Team 5 ROB 550 Botlab Report

Thomas Augenstein, Oscar de Lima, Jonathan Michaux, Stanley Lewis
{tomaugen, oidelima, jmichaux, stanlew}@umich.edu

***Abstract*—Robotic mapping, localization, and navigation are challenging and important areas of robotics research. Here, we present a two-wheeled differential drive robot that is capable of mapping and navigating within an unknown maze-like environment. We have implemented a particle filter based SLAM (simultaneous localization and mapping) algorithm, detailing both its component parts and the experimental results on a series of tasks. For motion planning, we implemented an A* path planning algorithm and quantified its performance on various static maps provided by the instructor. Lastly, we implemented an RTR (rotation, translation, rotation) based motion controller. Our results demonstrate that SLAM is able to successfully map and navigate the environments proposed by the instructor.**

I. Introduction

Mapping, localization, and motion planning are fundamental areas of modern robotics research. Indeed, the ability to localize a mobile robot in an unknown environment is a prerequisite for truly autonomous behavior. Successfully solving these problems will have applications in underwater and terrestrial exploration, as well as mining, construction, and search and rescue.

Classical approaches for localization relied on obtaining estimates of the robot's pose from wheel odometry. Unfortunately, these estimates tend to drift very quickly, leading to increasing errors that make the estimates unusable after few meters [1]. An alternate approach is to use Simultaneous Localization and Mapping, or SLAM [2]. SLAM algorithms allow a mobile robot to build a map of an unknown environment, while simultaneously enabling the robot to determine its location within the map. SLAM has been successfully applied to self-driving cars, 3D navigation of aerial vehicles, 3D scene reconstruction, and augmented reality.

In this project we enabled autonomous locomotion of a differential drive robot by combining occupancy grid mapping, particle filter localization, and A* motion planning. This approach allowed the robot to successfully build a map of, and explore, unknown environments.

II. Methodology

A. Motion Control Algorithm

We developed a motion control algorithm to navigate between entries in a list of destination coordinates (expressed in a global frame of reference), called *waypoints*. Each waypoint is comprised of a position (x and y) and a heading (θ) with distances Δx, Δy, and $\Delta\theta$ to the next waypoint in the list. In order to successfully navigate between waypoints, our motion controller's goal was to drive each Δx, Δy, and $\Delta\theta$ to zero. Because driving each of these variables to zero sequentially (first Δx, then Δy, the $\Delta\theta$) is usually inefficient and addressing them concurrently (simultaneously reducing Δx and Δy) creates a complex, multi-dimensional control problem, we simplified our controller by substituting new control variables α, ρ, and β (Figure 1) and driving each to zero sequentially rather than concurrently. α denotes the angle between the robot's current heading and the line between the current and final position, ρ is the shortest euclidean distance between the current and final position, and β is the angle between the line between the current and final position and the heading of the next waypoint. This controller is called a "rotation-translation-rotation" controller, or RTR controller, and is convenient because it addresses each variable one by one while still driving the shortest possible distance.

The motion control algorithm used three proportional control equations to eliminate both rotation and translation error, shown below.

$$\begin{aligned} C_t &= K_t\rho \\ C_{r,\alpha} &= K_r\alpha \\ C_{r,\beta} &= K_r\beta \end{aligned} \tag{1}$$

TABLE I: Motor gain parameters

K_t	K_r	$C_{t,max}$	$C_{r,\alpha,max}$	$C_{r,\beta,max}$
1	1.5	0.1	0.1	0.1

In the above equations, C_t is the translation command, $C_{r,\alpha}$ and $C_{r,\beta}$ are the rotation commands from α and β, respectively, and K_t and K_r are the translation and rotation feedback gains, respectively. During the first rotation in the RTR algorithm to eliminate α, only the second control equation was active. During the translation step, the first control equation was active, though the second equation was used to correct for heading error during the translation. Finally, in the last rotation, only the third control equation was active. To prevent large control signals due to high initial errors, each control command had a saturation limit. The saturation limits and gains for

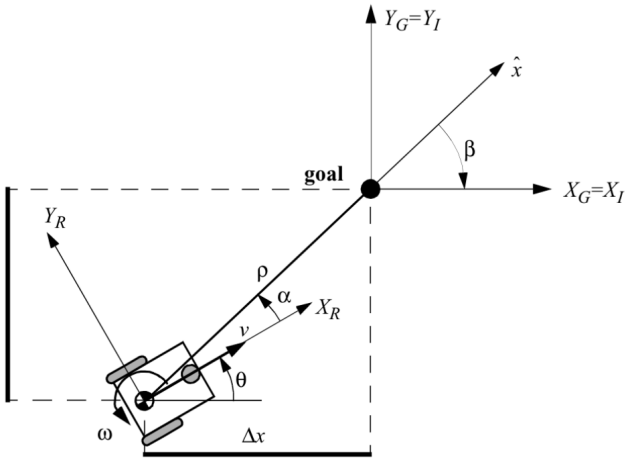

Fig. 1: Bird's eye view our mobile robot with labeled relevant dimensions. Δx and Δy are the distances to the next waypoint, and θ is the current heading of the robot. X_I and Y_I are unit vectors of the coordinate system fixed to the global frame, X_R and Y_R are unit vectors of a coordinate system fixed to the robot's frame of reference, and X_G and Y_G are unit vectors attached to global reference frame and aligned with the goal waypoint heading. α is the angle between the current heading and the line between the current and goal positions, ρ is the distance from the current to goal positions, and β is the angle between the line between the current and goal positions and the target heading. Our motion control algorithm drives α, ρ, and β to zero sequentially

each control equation are shown in Table II. Feedback from the robot's simultaneous localization and mapping was used to update each control variable.

B. SLAM

1) SLAM Overview and Block Diagram: In order to properly perform motion path planning and destination selection, a good estimate of the robot's environment and location is needed. This is accomplished via an iterative SLAM (simultaneous localization and mapping) algorithm. A block diagram of the slam system is shown in Figure 2.

2) SLAM Forward Sensor Model: The mapping portion of the SLAM algorithm requires a forward sensor model - that is, a model of $P(Grid \mid Lidar, Pose)$. Our implementation was quite naive and is described in Algorithm 1.

The function *Breshenhams* refers to Breshenham's line algorithm, whose implementation is detailed in [3].

The *logOddsHit* and *logOddsMiss* variables used in the forward sensor model are constants that are specified on program launch. They default to *logOddsHit = 3* and *logOddsMiss=1* respectively, and due to the acceptable performance of these defaults, no further modifications to their values was attempted.

Algorithm 1: Forward Sensor Model

```
Result: Populates the occupancy grid with
        results of each lidar scan
for scanBeam do
    scanDest.X = robotPose.X +
     scanBeam.length * cos(scanBeam.theta);
    scanDest.Y = robotPose.Y +
     scanBeam.length * sin(scanBeam.theta);
    Grid(scanDest.X, scanDest.Y) +=
     logOddsHit;
    Breshenhams(robotPose, scanDest, Grid,
     logOddsMiss);
end
```

3) SLAM Reverse Sensor Model: As the localization model is particle filter based, an inverse sensor model is needed to serve as a likelihood function. That is, we need some function $L(Lidar) \propto P(Lidar \mid Pose, Grid)$. This function was implemented as follows.

Essentially, this inverse sensor model casts each lidar from the robot into the map, and determines the distance between the casted ray and the length of the lidar return ping. The marginal likelihood of the beam is then the inverse of the distance delta, excepting when the distances are equal, then it is set to a constant of 3.0.

4) SLAM Action Model: Just as in the motion controller, the SLAM Action Model is based on an RTR (rotation, translation, rotation) scheme. The matrix form of the action model is

$$\begin{bmatrix} X_{t+1} \\ Y_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} X_t \\ Y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} (\Delta_{trans} + \epsilon_2) * \cos(\theta_t + \alpha + \epsilon_1) \\ (\Delta_{trans} + \epsilon_2) * \sin(\theta_t + \alpha + \epsilon_1) \\ \theta_t + \Delta_{theta} + \epsilon_1 + \epsilon_3 \end{bmatrix} \quad (2)$$

Where

$$\begin{aligned} \Delta_{trans} &= \sqrt{(X_t - X_{t-1})^2 + (Y_t - Y_{t-1})^2} \\ \Delta_{theta} &= \theta_{t-1} - \theta_t \\ \alpha &= atan2(Y_t - Y_{t-1}, X_t - X_{t-1}) - \theta_t \\ \epsilon_1 &\backsim \mathcal{N}(0.0, \delta_1 * \mid \alpha \mid) \\ \epsilon_2 &\backsim \mathcal{N}(0.0, \delta_2 * \Delta_{trans}) \\ \epsilon_3 &\backsim \mathcal{N}(0.0, \delta_3 * \mid \Delta_{theta} - \alpha \mid) \end{aligned} \quad (3)$$

δ_1, δ_2, δ_3 are user chosen parameters to describe the variance of the action model perturbations. Their final values are shown in table II.

These values were initially chosen effectively at random, and they achieved acceptable results, so no further tuning of them was performed.

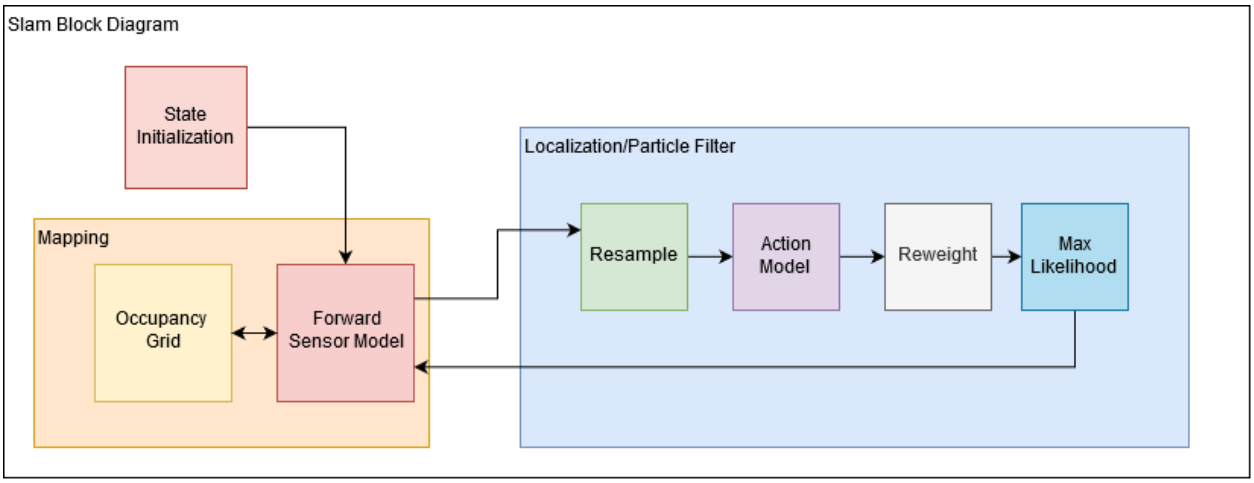

Fig. 2: A high level overview of the SLAM architecture

C. Path Planning

The A* algorithm was used to plan a trajectory between two different poses in the environment. The algorithm uses the following g and h costs:

$$g = \text{distance to starting cell}$$

$$h = \frac{1}{\text{distance to closest obstacle}}.$$

1) Obstacle Distance Grid: To get the value of the distance to the closest obstacle for every cell in the map the `obstacleDistanceGrid` object was created. The algorithm used to created the obstacle distance grid is the Brushfire algorithm [4]. The distance to the closest obstacle was then converted to meters instead of number of cells. The code used to build this object is implemented in `obstacle_distance_grid.cpp`. This implementation was able to pass the three tests posed in `obstacle_distance_grid_test.cpp`.

2) A star: The implementation of the A* path planning is done in `astar.cpp`. The `search_for_path` function takes a start pose, a goal pose, an obstacle distance grid and the search parameters as arguments. The only search parameter used is the minimum acceptable distance to an obstacle which is set to 0.2 meters. The value of 0.2 meters was chosen to give a margin of error over the radius of the robot which is approximately 1.54 meters. The algorithm then starts expanding the nodes of the tree ensuring that every node is actually inside the map and that every node is safe (the distance to the closest obstacle is greater than a threshold). Test results for the algorithm are shown in Table III.

3) Exploration: In order to exploring a map and return to the starting or home position the robot had to find the current frontiers. Frontiers were defined as groups of cells that have unknown occupancy and are adjacent to cells with known occupancy. The robot then had to plan a path to one of the frontiers with A* and repeat until there were no frontiers remaining. The robot would then plan a path home..

The `find_map_frontiers` function provided by the instructors returned the frontiers that were detected at the time. The goal was then to plan to one of these frontiers. The algorithm implemented in `frontiers.cpp` picked a random frontier and then picked the middle point in that frontier. Uniformly distributed random points were sampled inside an expanding box around this middle point until a valid A star path from the robot's current position to the sampled point was found. The robot would then move to its new target and the algorithm would repeat this process until no frontiers were found. An A* path would then be calculated from the robot's current position to the home position. Because of small errors in the mapping, the A* path planning algorithm would sometimes fail to find the home position. To solve this points would be uniformly sampled inside an expanding bounding box around the home pose. The real home pose would then be appended to the final path. The implementation of the exploration algorithm was done in `exploration.cpp`.

4) Localization in exploration: Localization was not modified in any way for the purposes of exploring the map. The implementation of localization for exploration is identical to the one described in the SLAM section of this report. Localization with incomplete information of the map is done the same way that localization with complete information.

Algorithm 2: Inverse Sensor Model

```
Result: Determines the likelihood of a sensor
        scan given a proposed location and map
Z = 0;
for scanBeam do
    scanThetaGbl = robotPose.Theta +
     scanBeam.Theta;
    castDist = 0.05;
    sensorMaxDist = 5.0;
    castCellX = robotPose.X +
     cos(scanThetaGbl)*castDist;
    castCellY = robotPose.Y +
     sin(scanThetaGbl)*castDist;
    while !Grid.IsOccupied(castCellX,castCellY)
     and castDist≤sensorMaxDist do
        castDist += 0.025;
        castCellX = robotPose.X +
         cos(scanThetaGbl)*castDist;
        castCellY = robotPose.Y +
         sin(scanThetaGbl)*castDist;
    end
    destCellX = robotPose.X +
     cos(scanThetaGbl)*scanBeam.length;
    destCellY = robotPose.Y +
     sin(scanThetaGbl)*scanBeam.length;
    distGridCoords = dist(castCellX, castCellY,
     destCellX, destCellY);
    if distGridCoords != 0 then
        Z += 1.0 / distGridCoords;
    else
        Z += 3.0;
    end
end
return Z;
```

TABLE II: Action model parameters.

δ_1	δ_2	δ_3
0.01	0.05	0.01

III. Results

A. SLAM

1) SLAM Overview: Our SLAM implementation had excellent final results. An example map and path is shown in figure 3. As can be seen there, the final map is highly correlated with the true one, and the slam pose is a very good estimate of the true one logged via motion capture.

For an example of the results from the particle filter, a diagram showing various particle sets during a motion plan is plotted in fig. 4.

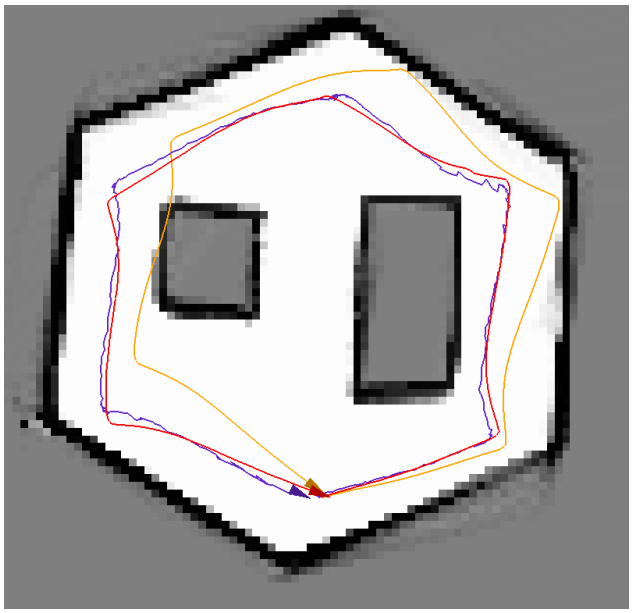

Fig. 3: SLAM output based on data logged in slam_obstacle_10mx10mx5cm.log. The true pose is shown in red, with the SLAM pose in blue, and the dead reckoning odometry pose in orange.

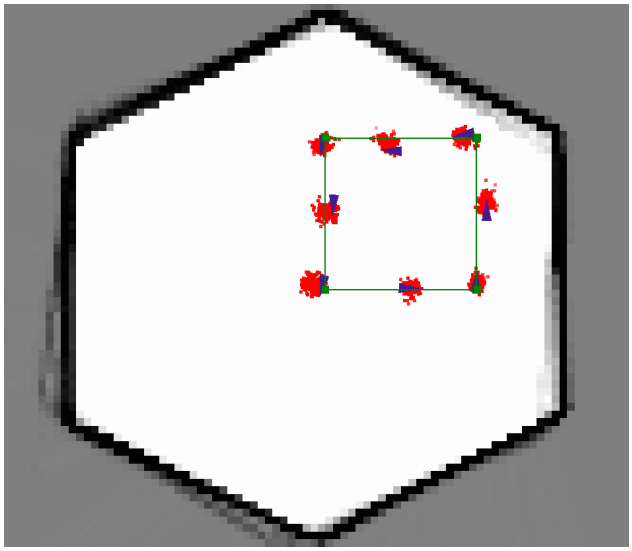

Fig. 4: The result of 300 particles being plotted at various points along the drive_square_10mx10m_5cm.log file. The particles are plotted at each leg's midpoint, as well as immediately following each 90 degree turn.

A* Successful attempts

	Convex grid (μs)	Empty grid (μs)	Maze grid (μs)	Narrow Constriction Grid	Wide constriction grid
Min	367	1331	512	5297	1142
Mean	367	3313	619.75	5983	2186.67
Max	367	6465	764	6669	3798
Median	0	2143	512	0	1620
Std dev	0	2253.32	94.73	686	1155.98

A* Unsuccessful attempts

	Convex grid (μs)	Empty grid (μs)	Filled grid (μs)	Narrow Constriction Grid	Wide constriction grid
Min	24	24	24	25	25
Mean	31	24.5	110.8	14869	25
Max	45	25	451	44534	25
Median	45	0	24	48	0
Std dev	9.89	0.5	170.122	20976.3	0

TABLE III: A* tests done in $astar\_test\_.cpp$. The top table are the attempts where the algorithm was able to find a valid path to the goal. The bottom one are the attempts with no solution.

2) SLAM Component Iterations: In terms of tweaks necessary to make SLAM perform well, primary efforts were focused on the action model, and the inverse sensor model. The initial action model was highly simplistic - the raw translation and rotation deltas from the immediate odometry timesteps were naively applied to each particle with some gaussian disturbance. While this showed decent results, it was found inferior to the RTR model presented previously. Additionally, the initial inverse sensor model was initially quite simplistic - a check was made against the cell where the lidar was expected to terminate, and then log odds were assigned based on the presence or absence of occupied cells at the destination, as well as previously along the lidar ray. This was found to be inferior to the ray casting method that was finally implemented.

3) SLAM Errors: Our SLAM implementation was ran against data which contained the ground truth pose data as obtained via motion capture. The figure in 3 shows the traces of the robot's path vs the true path and odometry. The mean, max, and standard deviation of the pose error in meters is shown in the table below.

TABLE IV: Pose error estimates

μ_{err}	$\max_{err}$	σ_{err}
0.057	0.19	0.05

Additionally, the robot was driven for a longer period of time (4 laps) around a similar square course. The final pose of the robot was relatively close, but not exactly on, the starting position - the error was predominatly in the +Y direction. The results of the robots motion is shown in figure 5 and the evolution of the delta between odometry and the SLAM position is shown in figure 6.

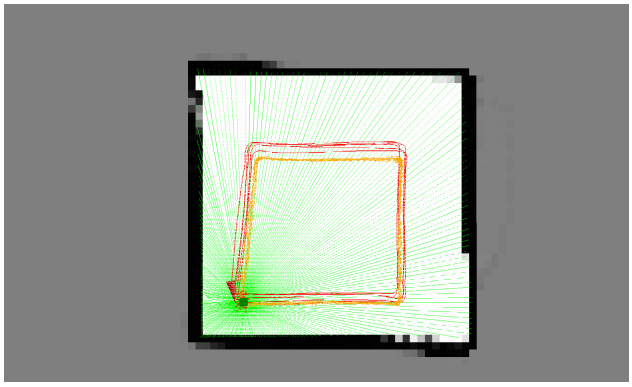

Fig. 5: A plot of the odometry and slam positions as the robot drives a four lap course. The odometry pose is shown and red and the slam pose is shown in orange

4) SLAM Runtime: We quantified the particle filter runtime for various particle counts. Results are shown in table V. Based on a linear fit of this data our algorithm can run with a maximum of 164 particles while still achieving 10Hz update rate.

B. Path planning

The `astar_test.cpp` provided by the instructor resulted in the values shown in Table III. The algorithm was not optimal in terms of finding the shortest path to the goal because the heuristic used included the inverse of the distance to the closest obstacle. This was done to

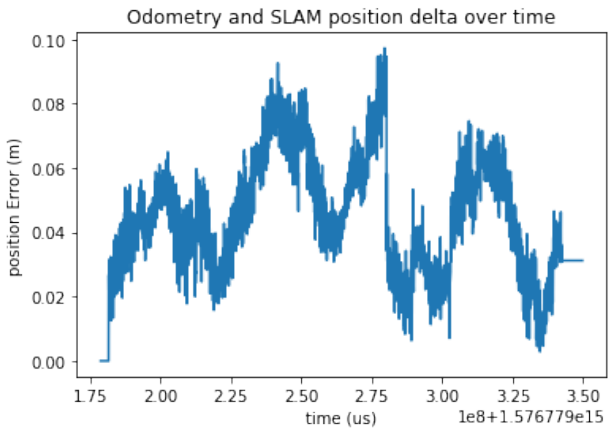

Fig. 6: A plot of the odometry and slam position error during the labs in fig 5

TABLE V: Runtime vs Particle count on RasPi

Particles	Runtime (μs)
100	6.31e4
300	17.94e4
500	29.62e4
1000	59.48e4

ensure that the robot moved far away from the borders. In this way, it was possible to minimize collisions with the walls due to errors in the mapping or the localization. Besides that, the algorithm took milliseconds as the worst case to find a valid path so it was fast enough for the competition.

In Figure 7, it can be seen how the robot is able to follow the A* path closely. It can also be seen how the difference between the odometry and slam positions differ.

IV. Discussion

The implemented SLAM algorithm produced more than acceptable results for the competition tasks. However, there are known areas of improvement had more time or resources been available. The existing implementation is not built with solving the 'kidnapped robot' problem in mind. Implementing this, however, would still have been feasible in the existing code architecture. Additionally, no extra precautions were made to deal with a dynamic map environment. While a certain amount of tolerance within the existing architecture exists for a change in ground truth map, future work could easily have focused on dealing with environmental perturbations.

Although the A* path planning algorithm worked well in the competition, there are some improvements

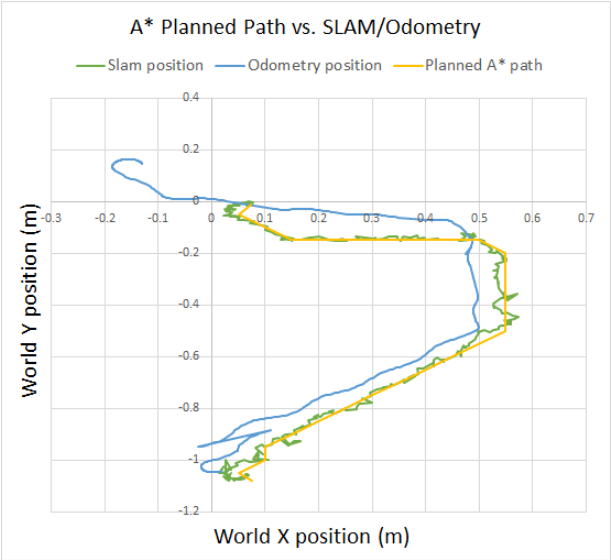

Fig. 7: XY plot of an A* path plotted on top of the SLAM and odometry positions.

that could have made it better. Implementing a function to reduce the number of points in the planned paths would have produced smoother trajectories and faster exploration.

The combination of our particle filter-based SLAM algorithm and A* path planning algorithm allowed our robot to successfully complete the *Looper* and *Dungeon Explorer* challenges with perfect scores on each task.

V. Conclusion and Future Directions

In this report, we detailed our development and implementation of the sensing, planning, and execution algorithms that allowed our mobile robot to navigate and build knowledge of unfamiliar environments. The robot used simultaneous localization and mapping (SLAM) to build its knowledge of the world and keep track of its position within the world, an A-star navigation algorithm to optimally navigate between two positions in the robot's world, and a rotation-translation-rotation motion control algorithm to efficiently navigate along the path from A-star. The results of experiments designed to test the accuracy of each component of this system showcased the robots ability to use these functions to build clear, accurate maps of the world and navigate it with high confidence.

Although our approach was successful on the proposed 2D environments, there are three fundamental limitations that would prevent this approach from mapping more complex environments: limited sensing capabilities, on-board computational constraints, and the inability to deal with dynamic obstacles. Therefore, we propose three possible future directions for research.

First, to get around the limited sensing of the 2D lidar, we propose using monocular RGB-D cameras to capture a larger field of view. Second, due to power and computational resource constraints, we propose a collaborative SLAM approach that utilizes working together to build a larger and more complex map [5]. Lastly, we propose incorporating object tracking to handle dynamic objects that might appear in an environment [6], [7].

REFERENCES

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[2] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (slam) problem," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 3, pp. 229–241, June 2001.

[3] P. Gaskell, "Botlab," University of Michigan, Tech. Rep., 2019.

[4] H. Choset, S. Hutchinson, K. Lynch, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun, *Principles of Robot Motion (Theory, Algorithms, and Implementation)*. The MIT Press, 2005.

[5] D. Zou and P. Tan, "Coslam: Collaborative visual slam in dynamic environments," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 2, pp. 354–366, Feb 2013.

[6] G. Q. Huang, A. B. Rad, and Y. K. Wong, "Online slam in dynamic environments," in *ICAR '05. Proceedings., 12th International Conference on Advanced Robotics, 2005.*, July 2005, pp. 262–267.

[7] C. Yu, Z. Liu, X.-J. Liu, F. Xie, Y. Yang, Q. Wei, and Q. Fei, "Ds-slam: A semantic visual slam towards dynamic environments," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2018. [Online]. Available: http://dx.doi.org/10.1109/IROS.2018.8593691

[8] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "Monoslam: Real-time single camera slam," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1052–1067, June 2007.

[9] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *Trans. Rob.*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016. [Online]. Available: https://doi.org/10.1109/TRO.2016.2624754

[10] T. Taketomi, H. Uchiyama, and S. Ikeda, "Visual slam algorithms: a survey from 2010 to 2016," *IPSJ Transactions on Computer Vision and Applications*, vol. 9, 12 2017.

[11] G. Younes, D. Asmar, E. Shammas, and J. Zelek, "Keyframe-based monocular slam," *Robot. Auton. Syst.*, vol. 98, no. C, pp. 67–88, Dec. 2017. [Online]. Available: https://doi.org/10.1016/j.robot.2017.09.010