ROB 550 Botlab Report

Brian Fogelson, Manas Buragohain, Li Chen, Nikhil Divekar
{bfogels, manasjb, ilnehc, ndivekar}@umich.edu

***Abstract*—A two-wheeled robot was used to demonstrate understanding of several topics in robotics leading up to SLAM. In this lab, concepts such as occupancy grid mapping, odometry action model, likelihood sensor model, and particle filter are implemented to form a complete simultaneous localization and mapping (SLAM) system. Obstacle distance, A* search algorithm, and frontier-based exploration algorithms were developed to explore a maze while simultaneously creating a map of the maze with SLAM.**

***Keywords:* SLAM**

I. Introduction

Mapping and localization are important abilities for an exploration purposed mobile robot. Simultaneous localization and mapping (SLAM) is still a question of ongoing research. SLAM based exploration was the main objective of the Botlab project. For the Botlab, all hardware and core software were provided. The robot provided had a differential drive (two wheels) with stops at the front and back for maintaining balance. Further, it was equipped with quadrature wheel encoders for determining wheel position, and 2D - LiDAR for range estimation. The lower level control of the wheels was performed on a Beagle Bone, while the higher level motion control was performed on a Raspberry Pi. The processing of the exploration algorithms was carried out on a laptop which communicated with the Raspberry Pi over the local wireless network via LCM messages, which use the UDP protocol.

There were several tasks that were completed to achieve a fully capable explorer. Firstly, the motion controller algorithm that guides that robot from a home position to a goal position was tuned to achieve desired kinematics. Next, the SLAM algorithm was implemented by means of a particle filter, and tuned to achieve the best results possible in both mapping and localization. Success could be measured by both accuracy of the map as well as accuracy in robot pose. This included varying the parameters of the action model, which accounts for errors in the robot's movement or odometry, and the sensor model, which accounts for errors in the LiDAR. Subsequently, the A-star algorithm was implemented to plan a path by creating a series of poses between the given home and goal poses in the most effective manner by appropriately weighing the cost of travel versus the cost of getting too close to an obstacle. Lastly, an exploration algorithm was developed that allowed the robot to drive around the maze (using A-star) with the goal of detecting, and then uncovering all frontiers (unexplored regions in a map), and then returning home.

The subsequent sections are organized as follows. In section II, a detailed description of the algorithms and parameters is provided for the motion controller, SLAM, A-star and exploration. Next, section III contains performance metrics from the tests carried out on the various algorithms developed. Lastly in section IV, the results achieved are assessed with respect to the project objectives, an explanation of some limitations and shortcomings is provided, and some areas of future work are considered.

II. Methodology

A. Motion Control

The bot was controlled using a Rotation-Translation (RT) method, differing from the usual RTR since a heading at the goal position was not required. Initially, the code was written in python (`mobilebot-f19/python/motion_controller.py`). Unfortunately, several issues were encountered during this process, such as handling several LCM subscribers and publishers. The final program was implimented in C++ by modifying `botlab-f19/src/planning/motion_controller.cpp`. In order to achieve smooth, yet quick motion, parameters needed to be tuned. Firstly, the part of the code that used the kPGain, kIGain, and kDGain was commented out. These values were originally used in the TURN state in order to not overshoot near the end of the turn; however, even after tuning the gains, the robot moved more smoothly without this section of the code. The kPTurnGain was used in the DRIVE state to maintain heading while driving. Experimentally, a value of 6.0 resulted in straight line driving without affecting other parts of the straight line motion.

kDesiredSpeed of 0.2 m/s and kMinSpeed of 0.1 m/s resulted in an acceptable trade-off between speed and consistency with the robot still being able to achieve desired locations. kTurnSpeed was set to 2.0 rads/s for turning during the TURN state, while kTurnMaxSpeed, which was only used during the DRIVE state, was set to

0.5 rads/s to maintain heading. A slowdown distance of 0.4 m was finalized after experimentation. An additional modification enabled the robot to turn in place at kTurnSpeed / 2.25 when no command was given. This was a strategic addition that allowed the robot to free itself when it ended up in an invalid cell by adding just enough noise to the map. Lastly, the threshold to stop turning was set to 0.5 rads. While this is a large threshold, it prevented the bot from getting stuck while turning due to steady state error in the controller. Accuracy was still ensured by interpolating with many points along a path.

B. SLAM

1) Mapping: For mapping, the location of the robot is assumed to be known. Moreover, it is assumed that the position and the laser scan line up at the same time, which is achieved using the `MovingLaserScan` function. The general idea behind mapping is that for each of the laser scans, the cells that the laser travels through are determined and weighed. Since the laser goes through these cells, these cells likely do not have obstacles in them, so a negative constant is added to the previous log odds in the cell. To figure out which cells the laser travels through, Breshenham's algorithm is used. The cell in which the laser terminates is assumed to be an obstacle, unless the range is the maximum distance. A positive constant is added to the previous log odds at this location. The odds can range from possible values of -127 to 127. The values of the constants to be added were determined by experimentation, resulting in values of Kp = 10 for the positive constant and Kn = -3.5 for the negative constant. The reason the positive constant is so much higher than the negative constant is because there are fewer final points of the laser than intermediate points, and sometimes negative values would not be high enough and walls would be grey or disappear. See algorithm 3 in the Appendices for the mapping pseudocode.

2) Action Model: The pose of a mobile robot is given by $(x\ y\ \theta)^T$, where x and y are the two-dimensional coordinates and θ is heading angle. [1] The probabilistic kinematic model, or action model plays a role in the state transition model of mobile robots. The model can be shown as a conditional density $p(x_t|u_t, x_{t-1})$, where x_t and x_{t-1} are both robot poses at consecutive times and u_t is the command. The odometry motion model, which was utilized for this robot uses odometry measurements by integrating wheel encoder information. During the time interval $(t-1, t]$, the robot moves from pose x_{t-1} to pose x_t. The odometry data is denoted $\bar{x}_{t-1} = (\bar{x}\ \bar{y}\ \bar{\theta})^T$ and $\bar{x}_t = (\bar{x}^{'}\ \bar{y}^{'}\ \bar{\theta}^{'})^T$. The motion information u_t is thus given by the pair $u_t = (\bar{x}_{t-1}\ \bar{x}_t)^T$. In order to realize this motion, u_t is transformed into a sequence of three steps: a rotation, a straight line transition, and another rotation (RTR). This transformation is denoted as $(\delta_{rot1}\ \delta_{trans}\ \delta_{rot2})^T$. The probabilistic model assumes that these three parameters are corrupted by rotational noise α_1, transitional noise α_3, their correlated noise α_2 and α_4. Thus, these variance are applied to generate three independent normal distribution noises, and then added into the motion estimation. Please see the Appendices for the equations of the odometry action model (equations 11-19) from [1], and the pseudocode of the algorithm that was actually implemented in this lab (algorithm 4).

With this method, in practice, often the particles with different poses would converge quickly to one pose and lose almost all variance. Therefore, if any error did occur due to slip or another variable, the robot often could not "catch up" to the true pose. After tuning values without success to account for this problem, a simplified action model was adapted that maintains constant variance. In other words, the particles cannot converge past a certain threshold.

This action model simply measures $\Delta_x, \Delta_y, \Delta_\theta$. These delta values are applied to each particle with some noise offset to maintain randomness. This spread needs to be maintained so that when an error occurs that causes a difference between the slam pose and the true pose, the slam pose is able to "catch up" to the true pose by weighting particles more heavily close to the true pose. Equations for this action model are shown below and the variance values are shown in Table I.

$$\delta_x = \bar{x}^{'} - \bar{x} \tag{1}$$

$$\delta_y = \bar{y}^{'} - \bar{y} \tag{2}$$

$$\delta_\theta = \bar{\theta}^{'} - \bar{\theta} \tag{3}$$

$$x^{'} = x + \delta_x + \mathcal{N}(0, \sigma_1) \tag{4}$$

$$y^{'} = y + \delta_y + \mathcal{N}(0, \sigma_2) \tag{5}$$

$$\theta^{'} = \theta + \delta_\theta + \mathcal{N}(0, \sigma_3) \tag{6}$$

TABLE I: Variance value in action model

Variance	Value
σ_1	0.01
σ_2	0.01
σ_3	0.01

These variance values were determined experimentally. Increasing these variance values increases the spread of particles. With a sigma of .05, the variance was large enough to cover an area of approximately 10 times the size of the robot. Large variance values worked surprisingly well, meaning the sensor model weighted more likely particles substantially higher than unlikely particles. Reducing these sigma values to .01 resulted in a particle spread of about 2 times the area of the robot. This seemed a reasonable compromise and allowed the robot to “catch up” to the true value when slip occurred without being too large of a spread that no particles represent the true pose well enough.

Combining this model with a relatively high number of particles, a combination of high accuracy and robustness was finally achieved, with the robot able to accurately track it’s true pose while adjusting when an error occurred. Since the sensor model used was a less computationally intensive version, 700 particles could be used without paying a penalty from lag. For more information about the number of particles, see Table V.

3) Sensor Model: The sensor model is used to determine the likelihood of a pose given a map and a scan. An extremely simplified sensor model was used since it could successfully track the true pose in localization while being less computationally expensive than other methods tested. A given scan object contains several hundred range and angle pairs. For a single range and angle pair, if the end of the ray is in an obstacle, then this is likely a correct position, and the log odds of the obstacle cell, which are positive, should be added to the total odds for the position. These odds are only added when the value in the cell is greater than 70, meaning there is relatively large confidence that the cell at the end of the laser is an obstacle. Sometimes, due to noise the end of the laser does not fall perfectly into the cell of the corresponding obstacle. In lecture, a suggestion was made to use Breshenham’s algorithm to check the cells just past and just before the end of the range to account for some noise, then add the log odds in the neighboring cells times a scalar less than 1 to reduce the importance of neighbors compared to the true terminating cell.

To save on computation by avoiding Breshenham’s algorithm, as well as to account for additional noise, all 8 neighboring cells were checked. When testing localization only with the obstacle slam log player, this method outperformed the method with Breshenham’s algorithm, tracking the true pose more closely. Additionally, to save on computation, a stride value of 6 was used. This means that only one in every 6 scans are checked and added to the total odds. This value was chosen experimentally, with a value of 6 not significantly affecting localization while saving computation time, compared to values less than 6 that took longer and values more than 6 that degraded the ability to accurately track the true pose. See algorithm 1 for the pseudocode of the actual implementation in the lab.

Algorithm 1: Sensor Model - Likelihood

```
Find likelihood of particle given laser scan;
odds ← $0$ ;
stride ← $6$ ;
i ← $0$
while $i <$ number of lasers do
    p ← laser end point ;
    m ← log odds at p;
    Check if obstacle at p;
    if $m > 70$ then
        odds = odds + m;
    end
    Check the $8$ cells neighbouring p;
    for $j$← $1$ to $8$ do
        n ← log odds at neighbour j of p;
        if $n > 70$ then
            odds = odds + $n * 0.5 * .125$;
        end
    end
    i ← $i + stride$ ;
end
```

4) Particle Filter: A particle filter is used to determine a discrete probability distribution of the pose of the robot. Many possible poses of the robot are given as particles, with weights corresponding to how likely each particle is to be the true pose. Initially, all particles are initialized to a pose with uniform weight.

If the robot has moved, the particles and weights of the robot are updated. This is done by first resampling the previous distribution with new particles so that the new distribution has more particles near the particles that were the most likely from the previous distribution. Low variance resampling is used to avoid the problem of particle depletion, or all particles converging to the same high weight particle. After resampling, the action model is applied to the new set of particles to move them in the same direction that the robot moved, but with some noise in order to get a spread of particles. Then, the normalized posterior distribution is computed from these newly moved particles. This applies the sensor model to each potential pose of the robot to determine their new weights, or the likelihood that the particle truly is the correct position given the map and scan. Finally, the pose is estimated from the new weighted distribution by taking the weighted average of the x and y positions, and getting the angle from the weights using the arc-tangent of the weighted sin sum and cos sum. In the future, this estimate could be improved by clustering the particles, and taking the weighted average from only the highest weighted cluster. This could avoid the problem of the robot position being estimated to be between clusters, when truly there are no points there.

5) Full SLAM system: In this project, the full SLAM system is implemented as Fig 1. In each iteration, odometry information is used to calculate if the robot moved. Then the low-variance resampling method outputs a filtered particle distribution. The action model and sensor model are applied to each particle to get posterior particle position and likelihood. Finally, the normalized particles distribution can generate estimated pose through simple weighted average. This means localization updates the pose, which is then used to update the map, which is a full flow of SLAM of the system.

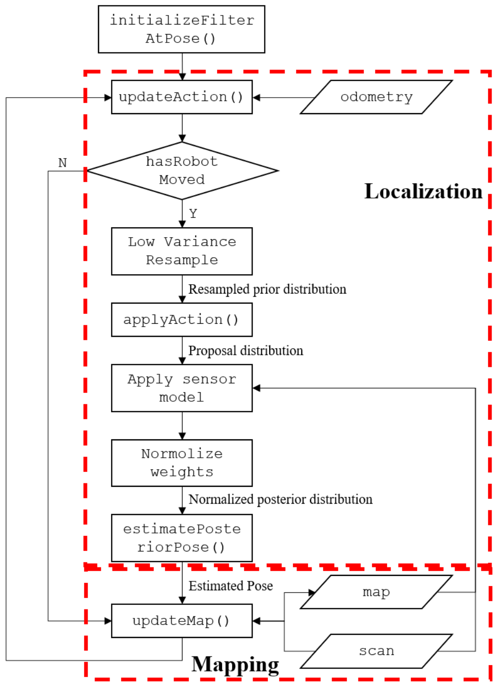


Fig. 1: SLAM system block diagram

C. Planning and Exploration

1) Obstacle distance: In addition to an occupancy grid map, an obstacle distance grid map is generated corresponding to each occupancy map to aid in path planning. To achieve this, a create a set of obstacle cells is created by comparing against cell log-odds from the occupancy map. Hereafter, each cell is assigned a obstacle distance defined as the Euclidean distance from it to the nearest obstacle cell. The pseudo-code is shown in Algorithm 5.

2) A planning:* An A* Search Algorithm with modified Heuristic cost is used for path planning, as shown in Algorithm 2. Each grid cell in A* algorithm is assigned three values, the cost of a path from the start position, $Gcost$, the heuristic cost, $Hcost$, which measures the cost to the goal cell in the form of Euclidean distance, and the total cost, $Fcost$, which is the total cost of the cell. In addition, a obstacle distance cost, $Ocost$, calculated using the obstacle distance grid, is incorporated to deter the A* algorithm from planning paths close to the obstacles. The overall $Fcost$ is calculated as in equation 7. Specifically, $Gcost$ here is calculated through 8-way diagonal distance but $Hcost$ is calculated through actual euclidean distance as in equation 8, 9.

$$F(n) = G(n) + H(n) + O(n) \tag{7}$$

$$G(n) = \begin{cases} G(n.parent) + 1.4 & \text{If diagonal} \\ G(n.parent) + 1 & \text{If adjacent} \end{cases} \tag{8}$$

$$H(n) = \sqrt{(goal.x - n.x)^2 + (goal.y - n.y)^2} \tag{9}$$

$$O(n) = (maxDistWithCost - dist\_(n))^p \tag{10}$$

Algorithm 2: A* planning

```
    Input: $start, goal, h(n), o(n), expand(n)$
    Output: $path$
  Check if $start$ and $goal$ are valid ;
  if $start = goal$ then
      return $makePath(start)$ ;
  end
  $closedList \leftarrow \emptyset$ ;
  $openList \leftarrow start$ ;
  while $openList \neq \emptyset$ do
      $sort(openList)$ ;
      $n \leftarrow openList.pop()$ ;
      $kids \leftarrow expand(n)$ ;
      foreach $kid \in kids$ do
          Check if $kid$ is valid ;
          $kid.f \leftarrow (n.g + 1) + h(kid) + o(kid)$ ;
          if $kid = goal$ then
              return $makePath(kid)$ ;
          end
          if $kid \cap closedList$ then
              $openList \leftarrow kid$ ;
          end
      end
      $closedList \leftarrow n$ ;
  end
  return $\emptyset$
```

During each search iteration, the minimum $Fcost$ cell is considered and expanded. If multiple cells have the same $Fcost$, then the minimum $Hcost$ cell is be considered. Through this heuristic search, the path is finally arrive at the goal and then a trace back algorithm can generate the final path.

3) Exploration: Once a method was in place to plan a path from a start point to a destination point using A-star, the exploration code could begin to be written. The map has certain frontiers, or unexplored areas. A cell is classified as part of a frontier if it contains a log odds value near zero, and the cell is next to a cell that is open (positive value). Cell values are initialized to 0, and a cell value near 0 means there is uncertainty as to whether to cell contains an obstacle or free space.

Once all the frontiers are found, the function `plan_path_to_frontiers` is used to plan a path from the robots current position to a future unknown position. The frontier chosen out of all the frontiers is the frontier with the single closest point. Then from this frontier, the middle point of the frontier is approximated by taking the point at the half point of the array of cells in a frontier.

At this point there is a desired start point and end point; however, the end point is not a reachable point since it has a value of approximately zero, and as a result is not considered an open cell. As a result, a breadth first search (BFS) is used to find the nearest open cell to the closest frontier's midpoint. This BFS was accomplished using a square radius, and checking the points in the 4 lengths along the square at a given radius, then increasing the radius if no free points are found.

Now there is a method of making a path from the current spot to the closest free spot to an unexplored frontier. By following all the paths to all the frontiers consecutively, the entire map can be explored.

We utilize a sequential state machine to autonomously explore an unknown environment. Since more information about the map is gained while the robot is explores, a check is put in place to determine if there is a need to update the path that the robot is on. Whether or not that path is safe is checked by the `isPathSafe` function by checking if all points along the path are valid goals,

If the path is no longer determined to be safe and the robot is currently in a safe starting position, then the a boolean is updated that tells the state machine to get a new path to the closest frontier. Likewise, if the robot achieves a position near the end of the path or the current path is empty and there are more frontiers left, the boolean `need_to_update_path` is set to true.

Once there are no more frontiers, the map is considered explored. During more than two thirds of the trial runs, the robot managed to explore the whole map and did not leave out any frontiers.At this point, the robot enters the return home state. The home state was initialized when the `exploration` program was first called. The robot then repeatedly calls the the `motion_planner` to plan a path from the current position to the home point until a valid path is made. Once the path is made, it can be assumed that the path is going to be safe because the whole map is known. Once the robot makes it close enough to the home point then exploration is complete.

III. Results

A. Motion Control

As discussed in Section II-A, final motion controller parameters are shown in Table II. These values were obtained through experimentation as explained in the section above.

B. SLAM

1) Localization Only: As is evident from the minimal error in the graphs below, the robot was successfully able to use localization to determine its location. Even as the odometry pose differed from the true pose in simulations, the SLAM pose was able to recover the true pose by combining the sensor model and the action model in the particle filter.

TABLE II: Motion controller parameters

Parameter	Value
kPTurnGain	6.0
kDesiredSpeed	0.2
kMinSpeed	0.1
kTurnSpeed	2.0
kTurnMaxSpeed	0.50
slowDownDistance	0.4

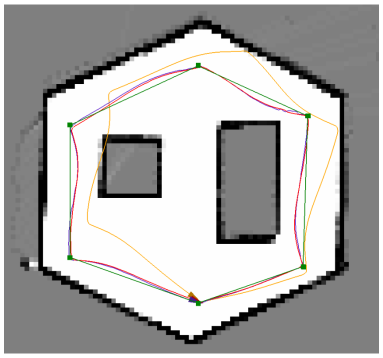


Fig. 2: SLAM on obstacle map from log file

2) Mapping Only: The robot was able to successfully create a map of its environment. Fig. 2 is a map created when our algorithm was run on the replayed log data.

TABLE III: The performance of SLAM implementation vs. ground-truth in `obstacle_slam_10mx10m_5cm.log` simulation

Pose error	Mean	Std dev	Max
x (m)	0.013565	0.048424	0.184257
y (m)	-0.003389	0.033226	0.158667
θ (rad)	0.754885	1.369215	3.135231

TABLE IV: Final x-y position of the robot to the starting point in `drive_square` test

Direction	Position (m)
x	0.0008244 ± 0.0002
y	0.02633 ± 0.0002

3) Full SLAM: In Full SLAM, the robot successfully made maps of the environment that it was in, both in simulation and on the physical robot. In Full SLAM, the map improves information about the localization, which improves information about the map, and the two keep augmenting each other if done successfully.

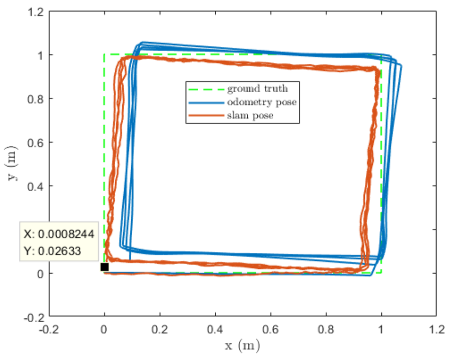


Fig. 3: SLAM pose estimate in a convex environment in addition to odometry and ground truth for `drive_square`

One example of this can be seen in drive square, where the robot improves its pose estimate with information from the map to drive in a square. The performance can be seen in Table IV and Fig 3.

The timing of full SLAM can be seen in Table V. As discussed in the methodology section, methods were implemented to make SLAM run relatively more quickly. As a result the average update times are lower than what was seen in several other groups who compared values. These average times were the average times when the robot had moved, which also increases the time relative to what it could be, since the process of updating the filter is nearly instantaneous when the robot has not moved. Gradually increasing the number of particles used, the first time the max of an update of the particle filter took more than .1 seconds occurred at 1200 particles. The number of particles could be increased to 2500 before updating the particle filter consistently took more than .1 seconds, or was slower than 10 Hz. In practice on the laptop, 700 particles were used since accurate maps could be achieved without pushing the refresh rate limits of the program on the laptop.

TABLE V: Update time for different particle numbers

Particle Numbers	Update Time (μs)
100	5111.14
300	14167.13
500	21462.81
1000	40102.99
1200	47109.10
2000	78804.26
2400	96168.364
2500	105599.95

During exploration, the map made by the robot looked nearly identical to an overhead image of the maze. Occasionally, lines around a wall would not line up to a perfect 180 degrees. This could be improved by improving localization. One way of doing this could be tweaking parameters in the action model and sensor model to improve localization performance, specifically with regards to the robot orientation.

4) Kidnapped Robot: The robot was successfully able to localize when the initial distribution of the particles as well as the initial odometry position, were randomized. In the video below, the convergence of the particles to the true position can be seen to be accomplished successfully. Link to Extra Credit Video: https://drive.google.com/file/d/1EmHvpy-f2k6Ay1b7UhldbKoxWOse2jqU/view?usp=sharing.

C. Planning and Exploration

The exploration task and `astar_test.cpp` are utilized to ascertain the performance of A* algorithm.

As shown in Fig 6, A* generates a path during exploration which is shown as green square dots. The yellow curve is the estimated slam pose. The `astar_test.cpp` result is shown in Table VI, successfully and failed planning respectively.

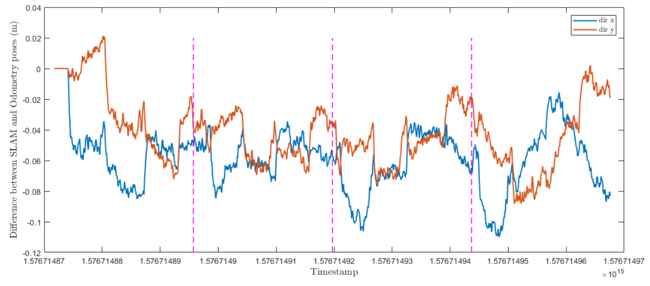
IV. Discussion

A. SLAM

SLAM required tuning of parameters as well as modification of algorithms to achieve desired results. Some of these modifications that could be tuned were found in the action model, sensor model, mapping, and particle filter programs.

1) Mapping: For mapping, a simplified model was used that assumed the laser must travel through free space and end in an obstacle or the max range. This simplified model does not account for random noise. Additionally, this model relies on the use of two constants, one for empty space and one for obstacles. The value of these constants relative to each other and relative to the maximum value of 127 affected both how quickly the map trusted its readings from scans, and affected whether or not free space or walls would potentially overwrite each other from noise. To make SLAM work effectively, both constants had to be increased from their initial values, with the object constant Kp being increased more to trust the sensor readings more quickly for SLAM, and to prevent walls from being overwritten by free space.

2) Localization: In the action model, there were trade-offs that resulted from the variance values as well as which algorithm was used. The simple odometry algorithm with constant added variance was used to directly control the amount of variance. Although this worked well enough to generate clean maps for the maze and allow the robot to complete exploration, it is possible


(a) Difference in x-y(m)

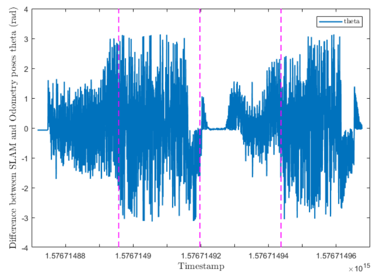

(b) Difference in θ(rad)

Fig. 4: Difference between SLAM and Odometry poses over time

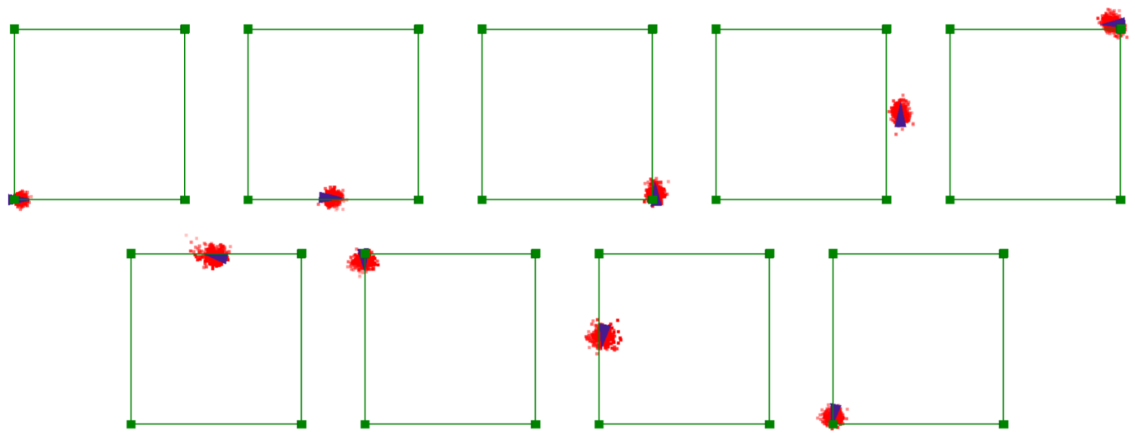


Fig. 5: 300 particles localization performance in `drive_square_10mx10m_5cm.log` simulation

(a) A* test results for successfully planning attempts (μs)

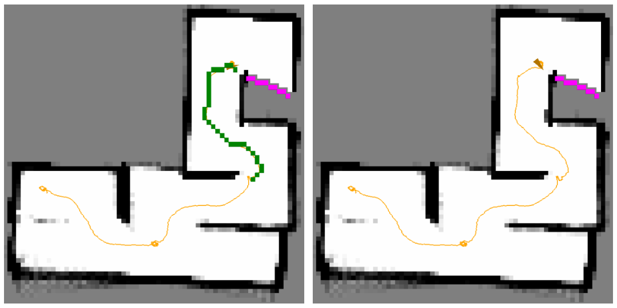
Test case	Min	Mean	Max	Median	Std dev
test_empty_grid	2044	2208	2521	2044	221.409
test_convex_grid	552	569.5	587	0	17.5
test_narrow_constriction_grid	1438	21860.5	42283	0	20422.5
test_wide_constriction_grid	1373	15610	43892	1565	19998.5
test_maze_grid	712	810.75	941	712	88.1543

(b) A* test results for failed planning attempts (μs)

Test case	Min	Mean	Max	Median	Std dev
test_empty_grid	11	21.5	32	0	10.5
test_filled_grid	10	10.4	11	10	0.489898
test_convex_grid	11	18	25	0	7
test_narrow_constriction_grid	10	1.2782e+06	3.83455e+06	24	1.80762e+06
test_wide_constriction_grid	10	10	10	0	0

TABLE VI: A* test results for `astar_test.cpp`

that the same results could have been achieved with the velocity transitional model with fewer particles. With more time in the lab, the four alpha parameters would have been tuned even more to further compare the results



(a) Planned path (b) Slam pose

Fig. 6: Planned path in an environment with the SLAM path driven by the robot overlayed on top

from these two different algorithms.

In the sensor model, there were trade-offs between speed and accuracy. Increasing stride and using all neighbors rather than only two neighbors from Breshenham's increased the speed of one update of the particle filter. This allowed for more particles to be used, which in turn increased accuracy. The downside of this speed increase was potentially losing accuracy by ignoring many scans and by accounting for many nearby neighbors. Accounting for all nearby neighbors does help account for some noise.

3) SLAM: After putting together localization and mapping with the changes above, the particle filter required the robot to estimate its pose from the set of weighted particles. Weighted averages were used across all the particles, with a slight tweak to angles discussed earlier. Another algorithm was explored simply choosing the single best particle, but this did not perform quite as well as the weighted average and the robot's pose often differed from the ground truth. If there were more time for this project, a method that uses clustering then selects the weighted average from the best cluster would have been implemented to avoid picking a point between clusters.

B. A* search algorithm

The greedy aspect of the simple implementation of A* Algorithm causes the generated paths to have a very small margin from the obstacles. This especially proved to be a problem while executing the exploration algorithm, as the robot would often end up in goal cells which were close to obstacles, resulting in an invalid start pose for the next path. We modify our implementation of the cost function to incorporate an obstacle distance cost with a very high $maxDistWithCost$ range. This results in a convex cost function, resulting in a path with each pose having the minimum cost from its nearest obstacles in real time.

When A* did not have a valid start point, as it was implemented in the code, the program could not plan a path and sometimes got stuck. For this reason, the robot was made to spin so that it would make it to a point just barely able to find a path with a valid origin. With more time, a breadth-first search would have been implemented from the start point as well to find the closest free point that the robot could navigate to.

C. Competition Performance

Since the arm was not integrated with the exploration state machine, the robot was able to successfully complete Task 1 and Task 3.

Task 1 required that the robot autonomously drive a square 4 times. During the competition, this task was accomplished with SLAM. As a result, Though the robot could not drive on the ground truth square lines, it could reach the final position accurately. Although SLAM was not perfect and the map still shifted a little, the localization was good enough to finish with a final position error of 3 cm in the y direction.

Task 3 of the competition was exploration. It required the robot autonomously explore a complex maze, building the overall map and return to the initial position. The performance here was better than expected, with the robot achieving the highest score at this task of any group in the section. It built a complete map though with only slight shifting and successfully returned home. With additional time, this task could also be improved to achieve a map with even cleaner lines.

V. Conclusion

In conclusion, a mobile robot was implemented for executing various tasks. The implementation required integration of several areas such as occupancy grid mapping, odometry action model, likelihood sensor model, particle filter, obstacle distance calculation, A* search algorithm, exploration algorithm. In the end, a full SLAM system was implemented on the mobile robot and was able to explore a maze and create a map from SLAM which outperformed the odometry estimation. The performance of the robot in the competition was better than expected, and many lessons were learned.

References

[1] W. Burgard and D. Fox, *Probabilistic robotics*. MIT Press, 2005.
[2] J. Borenstein and L. Feng, "Gyrodometry: a new method for combining data from gyros and odometry in mobile robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, April 1996, pp. 423–428 vol.1.
[3] "Rob 550 lab material."

VI. Appendices

A. SLAM

Algorithm 3: Mapping Algorithm - Bresenham

```
Identify cells that fall under line from robot center to
  laser range endpoint, increase weight of end cell,
  reduce weight of all other cells ;
$x_0, y_0 \leftarrow$ x,y coordinate of robot ;
$x_1, y_1 \leftarrow$ x,y coordinate of laser range ;
$dx, dy \leftarrow$ abs($x_1 - x_0$),abs($y_1 - y_0$) ;
$err \leftarrow dx - dy$ ;
$sx, sy \leftarrow$ sign($x_1 - x_0$),sign($y_1 - y_0$) ;
$x, y \leftarrow x_0, y_0$;
$k_n, k_p \leftarrow -3.5, 10$;
while $x \neq x_1$ OR $y \neq y_1$ do
    $logodds(x, y) = logodds(x, y) + k_n$ ;
    $e2 \leftarrow 2 * err$ ;
    if $e2 \geq -dy$ then
        $err \leftarrow err - dy$ ;
        $x \leftarrow x + sx$ ;
    end
    if $e2 \leq dx$ then
        $err \leftarrow err + dx$ ;
        $y \leftarrow y + sy$ ;
    end
end
$logodds(x, y) = logodds(x, y) + k_p$ ;
```

1) Action model:

$$\delta_{rot1} = atan2(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta} \tag{11}$$

$$\delta_{trans} = \sqrt{(\bar{y}' - \bar{y})^2 + (\bar{x}' - \bar{x})^2} \tag{12}$$

$$\delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1} \tag{13}$$

$$\hat{\delta}_{rot1} = \delta_{rot1} - \mathcal{N}(0, \alpha_1 \delta_{rot1}^2 + \alpha_2 \delta_{trans}^2) \tag{14}$$

$$\hat{\delta}_{trans} = \delta_{trans} - \mathcal{N}(0, \alpha_3 \delta_{trans}^2 + \alpha_4 \delta_{rot1}^2 + \alpha_4 \delta_{rot2}^2) \tag{15}$$

$$\hat{\delta}_{rot2} = \delta_{rot2} - \mathcal{N}(0, \alpha_1 \delta_{rot2}^2 + \alpha_2 \delta_{trans}^2) \tag{16}$$

$$x' = x + \hat{\delta}_{trans} \cos(\theta + \hat{\delta}_{rot1}) \tag{17}$$

$$y' = y + \hat{\delta}_{trans} \sin(\theta + \hat{\delta}_{rot1}) \tag{18}$$

$$\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \tag{19}$$

Algorithm 4: Action Model

```
Move particles using odometry measurements plus
  added random noise;
$\alpha_1 \leftarrow 0.01$ ;
$\alpha_2 \leftarrow 0.01$ ;
$\alpha_3 \leftarrow 0.01$ ;
$\Delta_x = ODO_x^k - ODO_x^{k-1}$;
$\Delta_y = ODO_y^k - ODO_y^{k-1}$;
$\Delta_\theta = ODO_\theta^k - ODO_\theta^{k-1}$;
for $i \leftarrow 1$ to number of particles do
    $error_x \leftarrow \mathcal{N}(0, \alpha_1)$ ;
    $error_y \leftarrow \mathcal{N}(0, \alpha_2)$ ;
    $error_\theta \leftarrow \mathcal{N}(0, \alpha_3)$ ;
    $p_x^i \leftarrow \Delta_x + error_x$;
    $p_y^i \leftarrow \Delta_y + error_y$;
    $p_\theta^i \leftarrow \Delta_\theta + error_\theta$;
end
```

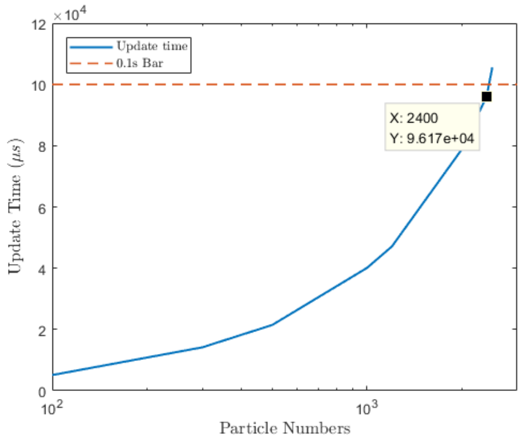


Fig. 7: Update time for different particle numbers

Algorithm 5: Obstacle distance

```
  Input: $grid, d(cell, obs)$
  Output: $distance\_$
$obstacles \leftarrow \emptyset$ ;
forall $cell \in grid$ do
    if logodds(cell) > 0 then
        $distance\_(cell) \leftarrow 0$ ;
        $obstacles \leftarrow cell$ ;
    end
end
forall $cell \in grid$ do
    if logodds(cell) > 0 then
        for $i \leftarrow 1$ to size of obstacles do
            $distance\_(cell) \leftarrow$
              $min\{d(cell, obstacles_i)\}$ ;
        end
    end
end
```