Lab #5 Report: Localization

Team 2

Esha Ranade
Ian Pérez
Jason Salmon
Ritika Jeloka
Sydney Chun

6.4200 Robotics: Science and Systems

April 15, 2023

Edited by: Esha Ranade

1 Introduction

Sydney Chun
The goal for this lab was to determine our robot's orientation and position within a known environment. In our previous labs, we were able to program our robot to follow a wall through Light Detection and Ranging (LiDAR) scans and to move in accordance with objects within its environment through visual servoing; however, our robot had no sense of its own orientation or position within the environment it was interacting with. To resolve this issue, we took laser scans from the car, a map of the environment, and odometry data and implemented Monte Carlo Localization (MCL) to obtain the robot's pose. With this approach, the robot can now locate itself within a given map; this knowledge is essential to making decisions about the robot's future actions, which will aid us as we attempt to implement motion planning in our robot going forward. Visual localization is commonly used in robots across the automation industry. Companies like iRobot incorporate this feature into products like the Roomba, an automated vacuum and mop, to establish its the position within a room, enabling robot actions like automatically returning to its charging dock once a cleaning job is complete or when its battery is low.

2 Technical Approach

2.1 Problem Statement

Esha Ranade

This lab's primary objective was to establish our robot's position and orientation within a specified map, which for our team was the basement level of MIT's building 32 (the Stata Center). To achieve this goal, our team subdivided our localization implementation into individual deliverables, each of which is described in further detail below, to effectively modularize our code structure and testing process:

1. **Motion Model.**

   The motion model processes odometry data, or the change in pose of the car from one time step to the next time step, to update the car's pose from the previous time step to the current time step.

2. **Sensor Model.**

   The sensor model serves to reduce the number of particles distributed by the model sensor by finding the likelihood of measuring a particle at a point from a hypothesized position on a static map at a given time.

3. **Particle Filter.**

   The particle filter combines the motion model and sensor model into a complete MCL algorithm which resamples the particles and calculates an inferred pose based on a weighted average of the particles.

4. **Testing in Simulation.**

   Before pushing our code to the robot, our team ensured that our implemented solution performed well in theory, which we tested through a simulation of our projected particles.

5. **Testing on Robot.**

   Our final step was to adapt our MCL algorithm code to run on the robot and analyze its performance in determining the robot's pose relative to our observations about its position and orientation at any given time.

2.2 Full process

2.2.1 Motion Model

Ian Perez

Given the car's odometry data, represented as a vector, and a set of particles, the motion model's goal was to move those particles by the amount described in the odometry message and to add some noise to the data. The odometry data is given in the car's frame, but we need it in the world frame, so for each particle, we take its θ_i to build the rotation matrix

$$R_c^w = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \\ \sin\theta_i & \cos\theta_i \end{bmatrix}$$

Now all that is left is adding the car's odometry data $(\Delta x, \Delta y)$ transformed to world frame to the previous particle position (x_i, y_i)

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \\ \sin\theta_i & \cos\theta_i \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} + \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

In order to get the new θ value all we need to do is add the old one to the new one received in the odometry message.

$$\theta_i' = \Delta\theta + \theta_i$$

Since the odometry data we receive from the car is inherently noisy, we need some way of representing that in the final positions of the particles. We opted to sample from a Normal distribution centered at 0 where we manipulated the standard deviation to dial in the noise levels as we deemed appropriate. More detail is on how we arrived at our final values for the standard deviatation is presented in Section 3. We sampled a unique random value for each particle and each variable within each particle denoted as $(x_{r,i}, y_{r,i}, \theta_{r,i})$.

$$\begin{bmatrix} x_i'' \\ y_i'' \\ \theta_i'' \end{bmatrix} = \begin{bmatrix} x_i' \\ y_i' \\ \theta_i' \end{bmatrix} + \begin{bmatrix} x_{r,i} \\ y_{r,i} \\ \theta_{r,i} \end{bmatrix}$$

2.2.2 Sensor Model

Esha Ranade

The sensor model assigns each particle a likelihood weight by comparing a given sensor reading z_k to the actual distance d. Giving a better hypothesis a higher weight makes it more likely to be sampled on the next iteration, and allows bad hypotheses to be filtered out.

Because the sensor model employs several algebraic computations, we opted for a strategy in which the probabilities of each combination of z_k and d are pre-computed and saved in a table, and later conveniently referencing the cell indexed by the desired pair of measured and actual distance values. The calculations involved in this process are expanded upon in section 2.3.1. This greatly increases the efficiency of our sensor model by ensuring that the most time-consuming processes only occur once. The table is normalized such that each column has a sum of 1.

The sensor model also involves ray casting, though its implementation was provided to our team. This process consolidates the input data into a matrix with each particle mapped to its lidar beams. In our implementation, we also downsample our laser scan. By changing parameters, we ultimately found that working with around 73 of the original >1000 beams was optimal: it resulted in good performance without having to compromise accuracy, allowing us to increase the number of particles we could maintain in real time to around 1000.

For a given particle, we collect the pre-computed probabilities of each of its remaining rays after downsampling, comparing the distance it measures to the reading of the car's LiDAR sensor at the corresponding angle. Because each measurement is independent of the others, we can multiply each of these probabilities to compute the overall likelihood of that particle representing the car itself, and assign the particle a weight proportional to its likelihood.

We conduct several steps of post-processing to ensure our data is in a format conducive to what it will ultimately be used for in the particle filter. All of the lidar observation values and ray casting scans are in meters; to convert them to pixels, we must scale them by a factor of $\frac{1}{self.map\_resolution * lidar\_scale\_to\_map\_scale}$. The distributions of the lidar and ray casting distances must also be clipped such that any value above z_{max} is replaced with z_{max} and any value below 0 is replaced with 0. To make the probability distribution less peaked, we raise each of the sensor model's probabilities to a power of $\frac{1}{3}$.

2.2.3 Particle Filter

Ian Perez

Whenever the car receives odometry data, we pass in the current particles with the odometry data to the motion model to evaluate the new particle positions. When the car receives lidar data, it passes that information into the sensor model to assign a weight to each particle. Based on those weights, the particles are resampled.

2.2.3.1 Pose Initialization

In order to start the localization process, we need to give its approximate location in the map. We do this by publishing the initial pose of the car to the "/inititalpose" endpoint. After this we distribute a given number of particles around that location. We add to these particles some noise sampled from a gaussian distribution centered at zero and manipulated by controlling the standard deviation. More detail on how we arrived at these values can be found in Section 3. Each particle was given an equal weight to be used when resampling later on. Additionally, a flag was set to start the algorithm.

2.2.3.2 Resampling

After the pose is initialized, whenever the car received LIDAR data, it will pass it to the sensor model so that it outputs a normalized weight for each particle. Then we resample by choosing particles randomly such that a given particle is chosen with probability equal to their weight.

2.2.3.3 Calculating inferred pose

Once the particles have been resampled, the inferred pose is calculated based on the weighted average of the particles. This means that the pose is calculated by taking the weighted sum of the position and orientation of each particle, where the weight is the normalized weight assigned to each particle by the sensor model. We opted for a weighted average because it was more reliable with outliers since they would a have a low weight. Additionally, special attention was taken in computing the average angle where we used a circular mean instead. This results in a more accurate estimate of the true pose of the system.

2.2.4 Testing in Simulation

Sydney Chun

The models were tested in simulation to ensure correct implementation and parameter optimization before testing on the robot. Four unit tests were used to evaluate the motion and sensor models as well as the precomputate model and map callback functions in the sensor model. The particle filter was tested in both simulation and autograder test cases, revealing an issue with the noise implementation causing particles to move away from the car after initialization. After adjusting the noise parameters, the optimal path was achieved with a standard deviation of position noise 0.8, angle noise 0.8, and particle pose noise 0.04.

2.2.5 Testing on Robot

Sydney Chun

During the process of launching the localization file on the robot, we encountered an issue where the required files were missing from the car, preventing the file from being launched. After thorough debugging, we identified that a workspace was missing in the ROS directory of the car. Further investigation led to conflicts in the operating systems of the car and the local docker used to import the workspace. The libracecar_simulator.so file, which was compiled with x86-64 architecture, was not compatible with the aarch64-linux-gnu architecture required to launch the localization on the car.

Jason Salmon

Additionally, having not used LiDAR in the last lab, we initially failed to account for the fact that the Velodyne data required pre-processing. We eventually retrofitted our code from the previous implementation in order to recombine the data and rotate the array to account for the angular offset.

After necessary fixes, we used a series of real-world tests to adapt our noise parameters. We would eventually achieve relative success, noting that performance was not only a function of pose initialization and noise, but also the distance from the router.

2.3 Required Computations

Jason Salmon

2.3.1 Sensor Model

For the sensor model, we accept float values from the LiDAR sensor. However, instead of using this float value and calculating the probability each time, we instead create a discretized table for ranges of z_k and d based on four different probabilities:

- The probability of the LiDAR measurement being shorter than expected.

$$p_{short}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{2}{d} \cdot \left(1 - \frac{z_k^{(i)}}{d}\right) & \text{if } 0 \leq z_k^{(i)} \leq d \text{ and } d \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

- The probability of the LiDAR measurement being mistakenly reported as the maximum possible value.

$$p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} 1 & \text{if } z_k^{(i)} = z_{max} \\ 0 & \text{otherwise} \end{cases}$$

- The probability of the reported LiDAR measurement being completely random.

$$p_{rand}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if } 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

- The probability of the LiDAR measurement actually corresponding to the distance value.

$$p_{hit}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(z_k^{(i)} - d\right)^2}{2\sigma^2}\right) & \text{if } 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

Each pixel in the table corresponds to a particular (z_k, d) pair, with the overall entry being the normalized sum of the four probabilities for each pixel (the Gaussian distribution for p_{hit} is normalized before being added to the other cases, and the entire table is then normalized across d values).

Note that, during evaluation, the sensor model will receive packets of possibly several hundred ranges from the LiDAR and even larger arrays from ray tracing. Therefore, for each particle being evaluated, there are several hundred probability lookups, each corresponding to a single comparison between a predicted range and the measured range. Assuming these probabilities are independent, we then multiply the several hundred probabilities to yield the probability of the robot having the pose of each particle given their respective positions.

2.4 ROS Implementation

Jason Salmon

The implementation in ROS chiefly involved 2 subscribers and 2 publishers implemented within the particle filter node. Our code needed to listen to both the odometry data from the robot itself during real-world trials ("/odom"), and to the LiDAR data (rotated and recombined) being sent from the Velodyne sensor ("/scan"). When receiving odometry data, the callback uses the new information to re-evaulate the sensor model, then compute and publish the average pose of the particles being considered ("/pf/pose/odom"). Similarly, when receiving LiDAR data, the callback evaluates the sensor model, resamples the particles based on their respective probabilities, and then publishes the average pose. No other nodes were created for this implementation; using the natively provided nodes/data streams from the robot and our retro-fitted code from adapting the Velodyne output in previous labs, we were able to obtain all necessary data for calculation.

However, there are other intricacies at play during the initialization of the processes. For one, the robot initially has no concept of what its workspace looks like, and so we use a subscriber to obtain the relevant map of the area. Additionally, the robot has no concept of how it is oriented in its workspace, and so we use another subscriber to feed in an initial pose estimate using RViz ("/initialpose").

3 Experimental Evaluation

Ritika Jeloka

After coding our motion model, our sensor model, and our final particle filter as outlined above, we decided to test our implementation.

3.1 Simulation Results

To test our odometry implementation in a car, we ran our algorithm in simulation. Figure 1 shows our original simulation results, first with no odometry noise and then with more odometry noise.

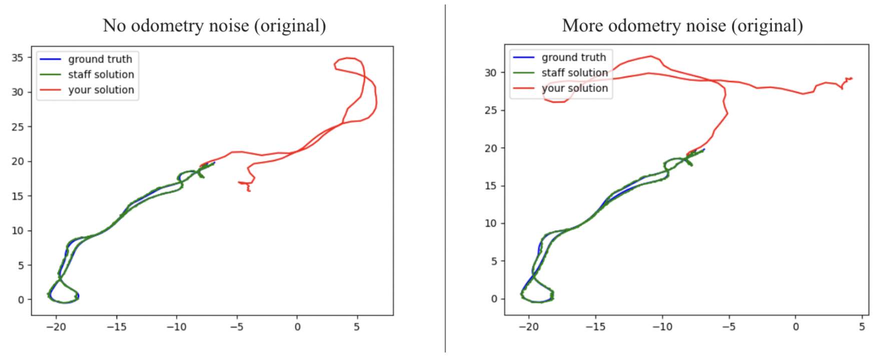

Figure 1: a. [left] Original simulation results
b. [right] Added noise results

Figure 1 shows that our original solution is similar to the ground truth/staff simulation, but it is shifted by some rotational degree. Adding noise shifted our results on the rotational axis, suggesting that adjusting noise could lead to a more accurate solution.

Therefore, we tried many different values to initialize our pose and add noise to the motion model. Our testing strategy involved isolating individual variables to identify the effects of increasing or decreasing their value. Some of our tests, along with our final values are outlined below in Table 1 shown, the average test result was around 91%.

With our adjusted noise values, we reran the simulation to get the messy path on the left shown below in Figure 2, to eventually have a path that more closely resembles the ground truth, as shown in the right side of Figure 2.

STD x	STD θ	STD noise of particle pose	Test 4 Score	Test 5 Score	Test 6 Score
0.2	0.17	0.04	0.8702832443	0.871541787	0.8576982611
0.4	0.17	0.04	0.8690737499	0.8602784522	0.8643140301
0.7	0.17	0.04	0.8638998863	0.8460254228	0.8811754228
0.3	0.5	0.04	0.8524413702	0.866314078	0.8540545215
0.3	0.9	0.04	0.8613089041	0.8644329635	0.8639538179
0.3	0.05	0.04	0.8678623962	0.8768083493	0.8752920015
0.9	0.01	0.04	0.8677734225	0.8637340856	0.8653126608
0.05	0.01	0.04	0.8647137966	0.8599826566	0.8733649493
0.4	0.01	0.04	0.8696016187	0.8561204809	0.8601370979
0.4	0.01	0.01	0.9174215759	0.9068031058	0.9107898484
0.4	0.01	0.05	0.8925843384	0.8853362203	0.8689829469
1	1	0.1	0.9091475927	0.9073945469	0.9068741333
1.5	0.17	0.04	0.8501244494	0.8808324337	0.8585042332
0.15	0.017	0.04	0.8668375602	0.8507393358	0.8606072398
0.12	0.17	0.04	0.8704290922	0.8645409454	0.8701407381
0.8	0.8	0.2	0.9079538989	0.9070723515	0.9162440491

Final Values	0.8	0.8	0.2	0.9079538989	0.9070723515	0.9162440491

Table 1: Adjusted noise values for initial pose and motion model and their corresponding test results

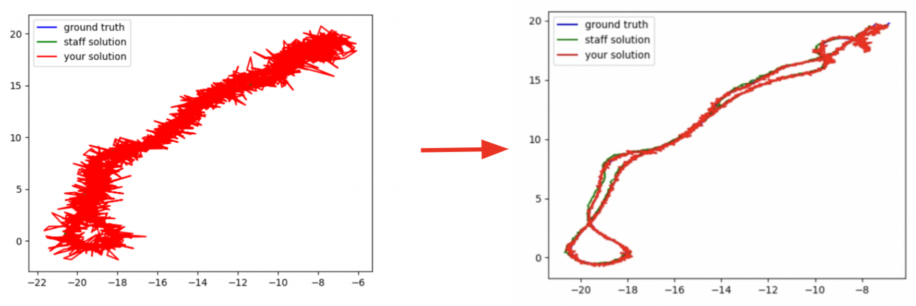

Figure 2: Noise Reduction to match ground truth

After having an algorithm that produces a simulation that closely resembles the ground truth, we wanted to quantitatively test our algorithm. To do this, we first computed convergence rate. We did this by timing how long the simulated racecar ran until the standard deviation of the x and y particles were under 0.01. As shown in Table 2, the average across 5 trials was around 0.03 seconds for both x and y coordinates.

Trial number	Time until std x <0.01	Time until std y <0.01
1	0.027366876	0.024885177
2	0.03369379	0.023045063
3	0.026900053	0.047758102
4	0.028886079	0.041861055
5	0.035041809	0.026783943
Average Time	0.0303777214	0.032866668

Table 2: Seconds until the standard deviation of the distributed particles' x and y coordinates fall below a threshold of 0.01

Another quantitative metric we used was computing the Cross Track error. This is the positional distance between the ground truth and average pose, or in other words, the positional error. As shown in Figure 3, this error decreases with time as the simulated racecar moves and odometry is being calculated, taking approximately 0.3 seconds to initially achieve a steady low value. We have noted that the decrease in cross track error (Figure 3) takes slightly longer than the convergence rate (Table 2). We believe that the particles have converged under our arbitrarily chosen threshold earlier on, but continue to move closer to the ground truth as time progresses. The higher degree of variation across poses initially is visualized in Figure 5, before all poses converge into having more aligned positions and orientations as more odometry data is collected.

Figure 4 visualizes a phenomenon we observed during our testing of the particle filter algorithm in simulation, where the particles appeared to have "jumped ahead" of the robot at intervals of around 1.0 second. We have several hypotheses about what may be causing this behavior, but particularly due to the consistent frequency at which this occurs, we believe there may be an error with the rate at which we are sampling our particles or a discrepancy between the rates at which certain topics are published to.

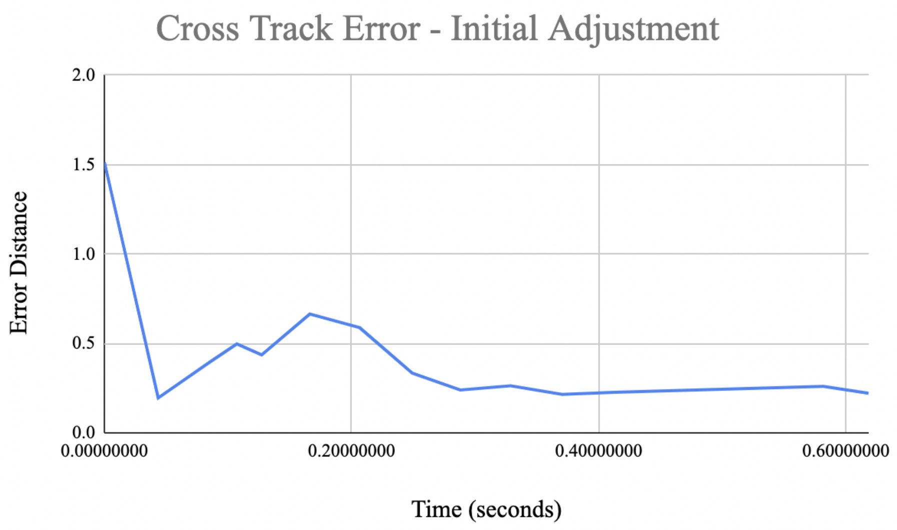

Figure 3: Error distance between ground truth robot pose and average particle during initial adjustment

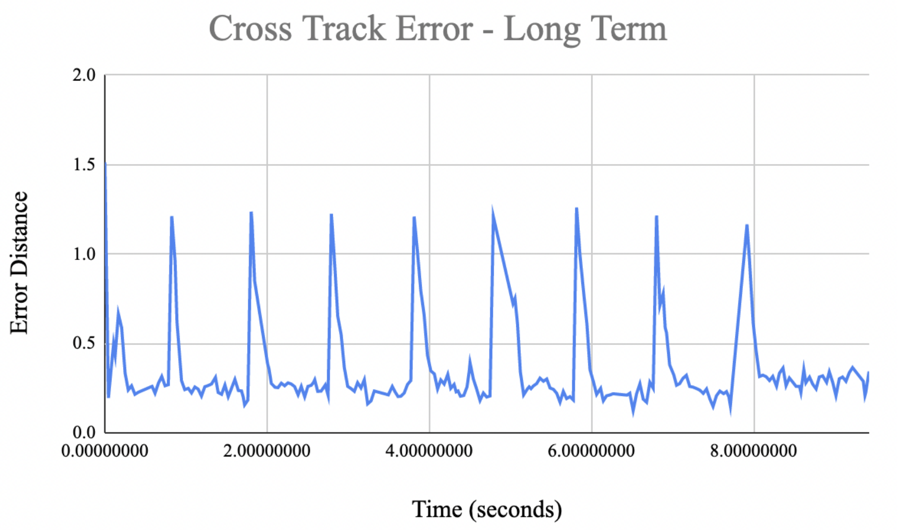

Figure 4: Error distance between ground truth robot pose and average particle across full run

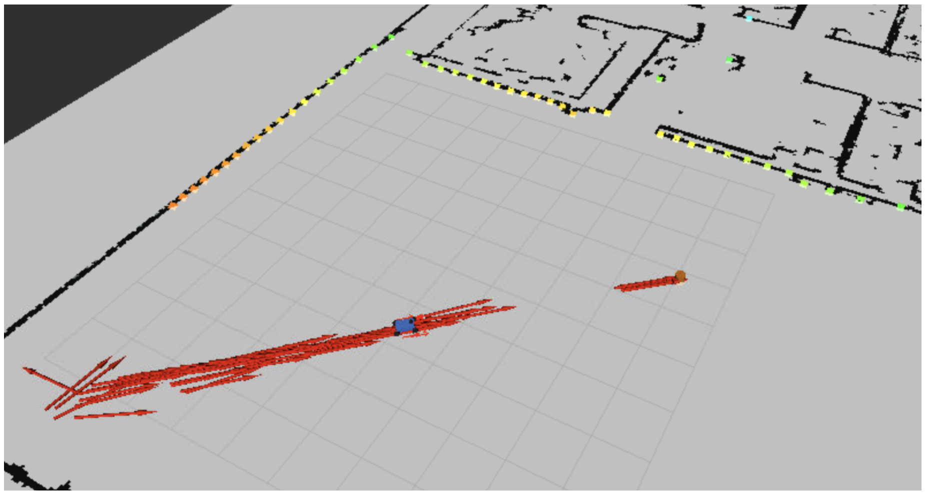

Figure 5: Visualization of increased pose variation initially

3.2 Robot Results

After verifying our implementation in simulation, we moved to test localization in the actual robot. We first started with 200 particles with 100 beams each. We then slowly incremented the number of particles upwards and the number of beams per particle downwards until we ended up with an accurate path of the robot around Stata basement. Figure 6 and 7 below show the localization of our robot in a hallway in the Stata basement. In Figure 7, it can qualitatively be seen that the robot's placement in space aligns with the simulation.

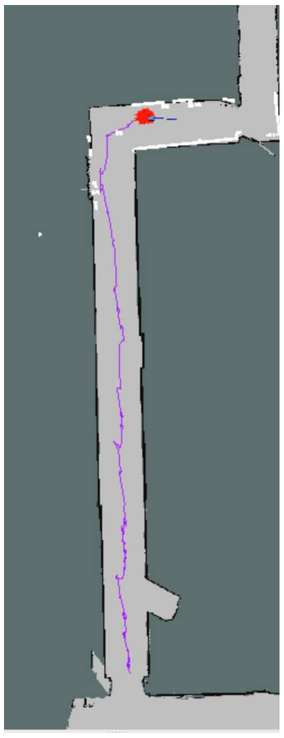

Figure 6: Final path

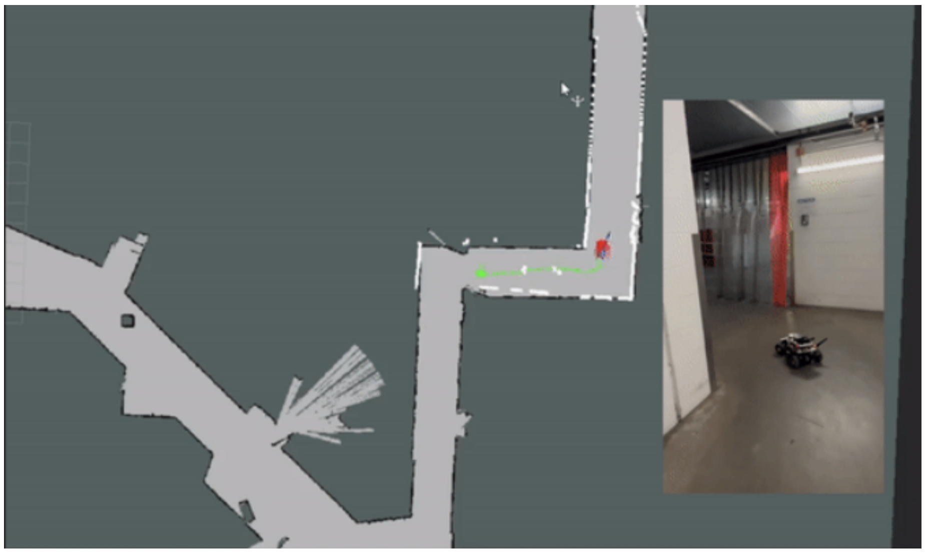
Figure 7: Final path with the Robot

4 Conclusion

Esha Ranade
Through this lab, we were able to achieve our goal of establishing the robot's pose, consisting of information on its position and orientation on a given map. This was accomplished using data from the robot's LiDAR sensor scans, a provided map of the environment (in our case, the Stata basement), and odometry data collected from the car's movements. We implemented Monte Carlo Localization (MCL), or particle filter, which created a distribution of hypothesized robot states as particles, resampled them according to their likelihood of representing the true robot pose, and calculated the weighted average of the particles to ultimately determine the robot's estimated pose. Knowing the robot's position and orientation on a map can inform its further actions; this development in our robot's capabilities puts our team one step closer to addressing future objectives, such as path planning.

5 Lessons Learned

5.1 Ritika

This lab showed me that iteration is a lengthy process. Although we had our code done a week ago, the process of adjusting noise and particle parameters, and then testing these adjusted values takes a substantial amount of time that we should account for. This lab also showed me the importance of teamwork as my team split up this lab into many parts and together, accomplished our goal.

5.2 Ian

During this lab, I learned the importance of understanding the underlying math and concepts behind the algorithms we were using. In particular, understanding how the particle filter works and how it utilizes Bayesian inference to estimate the robot's position helped me understand how to properly tune the parameters and improve the accuracy of our results. Additionally, I realized the importance of constantly testing and debugging our code to ensure that it was functioning properly. This involved carefully analyzing the simulation output and autograder results, and using the error messages to identify and fix issues. Overall, this lab reinforced the importance of taking a careful and methodical approach to programming and problem-solving.

5.3 Esha

Working on this lab reinforced the importance and value of both teamwork and modularization for me. Throughout the whole process, our team ran into different issues, many of which were related to the operating systems we were independently working on. Working to each person's strengths, and individuals taking initiative to contribute in the ways that they (and their OS!) were capable of, allowed us to be productive despite these limitations. By working in parallel to change noise parameters during our testing process, we were able to reduce the bottleneck of the autograder's response time. Therefore, efficient teamwork really improved our experience of completing this lab. Additionally, when our team realized this lab involved the use of our Velodyne sensor, which we had struggled to work with during the wall following lab, we were slightly dejected. However, being able to reuse a node that we had implemented for wall following, which updated the scan returned from the Velodyne sensor into a format consistent with the Hokuyo sensors and simulation, significantly simplified the process, allowing us to really appreciate the concept of modularity.

5.4 Jason

Throughout this lab, the biggest question was around time. It generally took a lot of time for each step of the lab, because unfortunately, we experienced many hurdles. There was time needed for debugging, time for the debugging *before* the debugging, time for experimentation and time for testing. The experience really highlighted the importance of making our processes as efficient as possible. For example, we might have been able to test more if we have been submitting to the Gradescope autograder in parallel from the beginning. Nonetheless, these are very important lessons that I'll hope to keep in mind for the next lab.

5.5 Sydney

This lab forced me to focus on the process of debugging. Our team was able to get our code working fairly quickly; however, the simulation and autograder demonstrated our code did not properly reflect the robot's position in the map. As I wasn't as involved with the coding of the particle filter, I learned how to read other people's code and debug it. Furthermore, I also learned the difficulties of working with different operation systems and how to fix them using error messages.