Team 14 ROB 550 BotLab Report

Xiangyu Peng, Xi Lin, Steven Schulte, Nalin Bendapudi
{xypeng, bexilin, spschul, bnalin}@umich.edu

***Abstract*— This report chronicles our work on a differential drive ground robot, and the methodology adopted to make it simultaneously map and localize itself in an unknown territory. The motion controller for this robot was used odometry signals from IMU and encoders. A LIDAR sensor was used to do SLAM. SLAM involved mapping using Breshenham's Algorithm and localization using a particle filter. The robot could also autonomously search for new frontiers on the map and plan a path to the frontiers using A-star path planning.**

***Index Terms*—Monte-Carlo localization, Mapping, Particle-filter SLAM, A-star Planning, Exploration**

I. Introduction

RESEARCH on ground-based autonomous systems has been an evergreen topic in robotics. While the control of differential drive robots has become an easy task, mapping of uncertain environments and simultaneous localization of a robotic system still poses many interesting challenges. The applications of Simultaneous Localization and Mapping (SLAM) based systems range from defence to space.

In this project, we implement SLAM on a bot and use it to navigate through a maze. The bot searches for new frontiers on the map and plans a path towards them in order to explore them. This project also included various other tasks such as tuning the motion controller of the bot, implementing a state machine for the arm on the robot, and integrating it with the SLAM and motion planner.

The rest of the report is organised as follows. In section II we discuss the action model, sensor model and particle filter used in our localization,and we also discuss the algorithms used in mapping. In this section we report the effectiveness of our SLAM when implemented on Raspberry Pi. In section III we discuss our planning and exploration algorithms. Finally, in section IV we discuss our performance in various competition tasks.

II. Simultaneous Localization and Mapping

In this section, we introduce the various components of our SLAM implementation. The block diagram in Fig. 1 shows the overall picture.

A. Mapping

The mapping algorithm relies on Bresenham's algorithm, a technique for finding the discrete cells that lie on the path of a ray (in this case, a ray from a Lidar scan). This algorithm is presented in Algorithm 1. Bresenham's algorithm produces a list of all of the cells that each ray passes through, and the map log odds are updated so that the spaces that the ray passes through (which are likely free) are more likely to be free and the spaces where the ray terminates (likely some kind of obstacle blocking the path of the light) are updated so that they are more likely to be occupied.

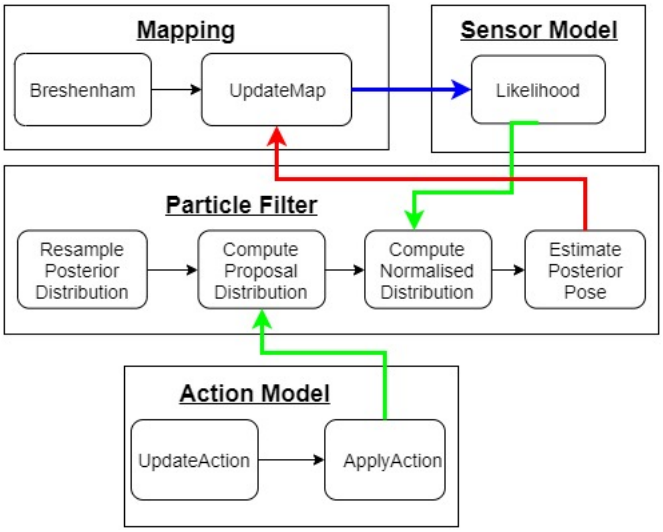

Fig. 1: Block Diagram of SLAM

The occupancy-grid mapping algorithm

Using only mapping on the log file `obstacle_slam_10mx10m_5cm.log`, the mapping algorithm produced the map shown in Fig. 2.

B. Monte Carlo Localization

We perform a version of Monte Carlo Localization called particle filtering. Here, we explain first the action model and sensor models that are used to estimate the motion of the robot and the way the sensors provide data given a fixed map, respectively, and then explain more about how the particle filter uses those models to localize.

1) Action Model: In order to implement Monte Carlo localization (discussed later), we needed to implement an action model, which is a model for how the robot moves when taking an action. In this case, we use

Algorithm 1 Bresenham's Algorithm

```
function Bresenham(start, end, grid)
    dx = abs(start.x-end.x)
    dy = abs(start.y-end.y)
    sx = abs(start.x < end.x? 1 : −1)
    sy = abs(start.y < end.y? 1 : −1)
    err = dx-dy
    x=start.x
    y=start.y
    cells=ϕ
    while(x!=x1 or y!=y1)
        cells.pushBack(cell(x,y))
        e2 = 2×err
        if e2 >= −dy
            err -= dy
            x += sx
        if e2 <= dx
            err += dx
            y += sy
    return cells
```

Algorithm 2 Occupancy-Grid Mapping

```
function Mapping(scan, pose, map)
    for all rays in scan do
        start_point = GlobalToGrid(pose.x, pose.y)
        end_point=start_point+range×(cosθ, sinθ)
        cells=Bresenham(start_point,end_point,grid)
        for all cell in cells except last_cell do
            cell.logOdds=cell.logOdds - missOdds
        for last_cell in cells do
            cell.logOdds=cell.logOdds + hitOdds
    return map
```

odometry and model each movement of the robot as a rotation α, followed by a translation along the direction of the robot Δs, followed by another rotation β. Since the total orientation change $\Delta\theta = \alpha + \beta$, the second rotation command could be expressed as $\Delta\theta - \alpha$. By combining many such motions at small time scales, the model attempts to smoothly follow the motion of the robot. The pose update computations using action model are shown in equation (1).

The action model also assumes that the motion of the robot will be perturbed by noises, which are described in equation (2). The noises ϵ_1, ϵ_2 and ϵ_3 are drawn from normal distributions that have zero means, and variances related to the magnitude of motions and coefficient k_1, k_2, shown in Table I. These values were chosen because they provided a distribution of particles that was distributed fairly closely around the actual value, while being distributed enough around the pose that we considered it to be sufficiently robust in the event of abrupt changes.

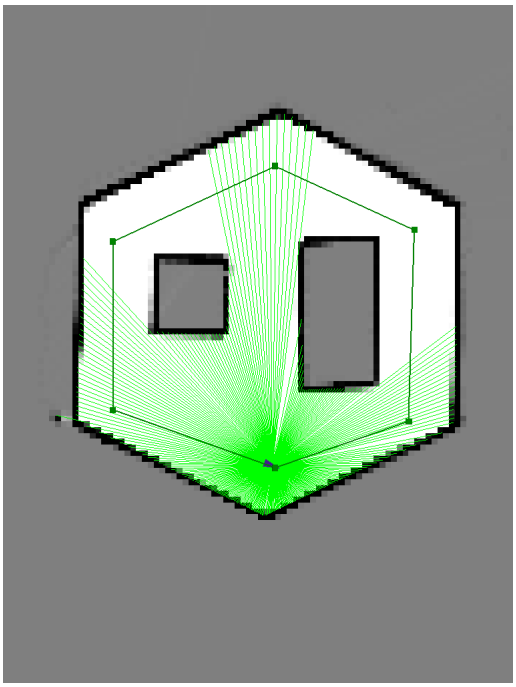

Fig. 2: The map produced by the robot's occupancy grid mapping algorithm, with knowledge of its true position, on obstacle_slam_10mx10m_5cm.log.

TABLE I: k_1 and k_2

k_1	k_2
.1	.1

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

$$\begin{cases} \epsilon_1 \sim N(0, k_1|\alpha|) \\ \epsilon_2 \sim N(0, k_2|\Delta s|) \\ \epsilon_3 \sim N(0, k_1|\Delta\theta - \alpha|) \end{cases} \tag{2}$$

2) Sensor Model: For an algorithm to understand how the evidence of sensors should influence its estimate of the robot pose, it requires a model of the sensor: a way to get the probability of the sensor providing a certain output given that the world (in this case, the map and pose) are known. To do so, we use a sensor model that produces the log-odds of a sensor measurement. We used the simplified likelihood field model given in lecture, presented in Algorithm 3. Note that the algorithm allows some leeway when a terminal cell is not a wall but adjacent cells are by adding log-odds of those cells weighted by .8. While we did not do any extensive tests, we found that this modification can help to nudge the distribution towards an appropriate distribution by

weighting particles close to matching the sensor data more than particles that are simply completely wrong.

Algorithm 3 Simplified Likelihood Field Model

function SensorModel(z_t, x_t, m)
$odds$ = 0
for ray in z_t **do**
  if ray terminates in cell with log-odd > 0 **then**
    $odds = odds+$ terminal cell log-odds
  else
    if cell after terminal cell along ray has log-odds > 0 **then**
      $odds = odds+$ after-terminal-cell log odds $*.8$
    else if cell before terminal cell along ray has log-odds > 0 **then**
      $odds = odds+$ before-terminal-cell log odds $*.8$
    end if
  end if
  return $odds$

3) Particle Filter: In this lab, we employed a particle filter for localization. A particle filter is essentially an approximation of Bayesian estimation which models the state of a robot as a weighted distribution of "particles", where each particle represents a probable state or pose of the robot. The filter works by iteratively applying an action model to all the particles, which creates a new distribution approximating the distribution of possible next states, and then accounting for the sensor measurement and updating all of the weights of the particles according to how likely they were to have produced the sensor measurement observed. A pose estimate for practical use is generated based on the weights of the distributions. This cycle repeats again with the creation of another distribution created by low-variance re-sampling from the old distribution based on the weights of each particle. The low-variance sampling is important: while we didn't explicitly test this, prior experience with particle filters has indicated that without low-variance sampling, it becomes easy for most of the particles to simply be near-copies of a single particle, resulting in a distribution that is overly confident that it is in a certain spot.

Computation time is of essence when applying particle filters. It is obvious that increasing particle numbers can make the prediction more accurate while rising the computation time at the same time. Thus, selecting appropriate numbers for particle filters can help with the efficiency. In Table II, we showed how computation time changes according to the particle numbers and the trend is shown in Fig. 3. We estimated that it will take about 0.1s to update particle filters with 1470 samples, so that the maximum particle numbers we can implement is 1470 at 10Hz on the RPi.

Algorithm 4 Particle Filter Algorithm

function ParticleFilter ($scan$, $particlePoses$, $particleWeights$, $odometryUpdate$, map)
  for m = 1 **to** $numParticles$:
    pick particle i with probability $particleWights[i]$
  for m = 1 **to** $numParticles$:
    ActionModel($particlePoses[m]$, $odometryUpdate$)
  for m = 1 **to** $numParticles$:
    SensorModel($particleWeights[m]$, $particlePoses[m]$, $scan$, map) $particleWeights[m]$ /= sum($particleWeights$)
  $finalPose$ = [0,0,0]
  for m = 1 **to** $numParticles$:
    $finalPose$+= $particleWeights[m]$ $\times particlePoses[m]$
    =0

TABLE II: The time it takes to update the particle filter for 100, 300, 500 and 1000 particles. Estimate the maximum number of particles the filter can support running at 10Hz on the RPi.

Particle Number	Time(s)
100	0.0127
300	0.0249
500	0.0392
1000	0.0635

The performance of the particle filter on the log file `drive_square_10mx10m_5cm.log` is depicted in Fig. 4.

C. Combined Implementation

The block diagram for our SLAM system is shown in Figure 1. When a new motion command is given, the robot pose is updated by localization on the current map through particle filter. Then map update could be done based on the current pose estimation.

We ran our SLAM algorithm with `obstacle_slam_10mx10mx_5cm.log` file, and got figures and statistics shown in Figure 5 and Table III. From the map and trajectory plot, it could be seen that the map built by SLAM has a rotational drift, and SLAM pose also deviate from true pose but maintains almost the same shape of trajectory. Rotational drift in map is possibly caused by fast turning at corners. For the error curve and statistics, we compute the distance between SLAM poses and true poses that are at closest time instances as SLAM pose error, but it's actually not precise since SLAM pose lags behind true pose during

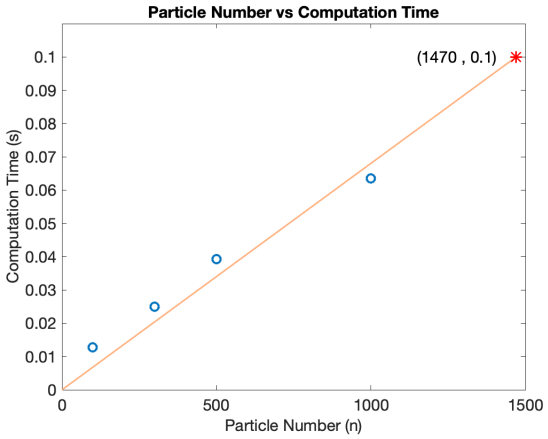

Fig. 3: Computation time increases as the particle number rises. If we want to run at 10Hz on the RPi, the maximum number of particles our filter can support is roughly 1470 according to the estimation.

turning. Thus some spikes appear in the error plot, and the maximum of SLAM error is quite large.

We also ran the drive square task for one circuit, and the plots for SLAM pose, odometry pose and true pose from log file are shown in Figure 6. Similar to SLAM pose error plot in Figure 5, we could see four spikes in the plot, which are probably caused by lag of SLAM pose when the bot turn at four corners. Unlike true poses gotten from log file, the distance between SLAM poses and odometry poses at the same instance could accurately reflect the difference between them. In lowest plot, the error between SLAM pose and odometry pose increases when sample index is larger , indicating that the difference between them grows with time. This observation also match SLAM and odometry pose trajectory shown in the highest plot. The SLAM pose error at the end of trajectory is 0.366 cm, showing that the SLAM estimation is quite accurate.

We made a few changes tweaks to the SLAM algorithm, mostly discussed in the sections on the particle filter. One particular change that we mentioned but did not discuss was that we chose to add the log-likelihood of a cell, rather than some constant value, to our odds. We did this in the hopes that it would punish sensor readings that suggested areas that were clearly walls were not walls, and that it would make log-odds only slightly above zero not have a significant impact on sensor likelihoods.

III. Planning and Exploration

Using the SLAM algorithm discussed above, we are able to build a map of environment as the MBot moving around. We then need to implement some strategies to

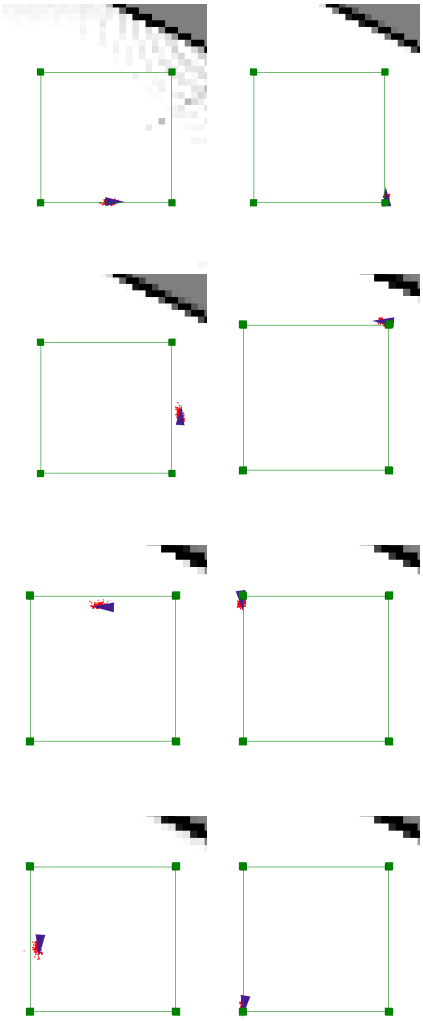

Fig. 4: The particle filter localization algorithm's particles compared to the robot's true pose as it makes its way in a square pattern across the floor.

TABLE III: obstacle_slam_10mx10mx_5cm.log statistics (mean, standard deviation, and max) of the error (as measured by Euclidean distance in meters) between the pose as measured by SLAM and the true pose. These distances were found by comparing each SLAM pose published with the true pose with the closest timestamp. Unfortunately, this matching method is not perfect; SLAM updates often lag behind the true pose updates.

Mean	Std. Dev.	Max
0.0444	0.0524	0.1601

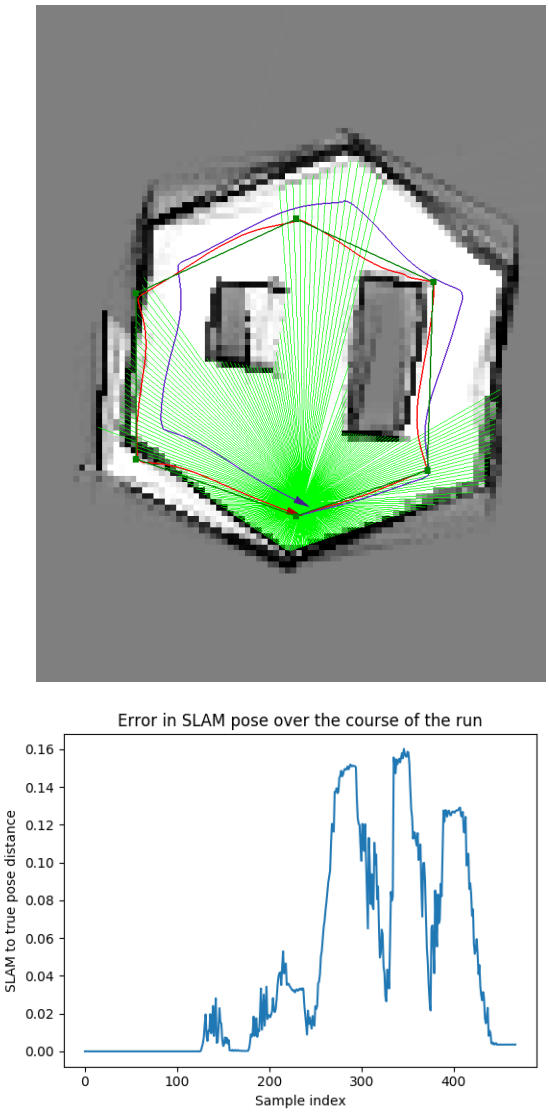

Fig. 5: Comparison of true pose and SLAM pose using `obstacle_slam_10mx10mx_5cm.log` file. In the upper plot, green line is the ground truth, red line is true pose, and the blue line is SLAM pose. In the error plot, we give index to SLAM pose samples in chronological order, and the X-axis is the sample index. For each sample, we find the true pose that has the closest timestamp and compute the euclidean distance between them as SLAM pose error.

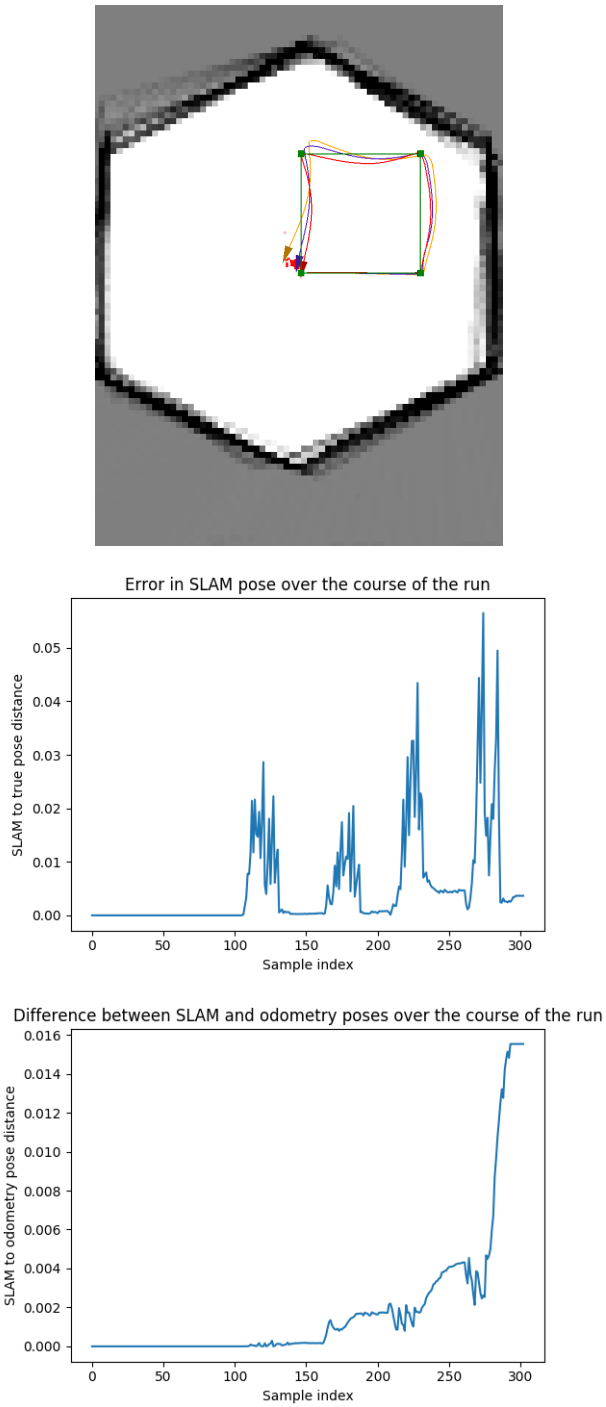

Fig. 6: Comparison of odometry pose, true pose, and SLAM pose for one square using a log file. In the map, orange is odometry, blue is SLAM, and red is true. Note that SLAM and odometry remain very close for some time, but eventually are very far from each other, indicating that SLAM is necessary for long-running robot navigation. Also, note the sudden spikes in the SLAM pose error compared to the true pose. These errors are likely caused by the fact that SLAM often lagged during turns. In this run, the distance between the SLAM pose and the true pose at the end of the run was 0.366 cm.

allow MBot to reach a target position in the maze without hitting obstacles.

A. Path Planning

Since we hope to find the shortest path in a maze and we know the start and end poses, the most commonly used strategy is A* graph searching algorithm. The cost function used in A* is shown in (3):

$$f(n) = g(n) + h(n) \tag{3}$$

A* considers the cost from the start pose on the graph as *g(n)*, and *h(n)* the *heuristic cost* that determines the closeness of the current pose to the end pose. That is to say, *g(n)* favors points closer to the start point and *h(n)* favors points closer to the end point. Therefore, compared to Dijkstra's algorithm, A* won't search over the entire graph, but prefers searching the direction towards the end pose, which saves lots of computation time.

Heuristic cost *h(n)* contains two parts. The first part is the euclidean distance towards the end pose, so the pose closer to the end pose is preferred to search first. The other part is the distance to the obstacle, which is shown as follows.

$$\begin{aligned} &ObstacleDistanceCost = 10\times \\ &|maxDistanceWithCost - ObstacleDistance| \end{aligned} \tag{4}$$

In (4), the obstacle distance cost reach minimum when *ObstacleDistance* equals *maxDistanceWithCost*, so the planned path prefer to stay at this distance from obstacles unless it must travel closer or farther to actually find a path.

Our implementation of A* path planner is described in Algorithm 5. The definitions of structure *Node* and *map_point* are given in Table VI. *closed_list* is a 2D vector that is the same size as map, storing information about whether cells are explored or not as well as the parent of them. With *closed_list*, we can not only check if a cell is explored efficiently, but also retrieve the planned path conveniently. *open_list* is a priority queue in which the node with small f pops out first.

To evaluate the performance of our A* algorithm, we use it to plan a path in a given map and command the bot to move through the path, and the result is shown in Figure 7. From the plots, it could be seen that our A* algorithm produce a path that has sufficient margin to forbidden red areas, thus it guarantee the safety of path. Besides, the path could also circumvent the obstacle area in advance, showing that the algorithm outputs path with optimal cost.

Table IV and V shows the time statistics for all successful and failed planning attempts in the given astar tests. It's shown that our A* algorithm is very efficient, and it only takes about 0.05s in the most complex case (The Median statistics seem to be not correct sometimes, we just put it here for reference).

Usually, our algorithm pass 5 of 6 tests, with some failures in test_maze_grid (Sometimes there could also be an "Incorrectly found valid path" failure in test_convex_grid). The failures appear probably because obstacle distance computation not accurate enough. Thus when searching around a narrow passage, the algorithm may mistakenly find a path or no path through it.

TABLE IV: Time statistics for successful planning attempts, the unit is μs. We use number 1-6 to represent corresponding test in the astar_test.

Test	Min	Mean	Max	Median	Std dev
5	683	683	683	0	0
1	563	964.667	1224	1107	288.01
6	468	468	468	0	0
3	27866	33850.5	39835	0	5984.5
4	31206	40536.3	50107	50107	7718.17

TABLE V: Time statistics for failed planning attempts, the unit is μs. We use number 1-6 to represent corresponding test in the astar_test.

Test	Min	Mean	Max	Median	Std dev
5	13	19.6667	30	13	7.4087
1	18	20	22	0	2
2	21	26.6	37	25	5.4626
6	28	31	36	36	3.55903
3	37	368.5	1012	1012	455.436
4	27	27	27	0	0

TABLE VI: Node and map_point structure

Struct **Node**		
data type	member	definition
Point<int>	*parent*	Parent Cell position
Point<int>	*self*	Current cell position
double	*g*	Path cost from start
double	*f*	Total cost value
Struct **map_point**		
Point<int>	*parent*	Parent cell position
int	explored	Exploration status

B. Exploration

Once we are able to sense the environment and has a path planning algorithm that can go between arbitrary two points, we are able to explore the whole map.

We firstly record those grid cells between free spaces and unknown spaces as frontiers. Then We select a cell

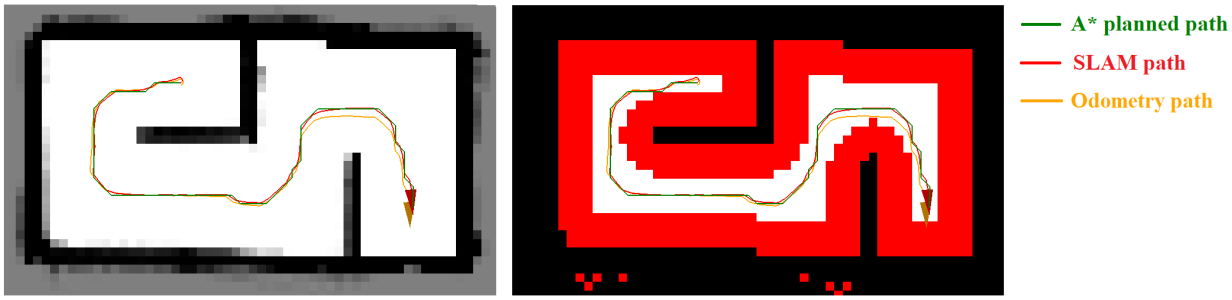

Fig. 7: Comparison of A* planning path, SLAM path and odometry path. The left plot shows paths on the given map, where free space is colored as white. The right plot also marks as red the spaces that the bot is not allow to enter because they are too close to obstacles. In the plots, the side of paths with arrows are the goal position, and the other sides are the start position.

Algorithm 5 A* Searching Algorithm

search_for_path(*start_pose*, *goal_pose*)
1. *start* = global_to_cell_position(*start_pose*)
2. *goal* = global_to_cell_position(*goal_pose*)
3. **if** *start* **or** *goal* not in map **or** too close to obstacles
4. **return** empty path
5. Initialize *closed_list* as a 2D vector of map_point
6. Initialize *open_list* as a priority queue of Node
7. *open_list*.push(*start_node)*
8. **while** ! *open_list*.empty()
9. *current_node* = *open_list*.top()
10. *open_list*.pop()
11. **if** *current_node* has been explored **continue**
12. Add *current_node.parent* to *closed_list*
13. Mark corresponding map_point as explored
14. **if** *current_node* is *goal*
15. Trace back to *start* through *closed_list*
16. **for all** intermediate map_point
17. Compute global position
18. Compute orientation wrt parent
19. Add state to path
20. **return** path
21. **for** *child* in neighbour of *current_node*
22. **if** *child* is not in map **continue**
23. **if** *child* too close to obstacles **continue**
24. Compute g and f cost of *child*
25. *open_list.push(child)*
26. **return** empty path

in free space and is near to a frontier point as the goal and use A* algorithm to plan a path to it and explore the environment meanwhile. When the bot almost reach the current goal, we re-plan a path in the same way until there is no frontier in the map. After building up the whole map, MBot will return home by setting current pose as the start and the home pose we stored initially as the goal of the A* searching algorithm. Thus, MBot is able to explore the map successfully. The algorithm we implemented is illustrated in Algorithm 6.

Algorithm 6 Exploration Algorithm

1. **State:** Initialize
2. **State:** Exploring Map
3. **if** *frontier* is empty
4. **return State** = Return Home
5. **while** not whole map is explored
6. Increment search radius by cell distance
7. **for all** *cell* in *frontier*
8. **for all** *neighbour* within radius
9. **if** *neighbour* is in free space
10. Try planning a path to it
11. **if** succeed **return** path
12. **return State** = Return Home
13. **State:** Return Home
14. **while** not whole map is explored
15. Increment search radius by cell distance
16. **for all** *neighbour* of *home* within radius
17. **if** *neighbour* is in free space
18. Try planning a path to it
19. **if** succeed **return** path
20. **return State** = Complete Exploration
21. **State:** Complete Exploration

Some detailed explanations of our algorithm are provided in the following. Firstly, before planning a path to a frontier, we need to ensure that the goal position is valid and lies in the free space, and we also want it to be as close to the frontiers as possible. Thus our method is to search through cells within a radius from all frontier points, if a cell in free space is found, we try planning a path to it with A*, and execute a path if the planning succeeds. Otherwise, the search radius is increased and we perform the steps again until a path is gotten. When the exploration finishes and we need

to plan a path back home, we also perform free space searching around home position (including itself) for valid goal in case home position somehow becomes not reachable.

Another thing is that we re-plan the path every time we are nearly finishing the current path. Otherwise, the MBot re-plan every time it moves, which waste a lot of time and is not necessary at all. Therefore, we set it to renew the path when it finishes the previous one. The reason we re-plan before it reaches the goal is that since the frontier is far away from current pose, the rplidar measurement at the goal position is likely to be not accurate when the path is planned. Thus a goal that is valid at the instance of planning may be found to be inside obstacle area when the bot move closer and get more precise measurement there. To avoid reaching an invalid goal, we choose to re-plan a new path when the bot has moved to a place within a given distance from the current goal.

C. The Kidnapped Robot Problem

While we did not implement a kidnapped-robot solver for the competition, from a localization standpoint the problem is fairly simple to solve. If instead of initializing each particle to the zero pose at the beginning of the problem, the distribution of particles is made up of randomly selected particles that are in non-occupied portions of the map, then the robot should have a good understanding of where it is likely to be. After that, the robot simply needs to take in measurements that should start to eliminate the possibilities of being in many states. The distribution should converge to be multimodal: that is, there may be multiple states that the robot is likely to be in that would all plausibly produce the sensor data given. At this point, the robot can take actions based on which direction of travel is least likely to crash into anything (hopefully, the robot can find a path that will not cause it to run into any obstacles no matter which state it is in) and take that route. By continuing to explore in this way, the particle filter will slowly eliminate particles that don't match the evidence received until the robot has found its true pose.

IV. Discussion

Individual components of the project such as SLAM and planning showed very good performance. When SLAM was run on the RaspberryPi we faced a trade-off between performance and computation-time. Planning and exploration tasks were pretty efficient computationally.

For the competition tasks, our motion controller encountered was found to be not tuned to perfection and hence the robot stopped short of the final line in the drive-square task. This was also because of the high tolerance limit that we had set. The SLAM map generated for this task was perfect though. Task 3 in the competition involved exploring a maze by seeking new frontiers. Our robot showed good performance in this task by planning a safe path and never hit any walls. Again the map produced was almost perfect. The maps were uploaded to the course's google drive folder.