Lab 5 Report: Localization

Team 16

Jared Boyer
Daven Howard
Amy Ni
Niko Ramirez
Kayla Villa

1 Intro - Amy

Localization is a crucial task in navigational robots. As a robot traverses its environment, it needs to have an accurate estimation of its pose in order to autonomously plan paths and follow trajectories. In this report, we present our implementation of Monte Carlo Localization on an autonomous racecar, allowing the car to estimate its own position in relation to its surroundings. Our implementation separates the problem into three connected sub modules: a motion model to digest odometry with noise, a sensor model to measure the probability of a lidar scan, and a final particle filter that employs these two, generating a dynamic cloud of potential robot poses. By combining the set of potential poses into one estimate, our racecar is able to accurately and quickly estimate its location in a mapped environment.

Our approach begins with the motion model, which predicts a set of possible poses of the racecar given its odometry data. Odometry data are the racecar's changes in pose with respect to time and can be used to calculate the racecar's next pose at each time step. Integrating odometry data provides a useful estimate of the robot's pose in deterministic scenarios, but it is very inaccurate in the real world where friction and physical interactions accumulate unaccounted errors when the car is moving. In order to model these effects, our motion model keeps track of a large set of possible robot poses, adding Gaussian noise each time we propagate a particle with odometry data. By maintaining this set of possible robot poses, we can then use the sensor model to decide which of these estimates are actually the most likely.

The sensor model takes in a set of poses and a scan from the racecar's on board LIDAR scanner to determine each pose's probability of being the actual pose. Because we assume that the map of the racecar's surrounding environment is given, the sensor model can find what the LIDAR scan of each pose should look like using ray casting methods. These scans are then compared with the actual LIDAR data to calculate how well each pose's expected scan matches with the real scan. The details of how exactly these probabilities are calculated will be discussed in the following sections.

The culmination of our solution is a particle filter, which estimates the racecar's true pose using both the motion model and the sensor model. The particle filter begins by initializing a set of two hundred particles, each representing a possible pose of the racecar. Everytime the racecar receives odometry data, the particle filter calls motion model to update the pose of each particle. Simultaneously, everytime the module receives LIDAR data, it calls sensor model to calculate the probability of each particle as mentioned previously and resamples a new set of two hundred particles based on their probabilities of being the true pose. Lastly, whenever the particles are updated, the particle filter calculates a single estimate of the racecar's pose by finding the average of the poses represented by the particles.

Localization is an important step in building a robot capable of fully-autonomous navigation. In the following sections, we will discuss our implementation in more detail and analyze the performance of the racecar.

2 Technical Approach

As previously mentioned, our primary goal for this lab was to build localization capabilities onto our car so that it would be able to estimate its pose in a known environment. This was done through an approach called Monte Carlo Localization. Monte Carlo Localization works by keeping track of the poses of a set of particles which represent possible poses of the actual car. The poses of these particles are updated using odometry data as well as Gaussian noise to model potential odometry error. This is performed by a motion model, which is detailed more in section 2.1 below. The poses of the particles are also updated when LIDAR scan data is received from the car, allowing the particles to be resampled based on their probabilities of being the car's true pose. This is carried out by a sensor model, which is written about in more detail in section 2.2 below. The procedures of both motion model and sensor model are then brought together in a particle filter. The particle filter calculates the average pose of the updated set of particles, providing an estimation of the racecar's current pose. Particle Filter is written about in more detail in section 2.3 below. These interconnected procedures that make up Monte Carlo Localization allow the car to localize itself in a known environment, a capability crucial for other tasks like path planning and trajectory following.

2.1 Motion Model - Daven

The motion model is one of the main parts of our implementation of Monte Carlo Localization. Given the odometry from the racecar and the previous pose of each particle, our motion model calculates the new poses of the set of particles. Each of these particles represents an estimation of the pose of the racecar at a given time. Every time odometry data is collected from the racecar, we add this movement data to each of the particles and update their poses to maintain reasonable estimations of the racecar's pose.

When updating particle poses, if we were to add the same odometry to each of the particles, all of them would remain directly on top of each other with the same pose. While this was ideal for unit testing and visualizing the deterministic performance of our particle filter, we don't want a uniform set of particles for MCL. In order to locate the racecar in a map, the other parts of MCL take the particle's pose estimations and resample the particles based on the probability of each particle's pose accuracy. If all the particles had the same pose, resampling would have no noticeable or advantageous effect.

In order to address this issue, we enabled the motion model to be run either deterministically or non-deterministically based on a parameter in the *params.yaml* file. In the deterministic case, the behavior is as explained above where all particles are updated with the

same odometry, mainly used for initial testing. In the non-deterministic case, before adding the odometry to each particle, it was augmented with added Gaussian noise. This noise was added to emulate noise and uncertainty in the real world while also allowing for particles to spread outward and give different estimates of the racecar's pose. These would later be resampled based on the probability of being accurate to the truth.

Initially, we added noise proportionally, multiplying each value of the raw odometry data by some Gaussian noise centered around 1. After visually testing in Rviz, we tried to tune the different noise parameters for each odometry value but ultimately found that proportional noise was not giving us the proper behavior. The particles would end up only spreading in the x direction if the racecar was moving in a straight line since the y value of odometry was zero. We decided to change from proportional noise to additive noise to remedy this problem and have the particles spread in all directions regardless of how the racecar was moving. We ended up adding Gaussian noise with mean and deviation values shown in ***Figure 1***. These values were obtained through slight tuning until we achieved a desirable particle spread around the racecar.

	Mean	**Standard Deviation**
X - Direction	0	0.025
Y - Direction	0	0.025
Theta - Rotation	0	0.01

Figure 1. Gaussian Noise Parameters

One overarching idea while programming the motion model was performance. Since we were calculating new poses for 200 particles each time odometry data was received, the code needed to be fast and efficient. After implementing an initial working solution we decided to reanalyze the calculations being performed on each particle and were able to refactor some parts for better speed. First, we decided to precompute the Gaussian noise being added to the odometry in an array with 1000 samples. We then only needed to read from a random index in this array in order to get the added noise instead of recalculating a Gaussian distribution for each particle. Additionally, we rewrote the main algorithm after determining the minimum calculations necessary to update each particle keeping total computational work as low as possible. We also took advantage of the numpy python package which is written in cython and allows for many computations to be done at once using list slicing. This was faster than default python list manipulation and provided us with a slight speed increase.

2.2 Sensor Model - Niko

With the completed motion model, the second main component to Monte Carlo Localization is sampling particles that are most likely to be the true pose of the racecar. In order to do this, our sensor model assigns likelihood weights to each particle that represent a good hypothesis of the racecar's pose. Therefore, the end goal of our model is to evaluate how likely each particle is given an observed sensor reading. The visualization below shows the overall function of the sensor model. For every particle in the particle cloud, we line up the laser scan beams (z) and calculate the probability of each ray compared to the ground truth (d). Since the sensor used on our racecar is a laser scanner, we chose to determine the likelihood weights utilizing four main cases: probability of detecting a known obstacle, probability of a short measurement, probability of a very large measurement, and probability of a random measurement. By modeling these cases, we account for all potential causes for a given measurement.

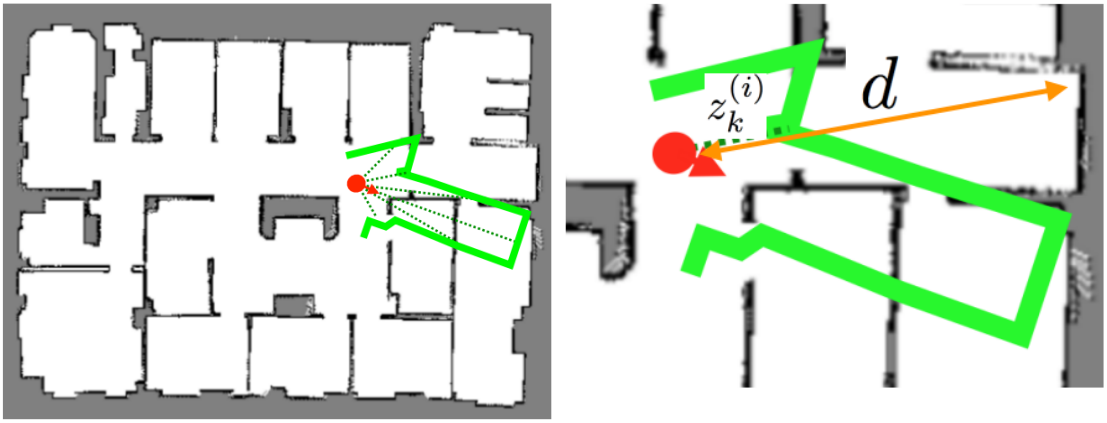

Figure 2. Laser scan visualization from lecture 12.

For the first case of detecting a known obstacle, we chose to model this with a Gaussian distribution centered on the distance between the particle pose and the closest map obstacle. We define z to be the measured range and d to be the ground truth. η is a normalization constant and $\boldsymbol{\sigma}$ is a set parameter with the value 8.0. The equation is shown below. Note that this probability is for a measured range ($z_k^{(i)}$) given a hypothetical pose (x_k) and map (m).

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

Probability formula for detecting a known obstacle.

We chose to model the second case of a short measurement with a downward sloping line. The intuition behind this choice is that objects closer to the robot are more likely to be hit by the lidar. The equation is shown below.

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

Probability formula for a short measurement.

For the third case of a very large measurement, this is modeled using a large spike at the maximum range value. The equation is shown below as p_max.

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

Probability formula for a very large measurement.

We chose to model the last case of a random measurement as a small uniform value added to all ranges. The equation is shown below.

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

Probability formula for a random measurement.

With these four distributions, we then take the weighted average with predetermined alpha values. Since it is a weighted average, the alpha values must sum up to 1. We chose the following alpha values α_{hit} is 0.74, α_{short} and α_{max} is 0.07, and α_{rand} is 0.12.

$$p(z_k^{(i)}|x_k,m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k,m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k,m) + \alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k,m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k,m)$$

Weighted average of four probability distributions.

In order to make our function efficient, we decided to precompute the model as a table of probabilities. In this table, the rows represent z values (range measurements) and the columns represent d values (ground truth values). By precomputing the values beforehand, we can directly index into the probabilities we want for a given z and d value. In order to index, we had to discretize our measurements. Specifically, we converted the measurement in meters to pixels by dividing our scans and observations by (map resolution * lidar scale to map scale). The lidar scale to map scale is a set parameter, in our case we chose 1. Finally, once we index into the table to gather probabilities, we chose to squash them by a factor of $\frac{1}{2.2}$ in order to make our distributions less peaked.

In conclusion, the sensor model takes in particles and laser scans. It discretizes both of these continuous values into pixels. With these discretized ranges, we index into our precomputed table with probabilities calculated from the four distributions mentioned above. By choosing to utilize a precomputed sensor model, we ensure a very efficient model and proper normalization for probabilities. This allows us to sample the best potential particles that represent the true pose of the racecar.

2.3 Particle Filter - Kayla

With our Motion Model and Sensor Model coded and tested, we then created our Particle Filter, which will use both Motion and Sensor Model to provide information to our car. This information is in the form of an Odometry message that tells the car an estimate of its current pose in a known environment. As mentioned earlier, having accurate knowledge about its current pose in a known environment will be important for the car to be able to plan its motion in order to reach a given goal location. This is a task that the car will be tested on for the final race challenge.

The first step in Monte Carlo localization is to initialize a set of particles. The poses of these particles will gradually change as the car moves. We chose to initialize a set of 200 particles at the starting position and orientation of the car. We made this choice for the sake of simplicity, knowing that with the addition of noise, these particles quickly expand to poses that are not equal to the pose of the car. This initialization is a design choice that we can experiment with in the future if we are seeking improved localization performance.

With the set of particles initialized, Particle Filter is then able to take in information from the car to update the poses of the particles. Whenever odometry data is received from the car, these data, along with the current poses of the particles, is fed into Motion Model. As described in the Motion Model section above, Motion Model uses the odometry data to update the poses of the particles. If the deterministic flag is set to false, Motion Model will add in Gaussian noise to the poses of the particles, as described above.

Similarly, whenever LIDAR scan data is received from the car, these data, along with the current poses of the particles, is fed into Sensor Model. As described in the Sensor Model

section above, Sensor Model calculates the probabilities of each of these particles being an accurate representation of the pose of the car. This is done by looking up the probability value for the given particle's simulated LIDAR scan distances and the car's true LIDAR scan distances in a precomputed probability table. Once these probabilities are generated, they are then normalized across all of the particles. Then, in order to select the particles that are the most accurate representations of the true pose of the car, the particles are randomly selected with replacement with each particle weighted by its probability value. The result is a new set of 200 particles that have higher probabilities of being accurate representations of the car.

Anytime the set of particles is updated, either through Motion Model with odometry data or through Sensor Model with LIDAR scan data, the average pose of these particles is calculated. To calculate the average orientation angle theta of the particles, we used the circular mean formula:

$$\operatorname{atan2}\left(\sum_{j=1}^{n} \sin \alpha_j, \sum_{j=1}^{n} \cos \alpha_j\right)$$

Circular Mean formula

This formula converts the theta values into points on the unit circle in Cartesian coordinates, takes the arithmetic mean of these points, and then converts this point back into polar coordinates to get the mean theta value of the particles. The average x and y coordinates of the particles is calculated by taking a simple arithmetic mean of all of the x and y values. We publish this pose estimate as a transform from the world frame to the lidar frame of the car, allowing us to also see the car's pose with respect to the intermediate base link frame. Using transforms are compact and useful as the distance from the lidar to the base axle is a parameter that differs between the simulator car and the physical car.

In conclusion, the Particle Filter uses received odometry and LIDAR scan data to update the set of representative particles and then calculates the average pose of these particles in order to provide the car with an accurate estimate of its own pose.

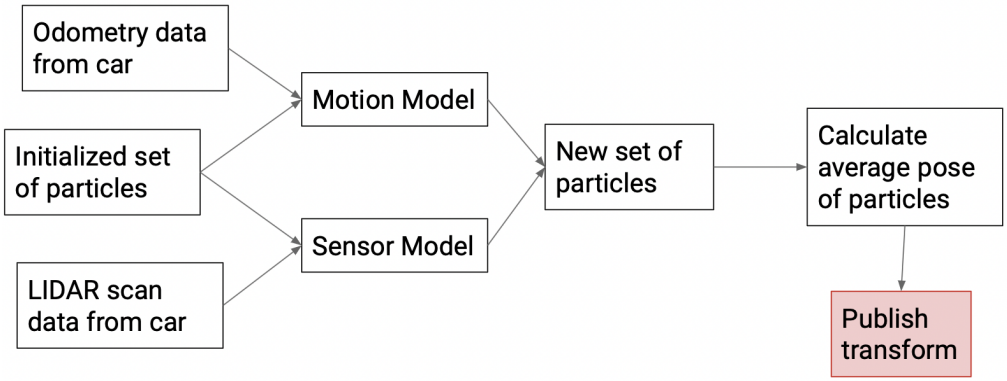

Figure 3. High level overview of the steps of Particle Filter.

3 Evaluation - Jared

In order to evaluate the performance of our MCL implementation, we conducted a few tests in both simulation and the physical world. Our general approach was to conduct quantitative testing in simulation, using the racecar's performance results to improve and iterate on our MCL implementation, and then conduct a suite of qualitative tests in the physical world to verify our racecar's performance. Our reasoning for this centered on the simulation's ability to provide accessible quantitative results, such as ground truth pose of the robot. For example, one of our primary heuristics for evaluating the racecar's performance was the error of our pose estimate compared to ground truth. In simulation, we can easily see what the ground truth odometry is and simply record a rosbag to evaluate our estimate, but in real life, we don't have the ability to measure the ground truth. In practice, this can typically be found with the aid of external sensors and motion capture systems.

Throughout the testing and evaluation performance of this lab, one of the primary sections of the algorithm we focused on was how we generated noise in the motion model. Given our success in the provided sensor model and motion model unit tests, we could confidently say that our sensor model and *deterministic* motion model worked to an acceptable level in both speed and accuracy, but we still had to test our non-deterministic noise generation. The goal for a successful non-deterministic motion model is to generate particles in a wide enough spread such that it covers all of the robot's potential poses, but in a small enough spread such that the estimator doesn't encounter any stability issues. To debug the noise generation, visualizing the point cloud spread in Rviz allowed us to get a full sense of how the particles were propagating due to added noise.

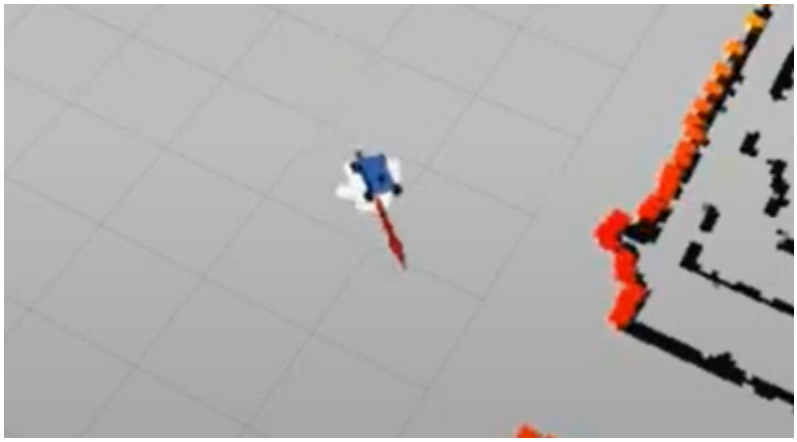

Figure 4. Typical Point Cloud Configuration

Our initial approach for generating noise was to create odometry-proportional noise generation, which would ideally propagate greater noise in the directions of greater movement. However, despite a few iterations of different noise distributions, our particle generation greatly under-represented noise in the y-directions and theta-directions, leaving a biased particle cloud that did not cover the robot's actual location. Figure 5 shows this phenomenon.

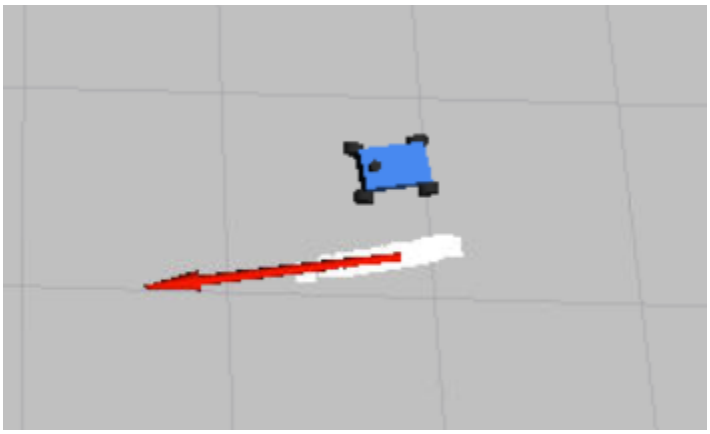

Figure 5. Proportional Noise Bias

As a result, we decided that pursuing a simpler noise generation method would be more effective, and switched to adding regular gaussian noise. After a few iterations of tuning the spread of the noise, we arrived at a motion model that propagated particles in a representative manner and no longer suffered from the proportional bias.

Once the individual parts of the particle filter were debugged, the next step was to test the system as a whole. One of the most useful preliminary evaluation tools we had access to is the gradescope unit test, which provided information on our racecar's estimation error in a few

set test cases. We found these useful because the test cases were repeatable, and it allowed us to compare our performance to the staff-approved performance threshold. After multiple iterations of using the gradescope unit tests and adjusting our noise model, we arrived at the following results. Figure 6 shows the path of our position estimate compared to the ground truth and the staff solution. The left graph indicates a scenario with low odometry noise and the right graph indicates a scenario with high odometry noise.

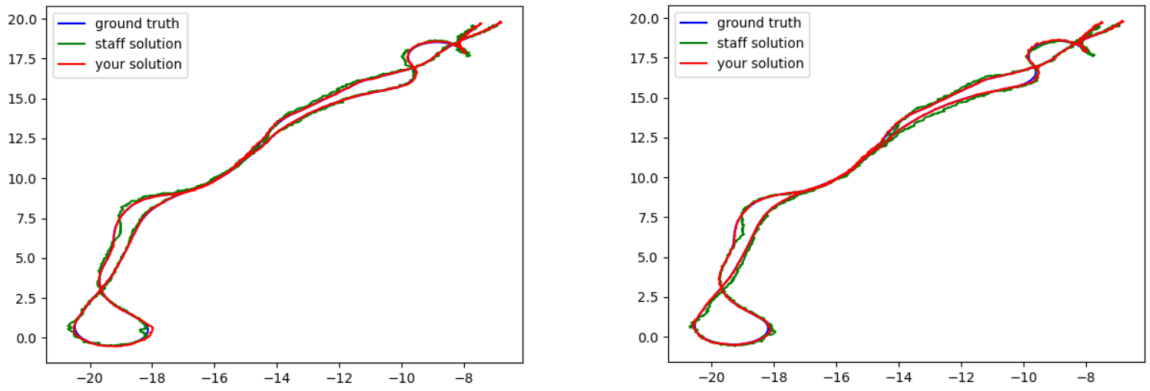

Figure 6. Low Odometry Noise and High Odometry Noise Test Tracking

As you can see from Figure 6, both test cases yielded similar performance, where our pose estimate accurately tracked the ground truth. Our solution's time average deviation from the ground truth was approximately 4 cm across all tests. Compared to the staff solution's average error, our estimate was on average five times closer to the ground truth.

We also conducted a few of our own tests in simulation to get a better grasp on the quantitative performance of our racecar. By calculating the estimator's ground truth deviation ourselves, we were able to plot more illustrative visualizations of our racecar's performance. Figure X. shows an example simulation test case alongside the plotted error.

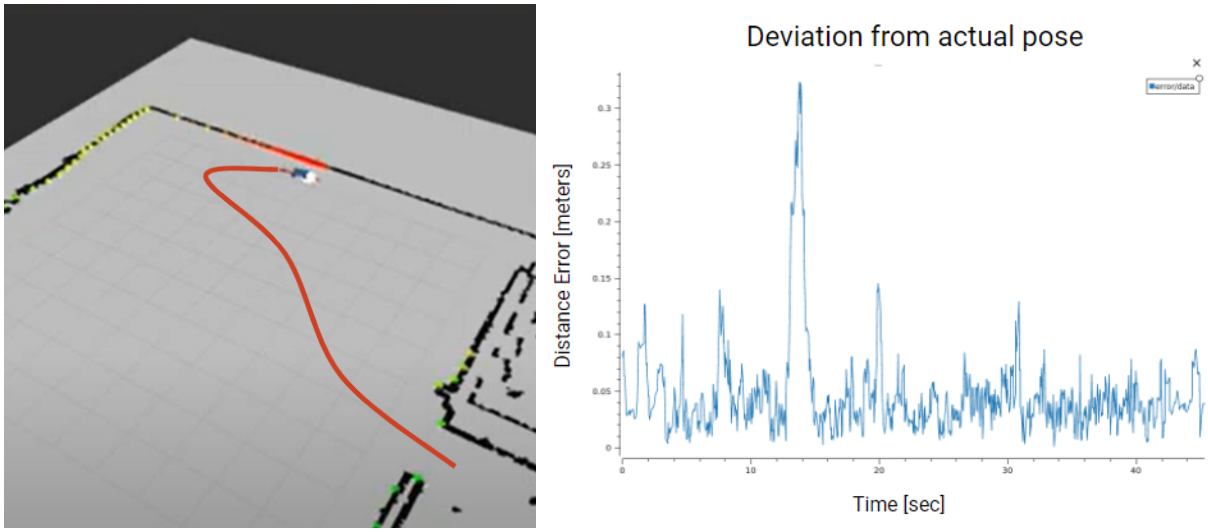

Figure 7. Racecar Simulation Path and Estimator's Deviation from Ground Truth

In this test case, we're able to see more clearly how the ground truth deviation evolves over time. Consistent with the gradescope results, the time-average deviation is around 0.04 meters, although there are some interesting spikes and inconsistencies in this error. Regarding the spike at 14 seconds, this is when the simulated racecar took a sharp turn, turning toward the empty portion of the room. We speculate that the spike in error is due to the rapid change in heading in combination with far away, noisy LIDAR scans from the other end of the room.

One example of how visualizing this deviation over time was helpful was to identify potential steady state errors in our system. During one of the first simulation tests we conducted, the error over time was bounded from below by a value of 30 centimeters, and never dipped below it. This contrasted our expectation, which was that the error measurement would be non-uniform, dipping below the average just as often as it rose above the average. This was a hint that allowed us to identify a critical error in our system - that we were expressing the robot's pose with respect to the wrong frame. The particle filter effectively predicts where the LIDAR sensor is in space, but we are interested in estimating where the robot's base link is in space, meaning we have to transform our estimate between coordinate frames. Doing this solved our issue and the time-average deviation from ground truth was cut by around 90 percent.

After conducting a suite of in-simulation tests, evaluating the estimator's performance and iterating on our MCL implementation, the next goal was to test the racecar's performance in the physical world. As previously mentioned, we relied primarily on qualitative results to verify if our robot performed satisfactorily, since the robot doesn't have access to ground truth information in the real world.

The following figure illustrates one of the tests we conducted in the Stata basement,

where we drove the robot around and visualized the laser scan, particle cloud, pose estimate, and environment map in Rviz. As the robot drives around, it publishes each subsequent laserscan in the reference frame of the most recent pose estimate. This means that if MCL is estimating the robot's pose accurately, the laser scan should line up with the environment map in Rviz. Our implementation of MCL quickly and accurately lined up the laser scan and the map boundary, as shown in Figure 8.

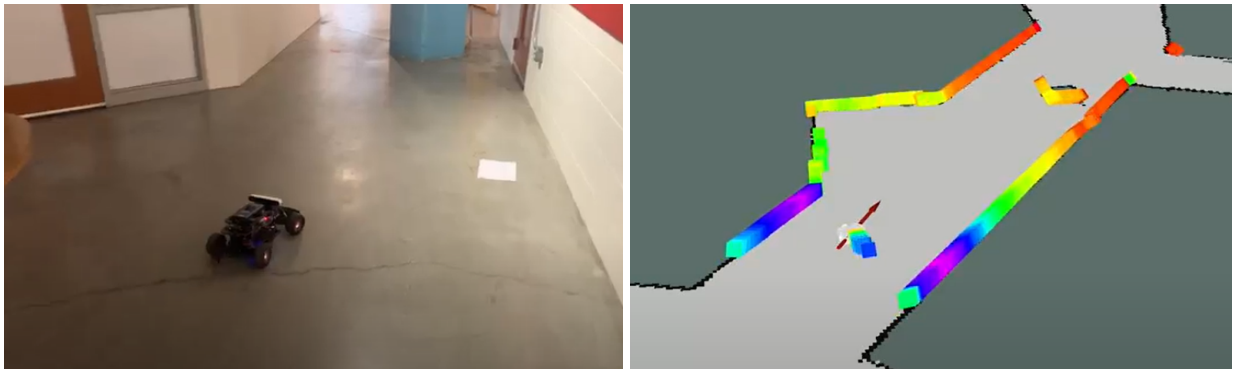

Figure 8. Stata Basement Localization Testing

4 Conclusion - Amy

Overall, we successfully built a localization module that estimates the racecar's pose with minor but acceptable errors. When tested in both simulation and in the physical world, the racecar accurately detects its position with an average error of less than 0.05 meter from the ground truth position. This localization capability we implemented is a crucial foundation for future navigational abilities such as motion planning, where knowing the exact pose of the racecar is essential.

While we are happy with the result, there are a few improvements we could potentially make in the future. One of the most flexible components of our implementation lies in the noise distribution we apply in the motion model, where small variations can greatly affect the accuracy of the final pose estimate. Even though our current choice of Gaussian noise produces desirable results, we believe that by tweaking and testing different parameters that contribute to applying the noises, we can achieve a better end result with smaller overall error. In addition, our initialization of the particles places all particles at the exact position the racecar starts at. By experimenting with different initialization approaches, such as placing the particles in a random distribution, we can perhaps identify a more logical or a better performing localization module.

With a proper localization module in place, our team can now look to implement more complex navigational abilities on the racecar. Keeping these possible improvements in mind, we

hope to build a racecar that excels not just in the final challenge, but also as an effective autonomous robot.

5 Lessons Learned - Everyone

5.1 Kayla

Technical: This lab really highlighted that efficient code is crucial for successful localization of the car. This is because the odometry and LIDAR scan data is being received from the car at a very quick rate (50 Hz) and our localization algorithm needs to be able to keep up with this quickly incoming data to update the set of particles with the most recently received data. If our code takes too long to run, then the particles will be updated using data that is many timestamps behind and the result will be an inaccurate estimate of the car's pose. We made a variety of changes in order to make our code run faster, including replacing for loops and Python lists with numpy functions and arrays and precomputing computationally intensive calculations.

Communication: For our briefing this week, I didn't write out a script of what I planned to say for my part of the presentation like I have for the previous briefings. I found that this actually helped me because I wasn't focused on remembering exactly what I had planned out to say, but instead was able to relay the points that I wanted to get across to the audience with a clearer mind. I think I will continue to use this strategy for future briefings.

5.2 Amy

Technical: The two lessons the team learned over the course of this lab was the importance of code efficiency and the difficulty around code integration. The nature of our implementation requires the motion model and sensor model to run at a high rate inorder for the particle filter to find the most accurate pose estimate. When debugging the particle filter, the team spent a lot of time making sure the two modules are running as fast as they can, and we will hopefully carry this experience into future labs as well as the final challenge. In addition, while individual modules did not take a long time to finish, trying to fit all the pieces together consumed a majority of our time. In the future, I will definitely pay more attention to understand each portion of the code earlier and more thoroughly to speed up the overall process.

Communication: As mentioned in the technical reflection, the integration of individual modules was perhaps the most challenging and time consuming part of this lab. One strategy we had to adopt was handing off unfinished code implementation/sections to each other since we each had limited time to work on the lab. Throughout this experience, I realized how important it is to clearly communicate my current strategy and thought process to whoever I am handing the code to. Being able to understand everyone else's work as well as giving them a summary of my contributions will definitely be a crucial skill to learn for the final challenge.

5.3 Jared

Technical: One of the common themes for this lab was building fast and efficient code, often pre-computing tables and trying to speed up matrix operations. I quickly learned some of my habits that produced slow code and learned from some of my peers how to create faster computation.

Communication: One of the major challenges of this lab was having to communicate and coordinate work through spring break and greatly differing schedules. Things were slightly less organized and a little more last-minute, and as a result I don't think we were as efficient as previous labs. In the future, I think spending more time having discussions on how we can delegate and divide up the workload at all stages of the project would prove to be useful.

5.4 Niko

Technical: This lab highlights the value of code efficiency in order to ensure accuracy for localization. Specifically, I learned how quick and efficient numpy is in relation to regular python loops. Numpy enables quick matrix manipulation and operations which is significantly faster compared to python. This is especially valuable for tasks such as localization which requires many calls and updates to be accurate.

Communication: My communication takeaway from this week is the value of remote work and conversations. This lab was over spring break and I got covid for the last part of the lab. As a result, it was extremely difficult to organize work without meeting in person. I learned that holding zoom meetings allows for more efficient and concrete distribution of work than group chat messages.

5.5 Daven

Technical: Since performance was more important in this lab than others, I was forced to learn and use numpy. While it took a while to wrap my head around, once I understood how numpy represents vectors and matrices it made translating the math to code easier. I also was able to see the speed increase upon refactoring code to use numpy and plan on using this package more in the future.

Communication: Due to spring break splitting the two weeks of this lab, it was difficult finding the time to work together since everyone had a lot to do that week back. We were able to pick a day on which we were all available and took shifts working picking up where the last members left off. This was a different way to work on things then I have in the past, but ended up

successful and may become something I suggest in the future. It allows for members with different schedules while also maintaining momentum on the project.