Lab 5 Report: Localization

Team 12

Angel Gomez
Ariel Fuchs
Gokul Kolady
Sharmi Shah

6.141

April 5, 2022

1 Introduction (Author: Ariel, Editor:)

In this lab our goal was to create an algorithm that would allow our robot to be randomly oriented within a map, and still be able to correctly identify its location and direction. We had three main stages to reach this goal. The first was creating a Motion Model. The second was to create a Sensor Model. The third was combining both to create a Particle Filter. These steps allowed us to create our final Monte Carlo Localization algorithm which achieved our lab goal.

Both the Motion Model and the Sensor Model were imperative to understand to succeed in this lab. The motion model uses odometry to determine the position of an object. The Sensor Model analyzes a group of data points and chooses the most relevant ones to continue sampling from. After getting this foundation down we were able to combine both the Motion Model and Sensor Model to create our particle filter, which allowed our robot to determine where it likely was in a given map. The Monte Carlo Localization algorithm allows for robots to localize using a particle filter.

After our Monte Carlo Algorithm was created, we needed to test our code both in simulation and physically. We used RVIZ to test our code in simulations. We tested our robot in the tunnels under STATA to see how it worked in real life. We ran our wall follower code and line follower code. Then we would ROSBAG our real life tests and simulate it in RVIZ. Since we uploaded the map of the STATA tunnels and replayed our robot runs from the correct locations, we expected to see a beautifully mapped path similar to real life. However, we had many difficulties throughout the lab, such as debugging, fixing our visualizations, matching the robot path to the RVIZ simulated map and so forth. We went back and forth on fixing our randomized error value ranges as well, making sure we were listening to the right topics, started at the correct pose, and laying with other features in our code to counter problems we faced. Ultimately we were able to resolve these challenges, and we will elaborate on what those challenges were and how we overcame them throughout this lab report. This lab was a big milestone for our team as it sets us up strongly to succeed in building an autonomous vehicle that can navigate on its own!

2 Technical Approach (Author: Gokul, Editor: Ariel)

The technical goal of this lab was to create a particle filter that is able to accurately determine the location of our robot in a known map at any given time. Essentially, the implemented algorithm maintains a cloud of particle locations on the map that represent hypotheses for where the robot could be, and using Monte Carlo Localization, continually re-samples the particles that represent the most likely locations in order to maintain a high-probability particle cloud, updates the positions of these particles over time based on the robot's movement, and averages the hypotheses in order to guess the robot's location at a given time-step.

This algorithm relies upon two crucial sub-components. The first is a motion model that updates the locations of a current set of hypothesis particles given the racecar's odometry (this allows the particle cloud to generally follow the robot's movement). The second is a sensor model that re-samples the most probable particle locations by comparing each of their poses relative to the map (a hypothetical laser scan) to the robot's actual laser scan. Finally, our overall particle filter module continually updates its particle cloud and location guess by calling upon these sub-components whenever their relevant input data arrives (odometry information and laser scans).

2.1 Motion Model (Author: Gokul, Editor: Ariel)

The first important sub-component of the particle filter was the motion model. The core motivation behind this model is to move the set of hypothesis particles along with the robot as it traverses the map. However, because the robot's actual location in the map is unknown (the goal of this lab is to triangulate it), the only information available to keep the particle cloud moving with the robot is the robot's odometry (a description of it's x, y, and rotational change at any given time).

However, there is one issue on purely relying on odometry: Odometry can be noisy. Thus, if the particle locations were updated based on raw odometry, the odometry error during each reading would compile over larger amounts of time to steer the particle cloud far away from the robot. Thus, we add random noise to the odometry that is used to update each particle's location in order to spread the particle cloud out over time and have a wider distribution of particle hypotheses to choose from (this makes it more likely that one of them will actually reflect the robot's real pose).

The following was the update calculation performed for each particle p in the motion model ROS implementation:

$$P_p^{World} = P_p^{World} + R_p^{World} \cdot (O_{Robot}^{World} + \epsilon) \tag{1}$$

where P represents a 3-length pose vector, R represents a rotation matrix, O represents a 3-length odometry vector, and ϵ represents a 3-length random noise vector with each element ranging from -0.01 to 0.01.

2.2 Sensor Model (Author: Sharmi, Editor: Gokul)

The second major aspect of the particle filter was the sensor model. Generally, the sensor model uses extroceptive measurements to define how likely it is to record a given sensor recording z_k given a hypothesis position x_k, in a known static map m at time k. As the motion model spreads out the particles, the sensor model uses its insight about the outside environment to pick the particles that are most consistent with the sensor reading from the robot. The probability distribution that defines how the sensor model "picks" particles is dependent on the sensor.

2.2.1 Pre-computing the Probability Distribution

For this case, the sensor in use is a laser scanner. The likelihood of a scan $p(z_k|x_k, m)$, is computed as the product of all the likelihoods of each of n measurements in the scan. For each range measurement i, there are different cases to be considered that when put together define $p(z_k^{(i)}|x_k, m)$, which is the likelihood of measurement i in the scan:

1. Probability of detecting a known obstacle in the map. This is represented with a Gaussian distribution centered around the ground truth distance between the hypothesis pose and the expected nearest map obstacle. If the measured range is $z_k^{(i)}$, the ground truth range is d, and η is a normalization constant that ensures the probabilities sum to 1, we can define this probability as:

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

2. Probability of a short measurement, potentially due to internal lidar reflections or hitting parts of the robot or unknown obstacles that are nearby. This is represented as a downward sloping line because the lidar is more likely to hit obstacles that are closer. In other words, since the area that an object will take up in the lidar range scales down with increasing distance from the scanner, the probability of hitting that obstacle decreases with increasing distance. This probability is defined as:

$$p_{short}(z_k^{(i)}|x_k, m) = \frac{2}{d} \begin{cases} 1 - \frac{(z_k^{(i)}}{d} & 0 \leq z_k \leq d, d \neq 0 \\ 0 & otherwise \end{cases} \tag{3}$$

3. Probability of a very large measurement. Usually when the measurement is very large, it means the lidar beam hit an object and did not bounce back to the sensor. This is represented by a large spike in probability at the maximum range value so that some very large measurement does not significantly alter the particle weights. This essentially is a delta function at the max range which can be approximated by a narrow uniform distribution at the maximum range, z_{max}, with small ϵ. This probability is therefore defined as:

$$p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{\epsilon} & -\epsilon \leq z_k^{(i)} \leq z_{max} \\ 0 & otherwise \end{cases} \tag{4}$$

4. Probability of a completely random measurement. This is represented by a small uniform value to account for some random unforseen events. It can be defined as:

$$p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}} & 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & otherwise \end{cases} \tag{5}$$

Each of these 4 probability distributions are combined by a weighted average depending on how much each case affects the measurements. If the weights are defined as α_{hit}, α_{short}, α_{max}, α_{rand}, we get:

$$p(z_k^{(i)}|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m) + \alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m) \tag{6}$$

This probability distribution when put together looks like this:

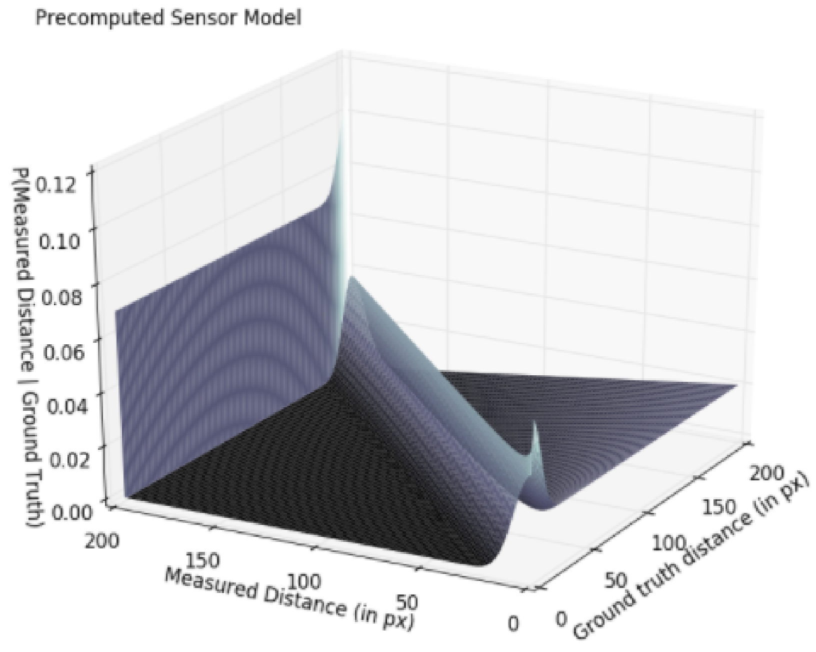

Figure 1: Probability distribution of the sensor model using values $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, $\alpha_{rand} = 0.12$, and $\sigma_{hit} = 8.0$

A z_{max} of 200 was used here and a 201 by 201 array was defined to store the probability distribution with the measured distance, z along the row axis and the ground truth distance along the column axis. This enabled future computations to be more efficient as one could simply index into this array given a z_k and an x_k.

Another attribute of the sensor model is the "squashing" parameter, which is essentially raising the probability distribution to a power less than 1. This parameter makes the distribution less peaked. For simulation purposes a value of $\frac{1}{2.2}$ was used. However, after testing values of $\frac{1}{1.1}$, $\frac{1}{2.2}$, and $\frac{1}{3}$, the first value was seen to work best when the particle filter was implemented on the robot.

2.2.2 Ray Casting

The list of particles that are given to the sensor model are inputted as an nx3 matrix, where each particle has pose $[x, y, \theta]$. The particles are passed through a ray caster which returns a stack of lidar messages, one for each particle. This produces a laser scan from the point of view from each of the particles. The laser scanner has a little over 1000 beams, which we downsampled to about 100 beams for more efficient computation purposes.

2.2.3 Evaluating Model

The observations and particle laser scans were pre-processed to ensure that they only ranged from 0 to $z_{max} = 200$. The precomputed probability distribution array and a list of particles with their own laser scans were used to compare the scan of each particle to the scan of an observation. If the scan from a particle was very similar to that of the observation, it was assigned a high probability. Next, the particles were resampled from the discrete probability distribution with weights equal to the probability of each particle. This allowed particles with higher probabilities to be selected much more and particles with lower probabilities to get discarded.

2.3 Scaling

The laser scans that the racecar obtains are not necessarily on the same scale as the scans from the particles. In order to tune the scaling factor, the lidar scan from the robot was visualized and the scaling was manually changed until it matched the map in Rviz.

2.4 Particle Filter (Author: Gokul, Editor: Ariel)

Once the motion and sensor models had been implemented, we moved on to the higher-level particle filter code. This code was responsible for subscribing to the right input topics, publishing to the appropriate output topics, and utilizing the aforementioned models to convert inputs like odometry and laser scans into real-time particle locations. There were five primary functions used within our particle filter class that allowed us to perform robot pose estimation.

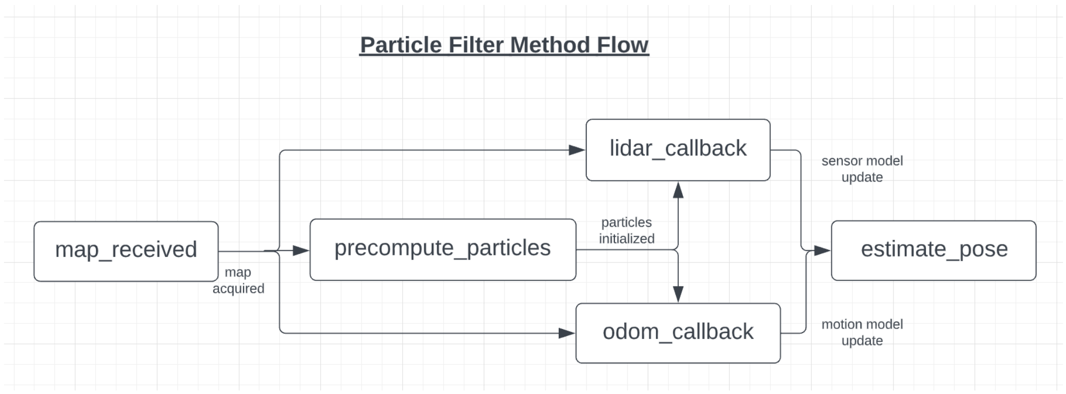

Figure 2: Shown here is the general flow of the methods used in our particle filter code. These methods are described in detail below.

2.4.1 map_received

Before this function was added to our implementation, there were errors produced immediately after launching the particle filter algorithm. These errors were a result of the short downtime that existed between the particle filter being launched and the reference map itself being initialized. The algorithm would try to start triangulating the robot's pose without having a map to reference, resulting in nonsensical computations. In order to tackle this, we implemented a simple method that set a "map_acquired" flag to True once information was being received on the map topic. The remaining functions all check for this flag to be True before executing their operations.

2.4.2 precompute_particles

This method is responsible for initializing the particle cloud. Whenever a message is received on the /initialpose topic (the message being an estimate of the robot's starting pose), a pre-determined number of particles are initialized as a matrix of particle poses. Each particle's pose is set to be the robot's pose with random noise ranging from -0.01 to 0.01 added to its x, y, and θ values.

2.4.3 lidar_callback

Whenever a lidar scan is received, it is down-sampled by taking 100 equally spaced out scans from the full scan range, then it is fed into the sensor model along with the current matrix of particles. The sensor model returns a probability distribution over the particles. This distribution is then normalized and called upon in order to randomly sample a new set of particles (those with higher likelihoods of being accurate are thus more likely to populate the new set of particles).

2.4.4 odom_callback

Whenever an odometry message is received, its x, y, and θ values are extracted from the message's twist component. These values are then each multiplied by dt, the amount of time that has passed since the last odometry message was received (as we want to scale the particle motion updates based on the amount of time that has passed since the last update). Finally, this scaled odometry vector is passed into the motion model to update particle positions.

2.4.5 estimate_pose

Once particles have been updated via either the motion or sensor model, this function is called in order to publish a pose estimate to pf/pose/odom and a transformation for the base_link_pf frame (particle filter's estimation of the robot base link) with respect to the map frame. Both of these values are drawn from the same pose estimate. The pose estimate is calculated by averaging the x values of all particles, averaging the y values of all particles, and averaging the θ values of all particles non-traditionally using the following equation:

$$\theta_{pose} = \arctan \frac{\sum_{i \in particles} \sin \theta_i}{\sum_{i \in particles} \cos \theta_i} \tag{7}$$

We decided to use this form of averaging due to the fact that when averaging these θ values, the averaging is being performed on circular rather than linear values. Thus, a traditional average would not reflect the spacial properties that these theta values possess. Finally, all particles are visualized by feeding their poses into a pose array message and publishing it.

3 Experimental Evaluation (Author: Angel, Editor: Gokul)

3.1 RVIZ Visualization

After setting up the motion model and the sensor model, and combining them into the particle filter, the next step was to test these implementations within a simulated environment using Rviz. To do so, we visualized the odometry arrow created by our particle filter in the /pf/pose/odom topic. We also created a new publisher in which we published each of the individual particles' states in the visualization, in order to qualitatively verify that our model was working and that the position of the /pf/pose/odom arrow made sense. The result of these functionalities can be seen below in Figure 3.

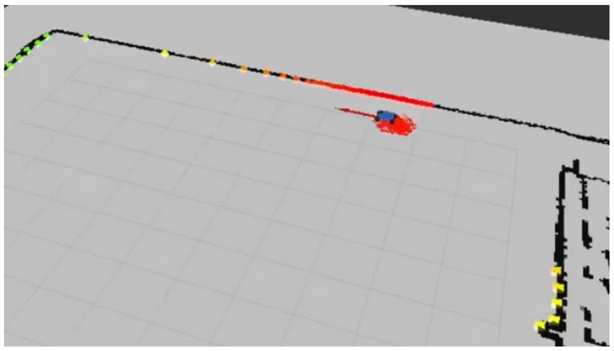

Figure 3: Rviz visualization of /pf/pose/odom arrow and individual particle states.

3.2 Particle Filter in Simulation

The first goal we had for our Particle Filter was to demonstrate the simulated success of the pose and orientation determination of the robot in a simulated environment. This was done in order to have both a quantitative and a qualitative proof-of-concept that showed our Localization Structure was able to find the robot's pose in a given map frame. In the following sections, we'll delve into more detail on the performance.

3.2.1 Testing Plan

A few trials were performed to evaluate the success of the particle filter as seen in the table below. These trials had the difference of containing different amounts of added noise to the respective x,y, and theta portions of each particle's odometry. For these trials, we needed an odometry input for the racecar to actually move, and thus we settled on using the wall follower code we developed for Lab 3 in the Stata Basement map provided to us in the racecar_simulator package. Also, given that we were in a simulated environment and we could assume that no deviations from ground truth would occur from just driving the robot, we used the /odom topic of the robot as the benchmark for ground truth in our comparisons with the /pf/pose/odom arrow we got from our MCL implementation. If the robot was able to perform at desired standards in this simulated environment, then theoretically it could run well within the actual racecar platform in real time.

Table 1: List of tests performed for simulated Localization

Test	Noise in X	Noise in Y	Noise in θ
1	0	0	0
2	0.3	0.3	0.3
3	0.6	0.6	0.6

3.2.2 Performance Metrics

The specification for this particle filter is to accurately predict the pose of the robot. There are two performance metrics that we utilized to evaluate this efficacy of our MCL implementation. The first of these is a qualitative comparison between the path that the /pf/pose/odom arrow took (our predicted robot pose). The second, and main performance metric, was cross-track error, which is defined as the distance between our predicted pose and the actual pose of the robot, or more explicitly:

$$\text{CT-Error} = \sqrt{(x_{estimated} - x_{actual})^2 + (y_{estimated} - y_{actual})^2}$$

3.2.3 Testing

All of the trials where ran in the region shown below in the Stata Basement map; just in front of 32-044 at different startign locations/paths.

Figure 4: Region where simulations where ran

The first trial shown below is that where no noise was added to the odometry whatsoever. Below we see in Figure 5 that the racecar, from its (-6,10) starting point, was able to quickly correct itself and track the pose of the car at an average error of 0.224 m, as seen from Figure 6.

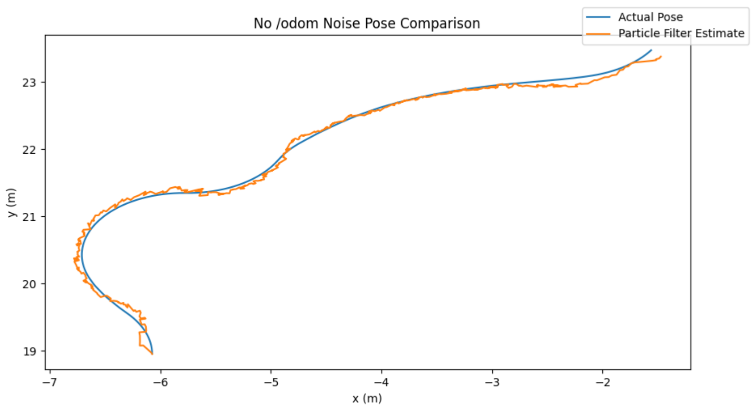

Figure 5: Pose Comparison for No Added Noise

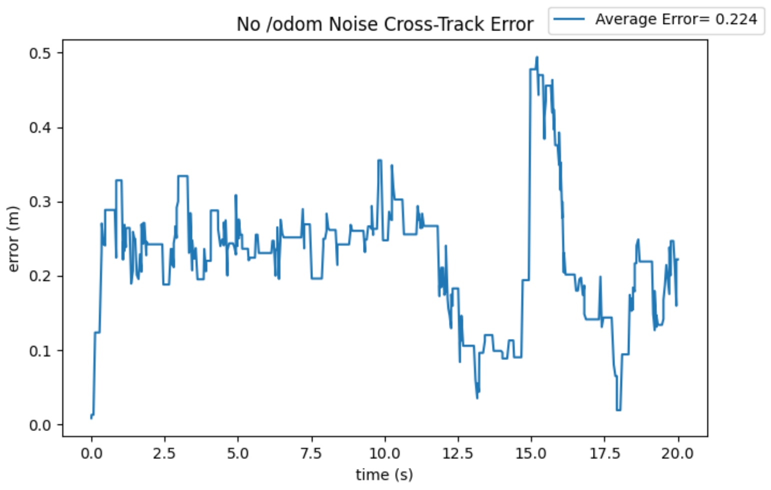

Figure 6: Cross-Track Error for No Added Noise

For the second trial, where we added a 0.3 scaled noise on all sections, the results can be seen below. We again see, from Figure 7, that the robot was also able to quickly correct itself from its (-6,20) starting point, all the way until the end, where it got less accurate due to it encountering the door area (left of 32-044 entrance). From Figure 8, we can more quantitatively see its behavior, and get an average error of 0.183 m, falling within margins. This lower value could be due to the fact that the path this trial took consisted of less sharp turning.

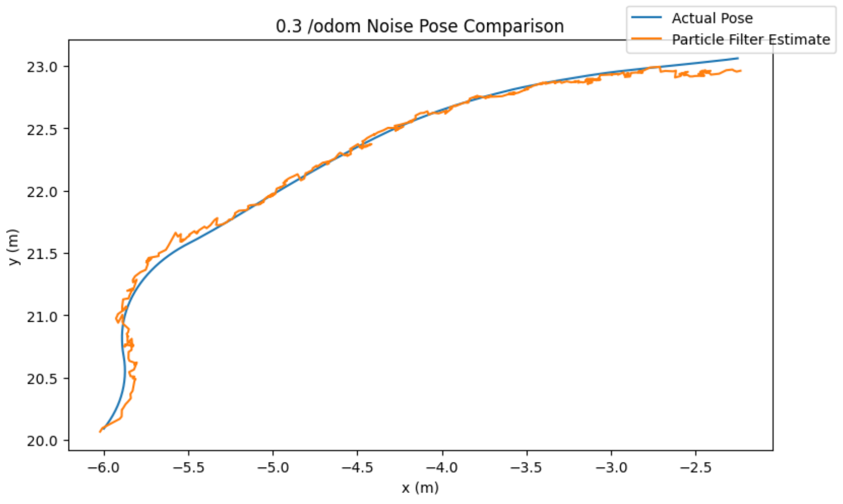

Figure 7: Pose Comparison for 0.3 Added Noise

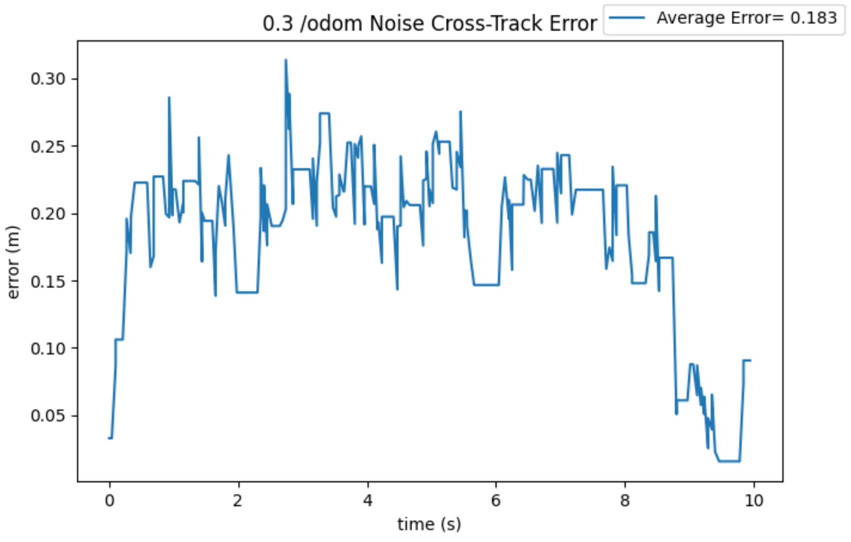

Figure 8: Cross-Track Error for 0.3 Added Noise

Finally, we can see the results for trial 3 where we added a 0.6 scaled noise on all sections. We once again see, from Figure 9, the robot quickly correct itself from its (-6,19) starting point, albeit in a more noisy way. This noisy-ness can also be seen in Figure 10, where we get that the average error is 0.261 m. However, this still falls within acceptable bounds, so we consider it a success.

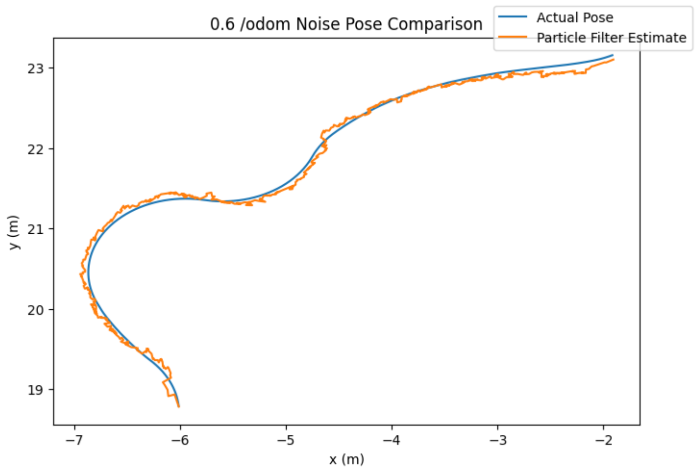

Figure 9: Pose Comparison for 0.6 Added Noise

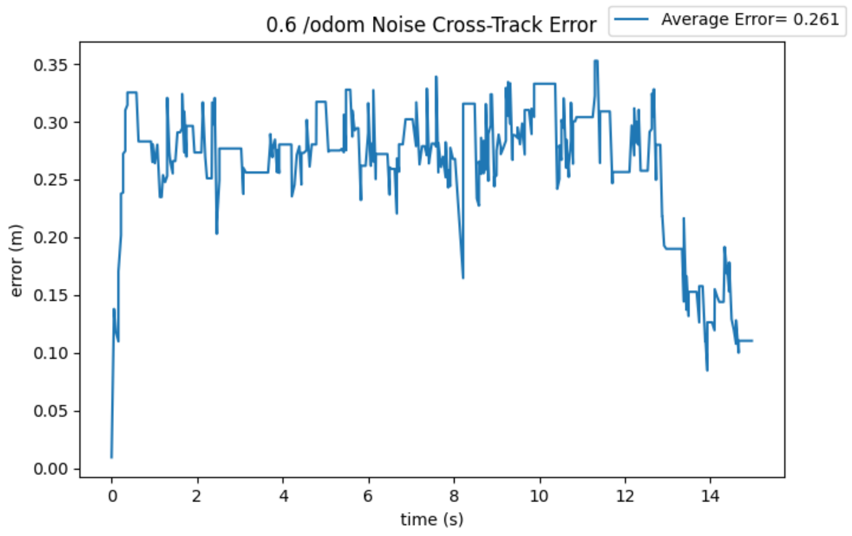

Figure 10: Cross-Track Error for 0.6 Added Noise

3.2.4 Performance Assessment

Overall, the MCL had a great performance and met the specification described. It was always able to predict the robots correct pose within error. With this, we can move onto implementing it into our racecar.

3.3 Particle Filter in Racecar

After obtaining these results within the simulated environment, it was time to proceed to test the implementation in the racecar platform itself in the Stata basement. Given the sensitivity our final particle filter had to noise, we had to toy around with the tuning of this noise before we began driving it around and outputting data. With a change of topics and some debugging, we proceeded to drive our car around.

3.3.1 Testing Plan Metrics

A few trials were performed to evaluate the success of our particle filter in the actual robot, as we'll see in the following section. Since we were having problems with outputting the data in real-time from Rviz, we proceeded to rosbag our driving commands from teleop and proceeded to replay them within our local machine. Given that there is no topic for a ground truth in the actual robot (as if there was, we wouldn't need a MCL to determine the robot's location), our analysis was based qualitatively on the comparison of the data shown in Rviz and the path we saw the robot take within the Stata basement.

3.3.2 Testing Performance Assesment

Below, we see the results:

$Trial1$

$Trial2$

From these two trials, we can see how the robot seems to be able to accurately enough determine its pose within the Stata basement map, and as such meets our needs for localization. Some of the errors seen could be due to errors in our /initalpose placement, as this was all estimation on the orientation/position of the robot within the basement. We can also see some misalignment from the scan, and that could be due to some errors in the choice of frame. Further work could be done in fixing this and in verifying that our localization works in real time.

4 Conclusion (Author: Ariel, Editor: Angel)

Lab 5: Localization was definitely challenging, but made us understand every component thoroughly, and ultimately we ended up reaching our goals for the lab. We were able to create a Motion Model, a Sensor Model, a Particle Filter, which lead into our Monte Carlo Algorithm. We were able to test these both in simulation, RVIZ, and in real life.

Our Monte Carlo Algorithm allowed the robot to determine its pose within a map. This works by filling the given map with randomized particles and orientation. With every timestep we would compare the car LIDAR scan to the particles that we had randomized. We would weight the particles that showed the most similar view to the LIDAR scan, remove particles that were not relevant, and then would resample. After repeated iterations of this process, we were able to correctly locate our robots pose with strong accuracy.

This lab has set us up well to create an autonomous vehicle. Our robot can follow lines, walls, stop in front of objects based on distance or color, identify objects, and now orient itself in a map just by driving around. We faced many challenging problems this lab, with creating the path that the robot drove in real life on the simulation RVIZ. We realized that our problem was because we were adding noise to the world frame in our motion model instead of the robot frame. We also had difficulty tuning the noise but after multiple trails we determined the best range was between $[-0.5, 0.5]$. Lastly, we had difficulty lining up the simulated map to scale with the actual robot map. We were able to fix this by frequently using visualization tools and mapping LIDAR scans, as well as trial and error with the scaling parameter. With that being said, we ended up being very successful and our robot can determine where it is on a map while it is navigating safely.

5 Lessons Learned

Ariel: Before this lab I never thought about the different ways robots had to use localization to figure out where they were in a given environment. It took me many reviews of the lecture, lab, and Youtube videos to really understand how the Monte Carlo Localization works, but now that I do understand it I can really appreciate how clever it is. I'm impressed with how random guesses of the robots location can narrow down to a very solid prediction of its pose and how it has moved. I am very excited to see that our robot is getting closer to a stage where it can move autonomously.

Angel: This lab, like for others, really gave me a sense on how important it is to understand the big picture of a task and the end goals of said task before trying to create the components that together will build said task. Over the course of this lab, we faced many debugging challenges and many times, for me, I focused too much on trying to fix an error that I thought was on one component of the system where in fact it lied in another. This, alongside with the countless hours we spent testing, really gave me a better appreciation and understanding of the processes that go into creating such a complex system like that of an autonomous vehicle, and I'm very exited to see how our robot keeps evolving from here on out.

Gokul: On the technical side of things, this lab taught me how important it is to understand the complete picture of an algorithm before attempting to implement its components. There were often moments where I would begin to dive into coding without a clear and concrete understanding of what the overall goal of my work was, which resulted in more debugging in the later portions of the lab. Not only does having a fundamental grasp over technical concepts prevent errors from being made while coding, but it makes the debugging process much more focused and intentional. In terms of collaboration, I learned how important it is to make sure every team member understands every component of a project conceptually, even if they may not necessarily be working on all of them. This allows components to come together very cleanly when the time comes because everyone is aware of how their piece fits into the larger puzzle. During this lab, I felt that by the second half we all had a solid understanding of what was happening due to frequent communication, which gave every team member the freedom to iterate and debug in parallel and individually, resulting in efficient execution.

Sharmi: For this lab I realized that it was essential for me to understand the big picture and how the separate pieces of the puzzle, the motion model and sensor model, work together to create the particle filter. This broad technical understand allowed me to implement the particle filter. This lab also helped me understand the technical aspect of how localization works and why the particle filter method is a good way of going about localization. I was able to get a better understanding of debugging tools during this lab as well because so many parts of the lab did not go as expected. In particular, visualizing as many of the involved components as possible is very beneficial. For example, visualizing the particles as well as the average pose estimate gave an indication of which particles were being resampled and what adjustments should be made to the algorithm in order to generate a better pose estimate. Similarly, while debugging the simulated rosbags captured by the racecar, visualizing the laser scan that the racecar was seeing was essential to adjusting the scaling factor necessary for the map to match up with the racecar. Overall, this lab was challenging, especially in terms of debugging and required countless hours of effort from each of our teammates to acquire even the racecar working partially in simulation with the recorded rosbags. I was grateful for all the work that my teammates put into this lab!