many turns. The benefits of 3D A* are discussed in Sec.III-A2a.

Suppose k is the next state and $k-1$ is the current state, the cost and heuristic of the next state is calculated according to Eqn.(17-19). $d(\cdot,\cdot)$ and $\psi(\cdot,\cdot)$ calculate the

translation and angle distances between two states respectively. p_k is the penalty on of the cell k if its distance to the nearest obstacle is within $d_{\text{threshold}} = 1.5$m.

$$g_k = g_{k-1} + d(k, k-1) + \psi(k, k-1) + (p_k + p_{k-1})/2 \tag{17}$$

$$p_k = \begin{cases} 0, & d(k, \text{obstacle}) \geq d_{\text{threshold}} \\ 0.5(d_{\text{threshold}} - d(k, \text{obstacle}))^2, & \text{else} \end{cases} \tag{18}$$

$$h_k = w \cdot d(k, k_{\text{goal}}) + \psi(k, k_{\text{goal}}) \tag{19}$$

D. Motion Controller

The parameters of the motion controller remained are unchanged from the script provided, but three major changes were made to the control algorithm.

1) The rotation-drive process is reduced to drive for the way-points other than the last one; I.e. the rotation state is eliminated. Instead, the forward speed is reduced linearly w.r.t. the angle error when this error is larger than a threshold. Therefore, the angle and position are controlled simultaneously, which results in smoother trajectory following performance.

2) The robot is allowed to drive backward. When the angle error is larger than $\pi/2$ or smaller than $-\pi/2$, the backward movement is enabled. This reduces the angle the robot needs to turn and results in smoother and more flexible movement.

3) A path type variable is added to the path message to determine the control strategy used.

- 0: Allow backward movement, no heading (yaw angle) control (only control the robot to reach a position).
- 1: Forward movement only, no heading control.
- 2: Forward movement only, control heading when going to the last way-point.

E. Exploration

The exploration of an unknown map can be reduced to the task of eliminating all the reachable frontiers in the map. A frontier is defined as a unexplored cell in the current occupancy grid map. To find this cell, a Breadth First Search (BFS) is started from the current cell (where the robot is at), once it reaches a frontier, that is the current nearest frontier. The complexity of this search is $O(n)$ where n is the number of the cells in the map.

The exploration process is shown in Algorithm 1 in Appendix. At each iteration, the map is updated first, if current frontier is no longer a frontier, a new frontier is obtained and the robot is asked to drive to the new frontier; if instead, there's no new frontier, the robots is asked to return to home and complete the exploration. Notice that, because the robot is exploring a map, therefore, a planned path that is valid in previous iteration might no longer be valid in current iteration. Therefore, if a path is no longer valid, a re-planning is conducted.

F. Block Capture

1) Grabbing Machanism: The mobile robot has a 4DOF arm with a pinch gripper mounted to the chassis for block capture and relocation in the *Simple Block Pickup* and *Treasure Race* events. We outfitted the gripper with silicone rubber to increase friction and ensure secure grasps. During the ArmLab Van Crey et al. showed that silicone rubber outperformed the other compliant-high friction materials available.

TABLE III: Robot Arm Physical Parameters

Parameter	l_1	l_2	l_3	l_4	d_{c2a}
Value (m)	0.056	0.054	0.054	0.112	0.023

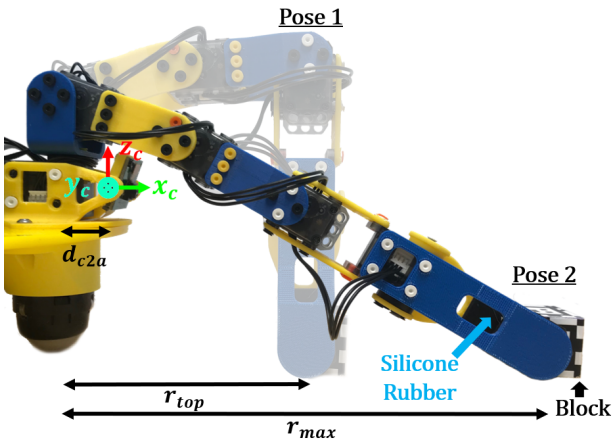


Fig. 4: Robot arm: two block grabbing poses

A camera is located at the proximal end of the arm to detect AprilTags on the faces of the blocks. AprilTag is a visual fiducial system that computes precise 6D pose within the camera coordinate frame (x_c, y_c, θ_c) from known tag class/size. We transform these coordinates into the arm frame (x_a, y_a, θ_a) and the mobile robot frame (x_r, y_r, θ_r). This system has difficulty detecting the blocks when the robot is in motion, so we chose to stop the robot while locating and manipulating blocks. Within the dungeon the blocks are expected to be located within convex corridors. To find candidate block locations within the map we construct a discretized potential map and check for local maxima along three directions (three walls). The robot will stop at a survey distance 0.2 m from the candidate location after reaching each target pose.

At each candidate location the arm module will identify any block in the current view and communicate the state over lcm to the explorer. If no message is received

from arm module with 3 seconds, the explorer will send the path to the next candidate location. If there are blocks detected within the reachable workspace of radius 0.2 m from the arm frame, the arm controller will capture the first block detected. The gripper will grab the block from directly above if the block is located within radius 0.14 m and from the side if 0.14 m$< r < 0.20$ m.

2) Arm State Machine: For block capture tasks, the mobile robot is required to switch between different modes such as map traversal, block detection, and block manipulation. These tasks require the actions of the arm to synchronised with the exploration module. This is accomplished with a state machine (Algorithm 2 in the Appendix) containing five states (*READY_FOR_NEXT*, *DRIVE_TO_POSE*, *GRAB_BLOCK*, *RETURN_HOME* and *RELEASE_BLOCK*).

The arm module also has an internal state machine for block detection and manipulation procedures. If the block is located outside the reachable workspace then the robot is too far from the block to capture it and must relocate. When a block is detected visual servoing is used to approach the block gradually. This method directly commands small rotations ($0.08\ rad$) or translations ($0.05\ m$), with the Apriltag of interest constrained within the centre range of the image.

3) Explorer State Machine: The exploration module requests from a pose queue and arm module for next target pose. The exploration module gives priority to the arm module when a block is detected in the current frame and resumes control after the grabbing process is finished, proceeding travel to next pose in the queue.

III. Results and Discussion

A. Simultaneous Localization and Mapping (SLAM)

1) Mapping: The occupancy grid mapping was evaluated with two test cases: 1) driving a square in a convex region, and 2) driving in a region with two rectangular obstacles. The pose information was provided from motion capture data, to serve as ground truth and test the mapping algorithm independent of the odometry. The mapping algorithm was able to generate maps with clear edges (Fig. 5).

2) Monte Carlo Localization:

a) Action Model: The action model results for pure translation, pure rotation, and a combination of translation and rotatation are shown in Fig. 6a, 6b, and 6c respectively. The particles are shown to translate in pure rotation and vice versa. These results indicate that the translation and rotation have nonzero co-variance.

In Fig. 6a, 6b, and 6c the robot was moving forward. Our robot also moves backward, however, and requires a angle limitation process. The results for the robot moving backwards with and without limitation are shown in Fig. 6d and Fig. 6e. Even when the robot moves backward with no rotation ($\Delta\theta = 0$ rad), without limitation $\alpha = \pi$ and $\Delta\theta - \alpha = -\pi$. These huge angles do not occur in practice and cause the particle distribution to "explode" in just one iteration (6d). While with limitation (6e), the distribution still reflects the odometry input. These results show the importance of the angle limitation process.

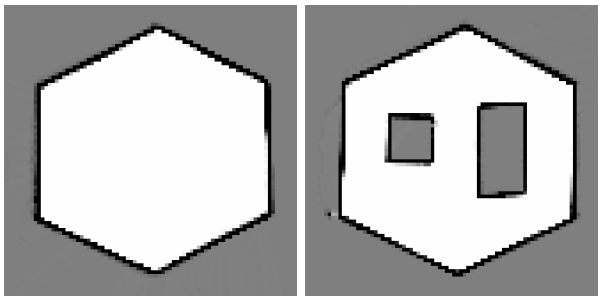


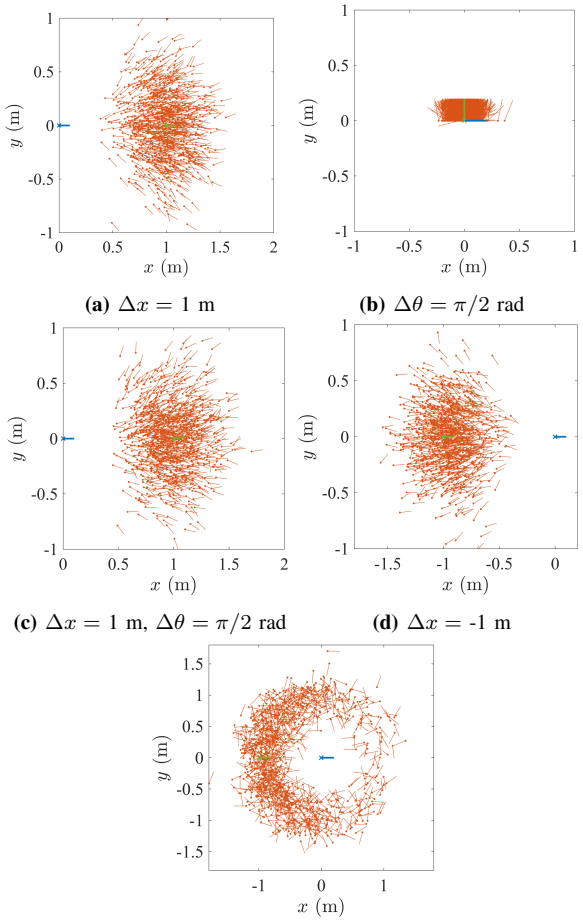
Fig. 5: Occupancy grid mapping results. Left: drive square in a convex region. Right: drive in a region with obstacle. The pose information is from the log file, and only the mapping algorithm is running.

b) Particle Filter: The Monte Carlo localization algorithm is first evaluated in the drive square test with 300 particle. The distribution at the midpoint of each 1 m translation and at the corners of each $\pi/2$ rad rotation are plotted and shown in Fig. 7. The distribution represents a Gaussian distribution around the MCL pose.

The accuracy of the MCL process was evaluated using pure localization mode for two test cases: 1) drive square and 2) drive in a region with obstacles. The resultant trajectories (Fig. 8), error statistics (TABLE IV) and error plots (Fig. 10,11) are shown below. The odometry input deviates from the ground truth, yet the MCL algorithm is able to output a position with accuracy around the map resolution of 0.05 m (TABLE IV). Also, while the odometry error become larger w.r.t. the time (Fig. 10, and Fig. 11), the SLAM poses do not have this trend.

As can be seen in Fig. 8, the shape of odometry trajectory is close to the ground truth pose, however, the distance error is large. Therefore, the Absolute Trajectory Error (ATE) [2], which evaluates the alignment error between two trajectories, is also calculated. According to TABLE V, the MCL results in half of the ATE as compared to odometry. The aligned trajectories are shown in Fig. 9.

The computation cost of the MCL is evaluated on raspberry pi with 100, 300, 500, and 1000 particles, the timing result is shown below. It's noticed that the computation cost of each components of the particle filter is: senser model $>$ action model $>$ low variance re-sampling $\approx$ pose estimation. In fact, they are all



(a) $\Delta x = 1$ m

(b) $\Delta\theta = \pi/2$ rad

(c) $\Delta x = 1$ m, $\Delta\theta = \pi/2$ rad

(d) $\Delta x =$ -1 m

(e) $\Delta x =$ -1 m, without limitation

Fig. 6: Action model results. Blue: previous pose, green: current pose according to odometry, orange: particle poses.

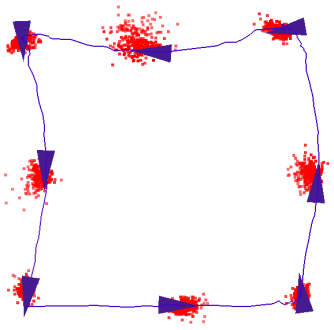


Fig. 7: Monte Carlo localization particle distribution during drive square test.

$O(n)$ process (n is the number of particles), while with different time for processing each particle. In sensor model, each particle needs to loop through all the laser rays; while in action model, there's several steps of mathematical operations. The time in TABLE VI roughly reflects this linear relationship ($O(n)$). Assuming this

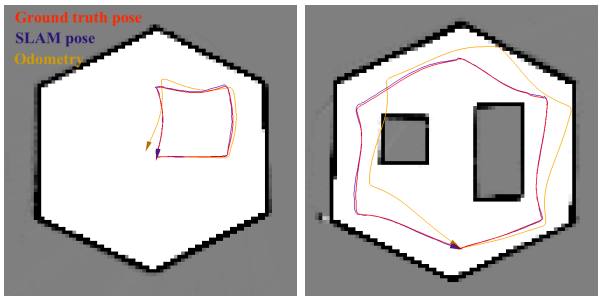


Fig. 8: Monte Carlo localization results. Left: drive square in a convex region. Right: drive in a region with obstacle. The map is from the saved map, and only the Monte Carlo localization is running.

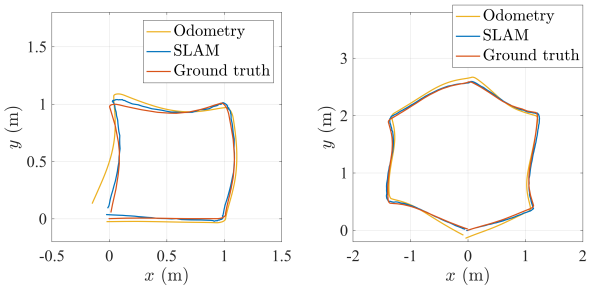
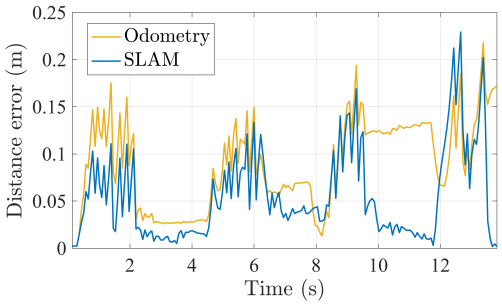
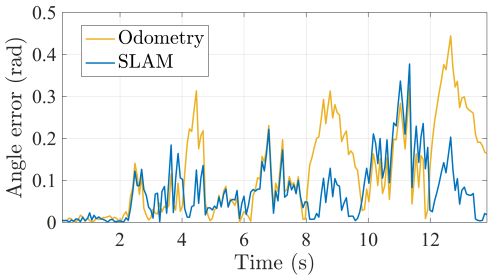


Fig. 9: Monte Carlo localization with pose aligned. Left: drive square in a convex region. Right: drive in a region with obstacle.



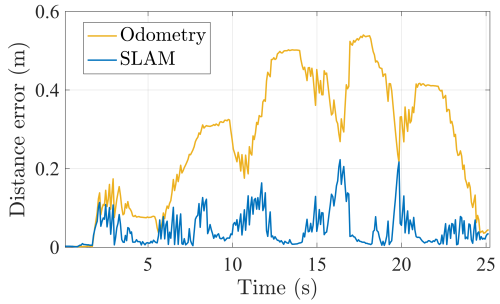
(a) Distance error



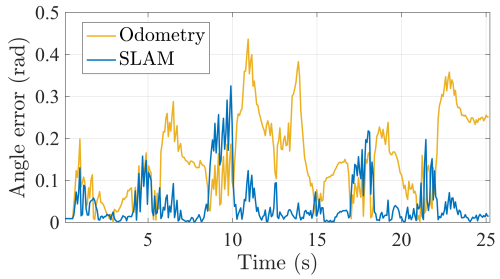
(b) Angle error

Fig. 10: Localization error (drive square test).

linear relationship, the maximum number of particles this MCL can support running at 10 Hz on the RPi is 860.



(a) Distance error



(b) Angle error

Fig. 11: Localization error (region with obstacle test).

TABLE IV: Localization error. DS: drive square, RO: region with obstacle.

Category	Mean	σ	Min	Max
DS distance error (m)	0.056	0.047	0.002	0.229
DS angle error (rad)	0.076	0.071	0.000	0.377
RO distance error (m)	0.046	0.043	0.001	0.223
RO angle error (rad)	0.046	0.057	0.000	0.325

3) SLAM: Finally, the occupancy grid mapping and the MCL are combined and running simultaneously and the SLAM result of the drive square and region with obstacle test cases are shown in Fig. 12. According to the figure, the map and localization result maintains the accuracy as compared to the case where they are running separately.

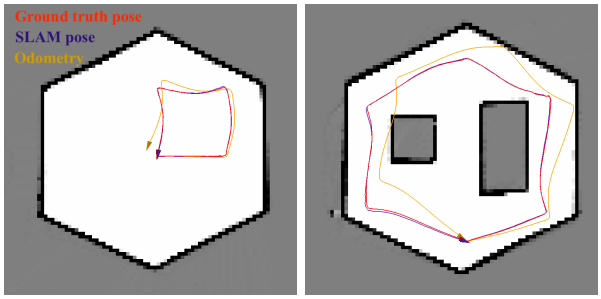


Fig. 12: SLAM results. Left: drive square in a convex region. Right: drive in a region with obstacle. Both the Monte Carlo localization and occupancy grid mapping algorithms are running.

TABLE V: Localization ATE (unit: m)

Test case	Odometry	SLAM
Drive square	0.084	0.047
Region with obstacle	0.092	0.040

TABLE VI: Particle filter computation cost.

Particle number	100	300	500	1000
Time (s)	0.0141	0.0331	0.0547	0.1176
Frequency (Hz)	70.79	30.19	18.27	8.50

B. Path Planning

The resultant paths of 3D A* have fewer turns and allow the robot to quickly traverse the map (Fig. 13).

We experimented with a weighted 3D A* ($1 < w \leq 20$, Eq. 19), which expands nodes closer to the goal first, and found the results consistent with unweighted A* and orders of magnitude lower computation time. The 3D A* conflicts with Weighted A*, however, because the algorithm highly weights trajectories which stay close to the end orientation (Figure 14). For competition we used unweighted 3D A*.

The time taken by 2D A*, 2D Weighted A* with $w = 20$ (WA*), 3D A*, and 3D Weighted A* with $w = 20$ are listed in table VII. 3D A* has the slowest computation time, but yields more optimal results. Future work includes fixing the suboptimality of 3D weighted A* since it is faster than 3D A*.

One example path generated by 3D unweighted A* is shown in figure 15 with the robot's trajectory. The waypoints are sparse due from removing waypoints in straight-line sections. The result is a smoother path with fewer stops.

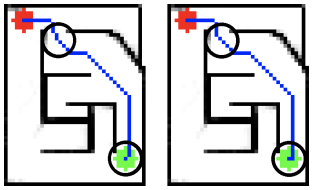


Fig. 13: Left: Standard two-dimensional A*. Right: Incorporating robot angle results in fewer turns in circled areas.

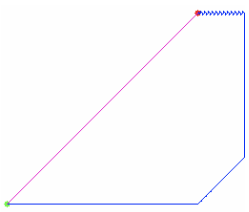


Fig. 14: Optimal path (magenta) result from A*, weighted 2D A*($w = 20$), and 3D A*. Sub-optimal path (blue) from weighted 3D A*($w = 20$).

TABLE VII: A* Computation Times (microseconds)

Name	2D A*	2D WA*	3D A*	3D WA*
empty_grid	509	562	3270	5479
narrow_cons_grid	1611	516	10135	4246
wide_cons_grid	5061	502	59493	5072
convex_grid	574	160	5836	1420
maze_grid	354	310	4958	3389

Fig. 15: Planned path and the executed trajectory. Green: planned path, yellow: SLAM pose.

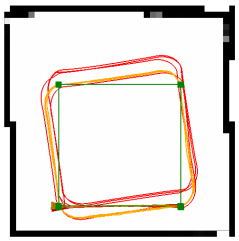


Fig. 16: Drive a square. Green: planned square path, yellow: SLAM pose, red: odometry pose.

C. Exploration

The exploration logic is run on the robot and two map exploration result is shown in Fig. 15 and Fig. 17. The robot is able to find the nearest frontier, explore the map, and finally return to the home pose. The map generated is clear.

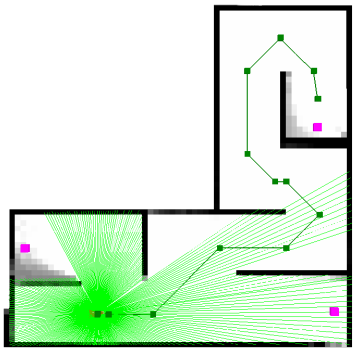


Fig. 17: Exploration. Dark green: return home path, light green: laser scan, magenta: location of blocks.

IV. Competition

Loopers: In this task, the robot was required to complete a square trajectory four times within error constraint ($\pm$ 3 cm, $\pm$ 15 °). This was impossible due to the error incurred by rotation as the last movement, so the robot is commanded to move one extra edge, then reverse onto the goal position. Its path and map generated could be observed in Fig. 16, and it managed to stay within the constraint at the final pose.

Simple Block Picking Up: Six blocks are placed in front of the robot within a closed convex arena. The robot managed to pick up three out of six blocks in this task. However, it took the robot around 10 minutes to complete it due to the visual servo approach applied and lagged 1 cm communication between arm and motion controller. It is conjectured that the delay results mainly from Apritag detection with high resolution frame on $RaspberryPi$.

Dungeon Explorer: In this task, the robot should explore the maze, record a map and return to the start points. The robot completed the task smoothly. Though due to the fact the SLAM algorithm is running on the team's laptop that communicate with robot through 1 cm tunnel, the lidar messages might be dropped from time to time, which destroyed the constructed map.

Treasure Race: Three blocks are hidden in the same maze as in the last task. The robot must utilize the map constructed in the previous task to searching and retrieve blocks. The robot was unable to grab the first block, however, for reasons similar to *Simple Block Pickup*. The arm module detected the blocks and took control of the motion controller from a far distance, but the movement towards the block was extremely slow due to communication lag.

V. Conclusion

In this paper, the design and implementation of SLAM, path planning, exploration and block retrieval function for a mobile robot is presented. The robot is equipped with `BeagleBone Green` for motion control and `Raspberry Pi` for SLAM and arm module. With these subsystems, the robot is capable of exploring an unknown environment, constructing a map of the environment and retrieving blocks in it. The occupancy grid mapper in the SLAM system is able to generate grid maps at the resolution of 0.05 m, the Monte Carlo localization algorithm is able to reach a mean accuracy of 0.05 m in translation and 0.08 rad in rotation, and the system placed 3^{rd} in the BotLab Competition. For future improvements, the integration between the arm module and exploration function can be modified, with dynamic resolution for the image capturing and more efficient algorithm for approaching detected block.

VI. Acknowledgements

The authors would like to thank Dr. Peter Gaskell for the informative lectures throughout the semester, and the GSIs Mohieddine Amine and Ji Hwang (Henry) Kim for helping debug hardware issues during office hours.

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.

[2] R. Kümmerle, B. Steder, C. Dornhege, M. Ruhnke, G. Grisetti, C. Stachniss, and A. Kleiner, "On measuring the accuracy of slam algorithms," *Autonomous Robots*, vol. 27, no. 4, p. 387, 2009.

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

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

VII. Appendix

Algorithm 1 Explore An Unknown Map

```
map = updateMap()
frontier = newFrontier(map)
path = planPathTo(frontier, map)
while frontier ≠ ∅ & path ≠ ∅ do
    map = updateMap()
    if !isfrontier(frontier, map) then
        frontier = newFrontier(map)
        if frontier ≠ ∅ then
            path = planPathTo(frontier, map)
        else
            path = planPathTo(home, map)
        end
    else
        if !isValidPath(path, map) then
            path = planPathTo(frontier, map)
        end
    end
    excutePath(path)
end
```

Algorithm 2 Robot Arm State Machine For Block Capture

```
blocks = updateDetectedBlocks()
while blocks ≠ ∅ do
    blocks = updateDetectedBlocks()
    curState = updateCurState()
    switch curState do
        case BLOCK_OUT_OF_RANGE do
            moveTowardsBlock() arm → explorer : grabBlock
        end
        case BLOCK_IN_RANGE do
            confirmBlock()
        end
        case GRAB_BLOCK do
            triggerArmGrabbing()
            arm → explorer : returnHome
        end
        case RELEASE_BLOCK do
            triggerArmReleasing()
            arm → explorer : readyForNext
        end
    end
end
```

Algorithm 3 Explorer State Machine For Block Capture

```
while poses ≠ ∅  blocks ≠ ∅ do
    pose = nextPose(pose, blocks)
    curState = updateCurState()
    switch curState do
        case GRABBING_BLOCK do
            doNothing()
        end
        case DRIVE_TO_POSE do
            sendPathToPose()
            explorer → arm : driveToPose
        end
        case RETURN_HOME do
            driveBack()
            if homeReached() then
                explorer → arm : releaseBlock
            end
        end
    end
end
```