Botlab 2: SLAM

Max Ernst
University of Michigan
maxernst@umich.edu

Andrew Greer
University of Michigan
alogreer@umich.edu

Dhillon Patel
University of Michigan
dhipatel@umich.edu

John Pye
University of Michigan
jepye@umich.edu

***Abstract*—In the second phase of building up the MBot and enabling intelligent and autonomous operation, it is necessary to deploy the SLAM algorithm first, in order to complete the final stage: path planning using A* and autonomous exploration of an environment. This paper discusses the methods and process of building Simultaneous Localization and Mapping (SLAM). The SLAM algorithm will utilize an occupancy grid for poses and the Monte Carlo Localization (MCL) particle-filter-based localization algorithm. Combining these two tools, we can use SLAM to construct a map of the MBot's enivornment.**

I. Introduction

Part 2 of Botlab deals with building towards implementing Simultaneous Localization and Mapping, or SLAM. Building in stages, the first stage will be creating an occupancy grid using known poses, followed by implementing the Monte Carlo Localization algorithm while using a known map, and finally integrating each stage for a complete, working implementation of SLAM where both the map and pose are unknown and simultaneously derived.

II. Mapping With Occupancy Grid

In order to describe the space around the robot, an *occupancy grid* is used. An occupancy grid is a method of representing an environment that breaks up a map into discrete sections, called cells. The likelihood of that cell representing an occupied space or a free space is tracked as a log-probability. This allows us to exponentially increase or decrease probability by linearly incrementing or decrementing probability - making for easier runtime execution. For the sake of this lab, we assigned each grid cell to be a 10 cm x 10 cm space, and the log-probability integer values to from -127 to 127, inclusively, with -127 meaning "certainly empty" and 127 meaning "certainly occupied".

The value of using an occupancy grid in this way is realized by using Breshenham's line algorithm. Any (x, y) coordinate in the environment can be represented as a discrete, integer coordinate in the occupancy grid. Therefore any point on the map that exists, whether be an obstacle or the MBot's position, can instead be represented as the cell it exists within. The LiDAR sensor on the MBot will provide data on laser scans as individual rays with a theta, the ray's angle, and a range, which is the distance the ray went until it contacted an obstacle. Breshenham's line algorithm helps us find the cells in the occupancy grid the ray passed through, and the cell the ray collided with.

Breshenham's line algorithm helps with scoring the log-probability of each cell being an obstacle or not. For each ray cast by the LiDAR, we use Breshenham's algorithm to find the cells in the occupancy grid that ray passes through, and the cell the ray collides with. We then lower the log-probability that each cell the ray passed through is an obstacle, meaning those cells or more likely to be free. We then increase the log-probability of the cell the ray collides with being an occupied. Each of these changes to log-probability is done by a small amount. Even though one ray does not provide enough information to completely fill out the map with high confidence, as the MBot casts more rays, a map can be drawn with high certainty.

We used the ground-truth poses provided in several log files, to construct the occupancy grid maps of each log respectively.

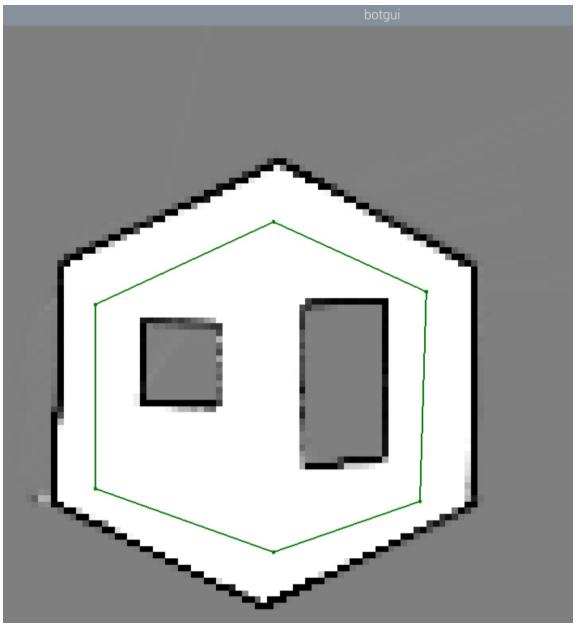

Fig. 1. Mapping log on 10mx10mx5m simulation using occupancy grid

Figure 1 shows the mapping log from simulating the MBot on the file obstacle_slam_10mx10m_5xm.log, with the command line argument –mapping-only. The brightness of each cell corresponds to the log-probability of that cell being occupied, with black being most confidently occupied and white being most confidently free. Gray represents unknown or little confidence either way. Even though the outlines of the walls in this map or not perfectly straight as they may be in reality, the overall structure of the map is retained. Note

the size of each pixel in the occupancy map, representing the 10cmx10cm resolution we decided on.

III. Monte Carlo Localization

Now that we have achieved mapping in simultaneous localization and mapping, we must tackle localization. We approached this using Monte Carlo Localization (MCL), which is a particle-filter based localization method which combines an action model, a sensor model, and a particle filter.

A. Action Model

The action model is a method of predicting the MBot's location in the map based on either its odometry data or its wheel encoder data. We decided upon using odometry data because it represents the MBot's movements in the coordinates of the world plane. We also have access to accurate odometry measurements, due to our implementation of Gyrodometry in Botlab 1.

An "action" is simply a movement of the MBot, taking it from an initial pose to a final pose. We can represent this movement as first a rotation δ_{rot1}, then a translation δ_{trans}, and then another rotation δ_{rot2}. Thus the entire action is represented as a vector u, where u is described as:

$$u = [\delta_{rot1} \;\; \delta_{trans} \;\; \delta_{rot2}]$$

This representation simplifies the movements of the MBot and allows for analysis and real-time adaptation. Given odometry data Δx, Δy, and $\Delta\theta$, the following equations can be used to calculate the action of the MBot.

$$\delta_{rot1} = atan2(\Delta y, \Delta x) - \theta_{t-1}$$
$$\delta_{trans} = \sqrt{\Delta x^2 + \Delta y^2}$$
$$\delta_{rot2} = \Delta\theta - \delta_{rot1}$$

Here, we calculate the first rotation and translation, where θ_{t-1} is the final angle of the last action. The second rotation is calculated by finding the difference between the $\Delta\theta$, the change in angle found with odometry, and α, the first rotation.

Given that we are sampling these readings from odometry, which is not accurate, each of the transformations in the action will have some error. To account for this, an error term is added to each transformation.

$$u = [\delta_{rot1} + \epsilon_1 \;\; \delta_{trans} + \epsilon_2 \;\; \delta_{rot2} + \epsilon_3]$$

These errors ϵ_1, ϵ_2, and ϵ_3 are sampled from normal distributions. Here, k_1 and k_2 are uncertainty parameters that increase the variance in the error distribution we are sampling from. k_1 represents the sampling variance in the rotation error and k_2 represents sampling variance in the translation error. These will eventually affect the dispersion of sampled particles in the particle filter. In the full action model, we additionally have the coefficients a_2 and a_4 which are values representing cross correlation between rotation and translation, and between both rotations respectively. They are omitted in our formulation, however, due to having been set to 0 and thus disregarded.

$$s_{r1} \sim \mathcal{N}(\delta_{rot1}, k_1|\delta_{rot1}|)$$
$$s_t \sim \mathcal{N}(\delta_{trans}, k_2|\delta_{trans}|)$$
$$s_{r2} \sim \mathcal{N}(\delta_{rot2}, k_1|\delta_{rot2}|)$$

To find k_1 and k_2, we needed to perform experiments to fine tune the parameters. This was accomplished by comparing our SLAM output to a solution SLAM output (with tuned k-values), as well as ground truth positional data. We iterated upon k_1 and k_2 until our SLAM solution, and the provided SLAM solution, aligned. This tuning was conducted purely visually to allow for speed and rapid iteration.

Parameter	Value
k_1	0.1
k_2	0.0015

(1)

Now, being able to represent the action of the MBot as two rotations and a translation with error, we can rearrange these equations to recalculate the change in the pose of the MBot. Equation 2 demonstrates how a sample, provided by the particle filter of the robot, evolves to a new potential pose.

$$\begin{bmatrix} x_{proposal} \\ y_{proposal} \\ \theta_{proposal} \end{bmatrix} = \begin{bmatrix} x_{sample} \\ y_{sample} \\ \theta_{sample} \end{bmatrix} + \begin{bmatrix} s_t * cos(s_{r1} + \theta_{sample}) \\ s_t * sin(s_{r1} + \theta_{sample}) \\ s_{r1} + s_{r2} \end{bmatrix} \quad (2)$$

The proposals calculated are then evaluated by our particle filter based on their likelihood of producing the received sensor measurements, with the most likely pose proposal becoming the actual pose for that time step.

These equations use the terms found in the action model to recalculate change in position of the MBot in world coordinates. Because we are including the error terms, sampling these transformations multiple times may result in different positions.

Thus, instead of sampling a single action, we sample the action multiple times to produce many "particles", each representing a possible position of the MBot after that action. This produces a probability distribution of possible positions which can then be scored and filtered to improve localization. These possible solutions, or *particles*, need to be filtered in some way, so that we can obtain our true position.

B. Sensor Model and Particle Filter

Now that we're able to determine a range of possible poses using the action model, we require a method of calculating the likelihood of all of those possible poses. This method is the sensor model. The sensor model uses laser scan data and the occupancy grid map to calculate the likelihood of a pose.

Given a group of particles, each representing a possible position the MBot may be in, we must find which particle is most likely to be the MBot's actual position. We do so by simulating the LiDAR scan data from every possible particle, and compare that to the results of the occupancy grid. Because

the occupancy grid is filled by the real scan data, the simulated scans can be graded on how well they line up. These simulated scans are graded on whether the rays terminate where they are expected to, based on the occupancy grid. Then, these scores are used to predict the likelihood that each particle is the MBot's real position.

We first characterize the maximum number of particles our MBot is able to process. Our MBot LIDAR updates at a frequency of 10 Hz, or 0.1 seconds. This means our particle filter needs to be able to process all data from the LIDAR in less than 0.1 seconds, so this time sets our maximum number of particles. To determine this, we performed 5 trials per sample count, at sample counts of 100, 300, 500, and 1000. Each time, we measured the time the particle filter function took to update. We then recorded the median time the filter took to update per sampling level.

Particle Count	Particle Filter Update Time (seconds)
100	0.00452
300	0.01365
500	0.02264
1000	0.04446

As we can see, our sampling points are all under our 0.1 second cutoff time. Next, we performed a linear regression on the execution times at different sampling levels, so we could determine the maximum possible number of samples our particle filter could incorporate. We then extrapolated to find the maximum number of permissible samples. We believe such an extrapolation was appropriate, due to the strong linear correlation seen in Figures 2 and 3. Figures 2 and 3 respectively show the time it takes to process some number of particles - the "processing time" - and the number of samples which could be processed in some amount of time - the "samples processed". The graphs are inverses of each other.

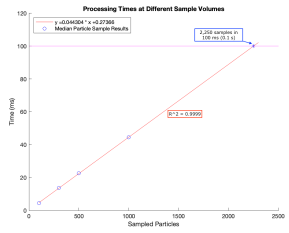

Fig. 2. Processing Time

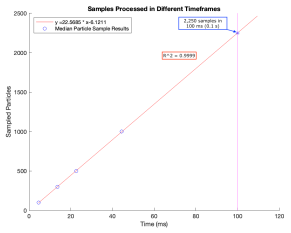

Fig. 3. Sample Processing Capacity

As seen in Figures 2 and 3, we can process at most 2,250 samples in 100 ms, or 0.1 s. Further testing is likely necessary to determine the exact sampling rate of the LIDAR, as well as the processing power of the Raspberry Pi onboard computer. However, this shows us that our planned testing regime of 100, 300, 500, and 1000 particles will be safely within our processing capacity, and so adjustment of the LIDAR sampling will not need to be conducted.

Next, we tested to ensure that our particle generator was indeed sampling particles from likely poses. We applied our localization algorithm to the "$drive_square10mx10m_5cm.log$" plot, to illustrate the inputs to our particle filter. The 300 particles sampled at each square corner after rotation, and each side midpoint, are shown below in Figure 4.

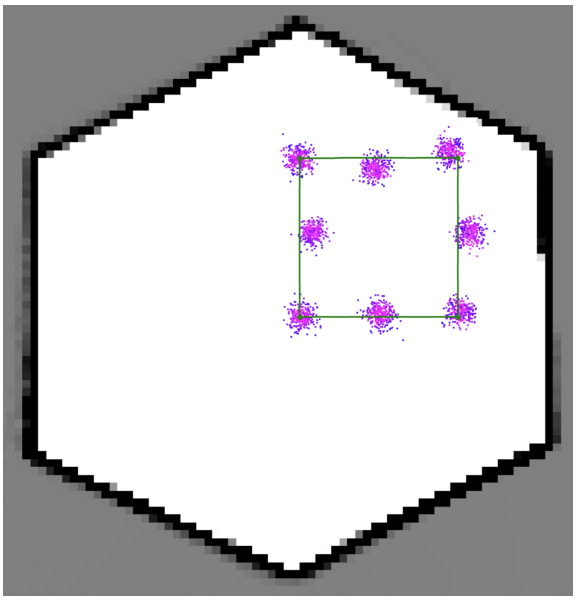

Fig. 4. Position of 300 particles at 1m intervals in a 2mx2m square

As we can see in Figure 4, our particle filter is indeed operational. Clusters of particles are correctly generated, and track with the movement of the robot. Although some clusters are clearly offset from the desired (green) path, this tracks with the actual motion of the robot, and so verifies our tuning of k_1 and k_2 and the ability of our localization algorithm to accurately keep track of the robot's position.

Finally, we set out to quantify the accuracy of our localization module. Such localizations run the risk of drift over time, and so it was necessary to evaluate whether or not our algorithm was capable of stabilizing to a steady error. We tracked the pose error between our SLAM estimate, and our odometry estimate, when tried against the more difficult $obstacle_slam10mx10m_5cm.log$ trial. We graphed the total pose euclidean distance error, and the individual pose component error, at different particle counts for our filter. The results are shown in Figures 5 through 20.

Figures 5 through 20 and the table above show the costs and benefits of including more or fewer particles in the particle filter. The table shows that utilizing more particles in the particle filter linearly increases update time. Longer update times will lag the MBot's localization, which may lead to further accumulation of error as the MBot's processing is running behind its actual movements. However, the figures

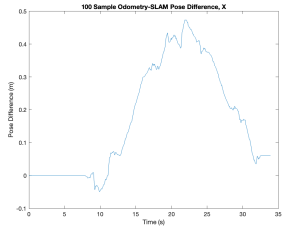

Fig. 5. 100 Particles: x

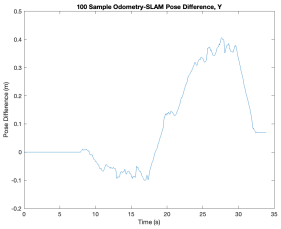

Fig. 6. 100 Particles: y

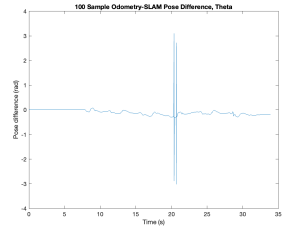

Fig. 7. 100 Particles: theta

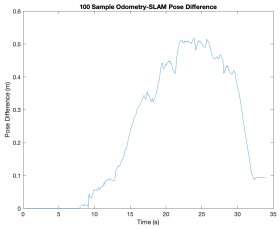

Fig. 8. 100 Particles: diff

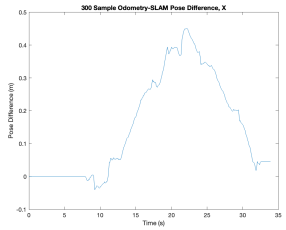

Fig. 9. 300 Particles: x

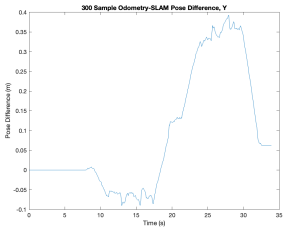

Fig. 10. 300 Particles: y

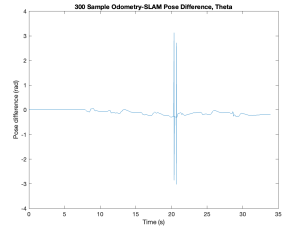

Fig. 11. 300 Particles: theta

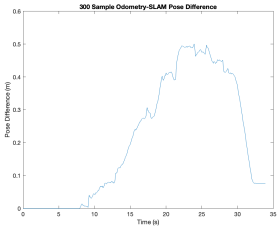

Fig. 12. 300 Particles: diff

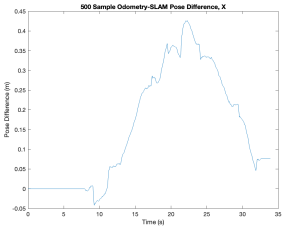

Fig. 13. 500 Particles: x

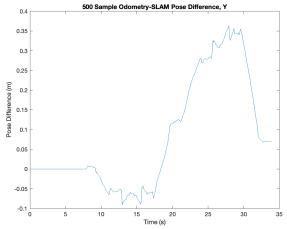

Fig. 14. 500 Particles: y

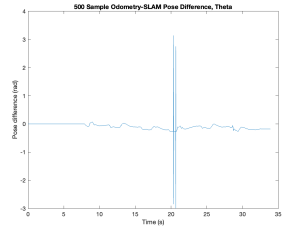

Fig. 15. 500 Particles: theta

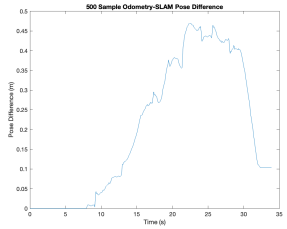

Fig. 16. 500 Particles: diff

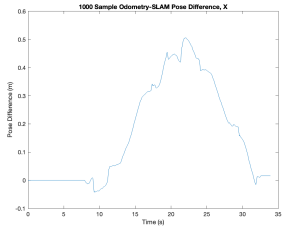

Fig. 17. 1000 Particles: x

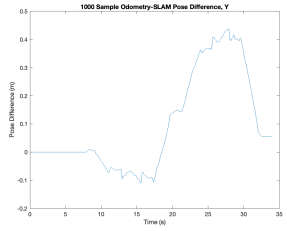

Fig. 18. 1000 Particles: y

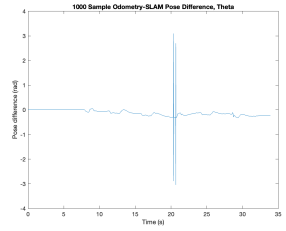

Fig. 19. 1000 Particles: theta

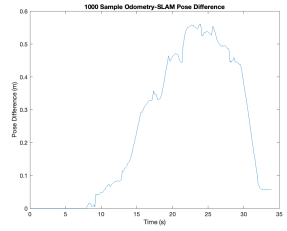

Fig. 20. 1000 Particles: diff

above show how utilizing more particles may lead to increased accuracy. For example, the graphs for the pose difference for 100 through 1000 particles show that 500 particles results in the lowest deviation in pose difference. Of particular note here are the graphs displaying the error in heading - Figures 7, 11, 15, and 19 - as we can see significant spikes in error at points. This is due to heading wraparound - a small heading error can appear extremely large, when one algorithm wraps back to -pi, and the other does not. This error is thus not overly concerning. We see for all other plots, quite significant positional error, on the order of approximately half a meter. This is not at all surprising. As was found in Botlab 1 (and further reinforced here), pure odometry dead reckoning is extremely prone to drift over time. This error thus highlights the accuracy of SLAM, and its resistance to drift over time.

IV. Simultaneous Localization and Mapping (SLAM)

Simultaneous localization and mapping (SLAM) is difficult because the mapping of an environment is informed by the MBot's localization in that environment, and the MBot's localization is informed by its mapping. To get around the circular nature of localization and mapping, these two systems must communicate actively and frequently with each other, hence the "simultaneous" aspect of SLAM.

The flow of information during SLAM is shown in Figure 21. The MBot's sensors provide information about its environment and movement through its wheel encoders, which informs odometry readings, and its LIDAR. This information is passed along to both our localization and mapping modules. Our mapping model updates the map around it, while the localization model uses the previous iteration's map and, based

on that information, takes samples and places the robot in a global positional frame. This pose estimate is finally passed back to the MBot, which takes some action based on it, and closes the loop.

The result of this approach is seen in Figure 22, where we again test ourselves against the $obstacle_slam_10mx10m_5cm.log$ trial. However, we do not have the benefit of an accurate map beforehand, as we did previously, and so we will utilize the full power of SLAM to construct one as we localize. Figure 22 shows the true pose in yellow, our estimated pose in blue, and the desired pose in green.

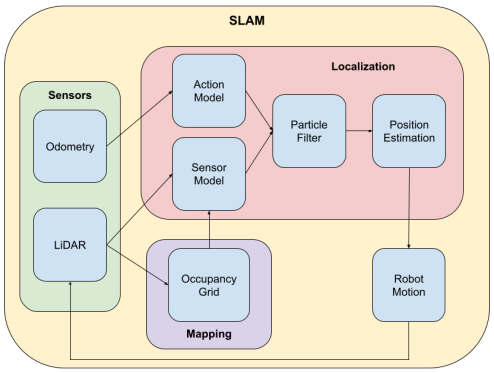

Fig. 21. SLAM System Block Diagram

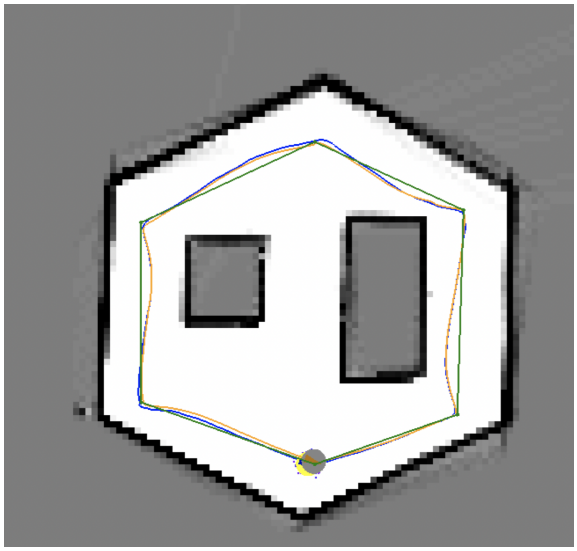

Fig. 22. Full Slam Map

Figure 22 shows the excellent performance of our SLAM algorithm in both localization and mapmaking. The SLAM positioning was overall very accurate, and the two paths only diverged marginally towards the end of its path before rejoining at the end. Two things of note are that the error between the two paths increased after rotations, and error also accumulated as more movement was taken, both then decreased again as time went on. These are to be expected because of inherent errors with odometry and SLAM readings. However, the overlapping lines suggest extremely good agreement between true pose and SLAM-estimated pose.

Additionally, the map produced is fairly accurate. There is some false readings for obstacles or free space outside the bounds of the map, which fortunately does not ultimately affect the MBot's belief of what is inside the bounds of the map. Though some of the walls are somewhat fuzzy, the general regions and areas the MBot are able and unable to traverse are very clear. Furthermore, the walls of both the arena and of the contained obstacles are very dark, indicating a high likelihood of occupancy - which is correct for this particular map. When compared the the provided solution map in Figure 23, we can see our agreement.

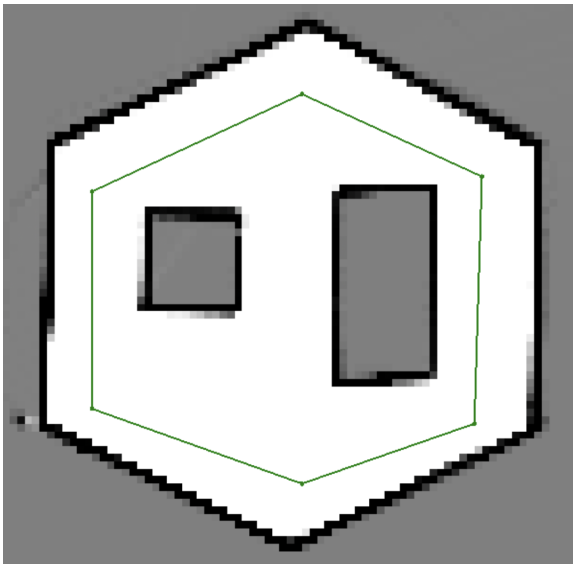

Fig. 23. Instructor SLAM Map

We then set out to quantify the accuracy of our SLAM algorithm by comparing it to the provided ground truth pose determinations throughout the robot's path. Once again, we took this accuracy measurement at 100, 300, 500, and 1000 particles in our filter to see the evolution of our accuracy as our processing power applied to the problem increased. To quantify this error we plot the accumulated Root Mean Square (RMS) error over time for the euclidean pose difference, as well as each pose component (X, Y, and θ). RMS is a cumulative measurement, and so the RMS error will only increase. The results are shown in Figures 24 through 35.

As we can see, although error climbs at the beginning of the robot's movement, the ground truth pose determination and the SLAM-provided pose estimation eventually converge, represented by the RMS error plateauing. We can see, however,

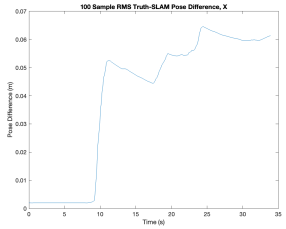

Fig. 24. 100 Particles RMS: x

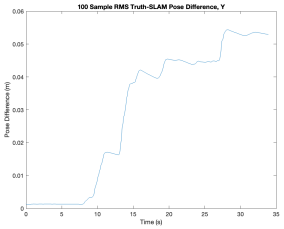

Fig. 25. 100 Particles RMS: y

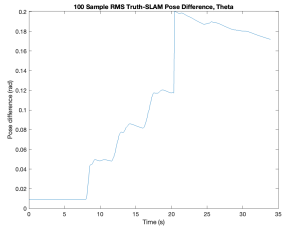

Fig. 26. 100 Particles RMS: theta

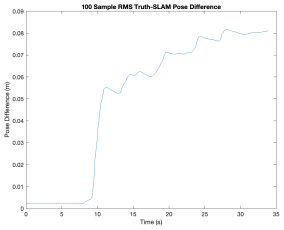

Fig. 27. 100 Particles RMS: diff

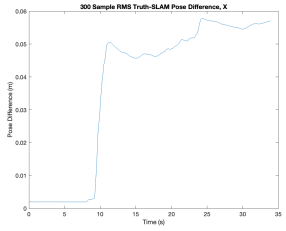

Fig. 28. 300 Particles RMS: x

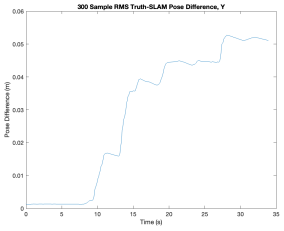

Fig. 29. 300 Particles RMS: y

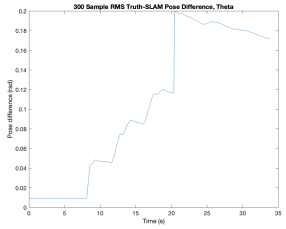

Fig. 30. 300 Particles RMS: theta

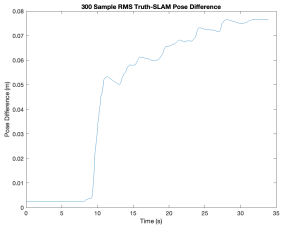

Fig. 31. 300 Particles RMS: diff

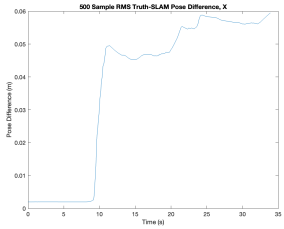

Fig. 32. 500 Particles RMS: x

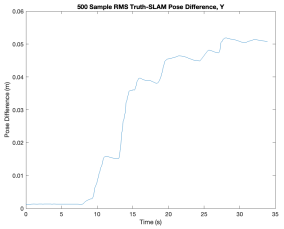

Fig. 33. 500 Particles RMS: y

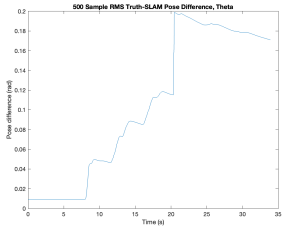

Fig. 34. 500 Particles RMS: theta

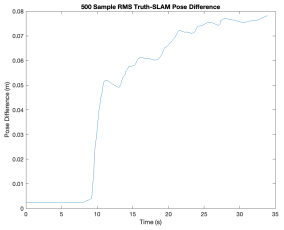

Fig. 35. 500 Particles RMS: diff

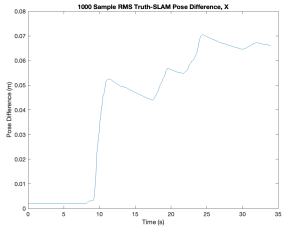

Fig. 36. 1000 Particles RMS: x

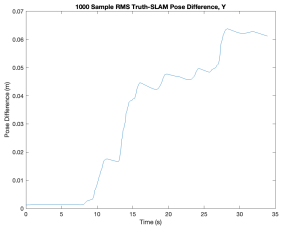

Fig. 37. 1000 Particles RMS: y

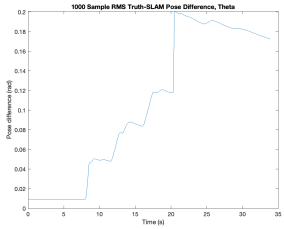

Fig. 38. 1000 Particles RMS: theta

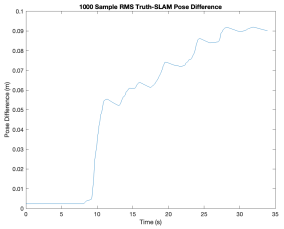

Fig. 39. 1000 Particles RMS: diff

that it stabilizes at a low level of error - under 10 cm across each test. Predictably, θ is more inaccurate, due to the weakness of SLAM in estimating heading (such a determination is far more effectively made by odometry). These figures demonstrate that our algorithm is indeed accurate, because its RMS error stabilizes to a low distance, and it is for the most part in agreement with the true pose.

These figures demonstrate our ability to now apply our SLAM implementation in a real-world test.

V. Checkpoint

Our final step was to test our algorithm in the real world, with the robot moving and the algorithm running on-the-fly. This considerably increases the challenges faces by the SLAM algorithm, which now needs to fun in real-time and with a moving robot, which can introduce unpredicted perturbations into the system, such as sliding or vibration. To do this, we constructed a maze as seen in Figure 40. The MBot began at one end of the maze, was teleoperated through the maze by a team member from their laptop, and stopped at the other end. Meanwhile, the LIDAR recorded its readings and fed them into our SLAM algorithm, and a map was constructed. The map is shown in Figure 41. Note: included in this folder are the log and map outputs of this checkpoint trial, for future analysis and verification of accuracy.

As we can see, the map created and the maze constructed visually appear to be in agreement. The blue trace in Figure 41 shows the SLAM-estimated pose evolution of the robot as it traversed the maze. We now plot the odometry pose estimate as well, in Figure 42, and then plot both pose estimations and evolutions in Figure 43 in the same global frame.

The respective estimations appear to be similar in terms of

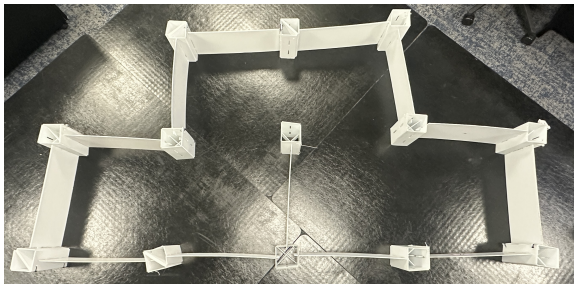

Fig. 40. Physical Maze Layout

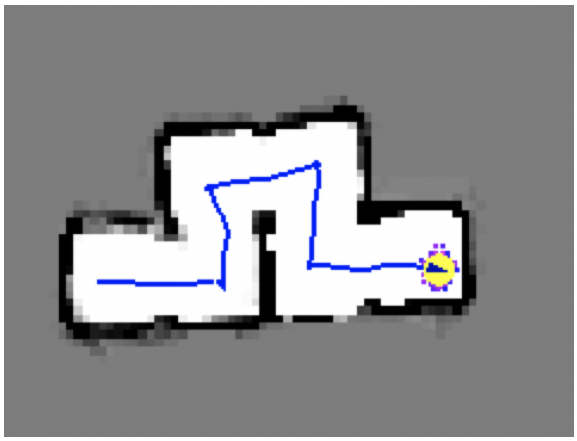

Fig. 41. Checkpoint SLAM Map

Fig. 42. Checkpoint Odometry Path

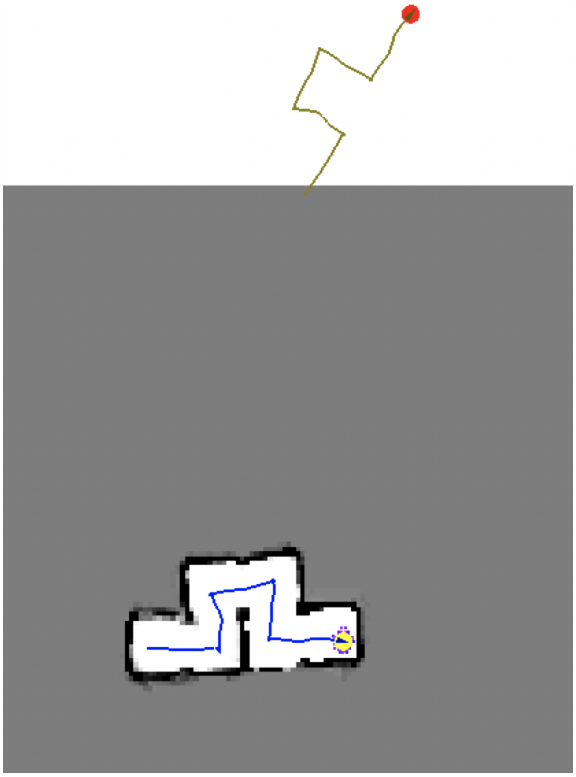

Fig. 43. Checkpoint SLAM Map and Odometry Path

offset, however, the odometry estimation has no idea where it actually is in the world. We see the trace both start and end *outside* the maze, in Figure 43. Furthermore, we can see that its heading is not aligned with the heading the robot actually took. This is to be expected, as odometry has no actual idea where it is with respect to the world, only with respect to its starting location. Therefore, for purposes of estimating speed (like in Botlab 1), it is adequate; for avoiding obstacles, as we attempt to do here, it is not.

Our SLAM algorithm does not significantly diverge from the provided SLAM architecture. We did not implement optional modules such as low-variance resampling, or cross correlation coefficients. Our slam system comprises a mapping module to update our occupancy grid; an action model to compute new poses based on samples; a sensor model to use sensor rays to localize the robot; and a particle filter to sample possible poses and compute the most likely one. Future development may add new modules to improve ground truth - estimation agreement, but due to the low RMS error between ground truth and SLAM estimation, such measures were not determined to be necessary.

VI. Conclusion

After completing Part 2 of the Botlab, we now have a working implementation of Simultaneous Localization and Mapping (SLAM). The first thing required to implement SLAM, was building a occupancy grid mapping algorithm to describe the space around the robot. After the mapping algorithm, we used the particle-filter-based Monte Carlo Localization (MCL) algorithm for robot pose predictions using sensor measurements, normalized particle weights, and a weighted mean of the particles. With the use of onboard odometry, the LiDAR, and our newly implemented occupancy map, we accomplished our implementation of SLAM.

REFERENCES

[1] EECS 467 WN24 Slides, Google Drive Folder. Available online: https://drive.google.com/drive/folders/1k2LsNlQYenkD3fTVSE6PTB4Cy1tz1eGF?usp=drive_link (accessed on February 7, 2024).

[2] Thrun, Sebastian; Burgard, Wolfram; Fox, Dieter. The MIT Press, 2006. ISBN-13: 978-0-262-20162-9.