6.141/16.405 Lab 5 Report

Team 2: Grace, Hoang, Akila, Quang, Noah

Editors: Akila and Grace

April 4th, 2022

1 Introduction - Grace

The primary goal of Lab 5 was to implement the Monte Carlo localization algorithm into the robot so that it could navigate the hallways of the Stata basement by deducing its position and orientation using a lidar scanner and odometry. This was done by creating a particle filter utilizing a motion and sensor model and implementing it in the robot.

This lab builds on previous labs by using knowledge of wall following and how to transform data between different reference axes so that everything lines up with the base frame and orients the robot correctly within its environment. This lab will then be used as a base for writing algorithms in the next lab, allowing the robot to navigate its environment without a map. It will also be extremely important for the final challenge when the robot needs to guide itself around the Johnson track. Without being able to figure out where it is located, it could get lost and fail the challenge. This implementation will help us prevent this.

This lab report will detail how the algorithm was written and implemented in simulation and reality.

2 Technical Approach

2.1 Motion Model - Grace

The goal of the motion model was to spread out the particles from the previous pose so that the algorithm could find a new pose. This is important for the Monte Carlo localization algorithm because it allows the robot to finds it position and orientation within the environment.

The motion model was designed using odometry. The motion model started by taking in the Odometry message, specifically focusing on the pose values. Other values were not used since they did not help find the new pose of the robot.

After taking in the pose data, the pose is transformed using:

$$x_k = x_{k-1} + T\Delta x,$$

where T is a rotation matrix, and x_{k-1} and Δx are provided by the Odometry message. Specifically,

$$T = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The reason that the Δx values must be modified is so that they are referencing the global frame, instead of just navigating in relation to the robot. Normally, the rotation matrix would be enough in order to find the new pose transformation. However, since the motion model provides a set of particles, these particles need to be "pushed away" from the robot. This is because all the particles in the message are located on the robot. Therefore, in order to spread the particles out, random noise was added to the $T\Delta x$ term. The random noise was generated via Numpy's random normal Gaussian number generator. This noise was then incorporated by using the original equation for x_k by subtracting the noise from the right side, giving:

$$x_k = x_{k-1} + (T\Delta x - a * r),$$

where r is the randomly generated number and a is a designated constant set by trial and error. In this case, a = $[.5, .5, .15]^T$.

After the noise was integrated, the model returns a series of pushed out particles, giving potential poses of where the robot is after it has travelled. This model will be used in the particle filter so the robot can identify which pose is the most accurate representation of its location and orientation.

2.2 Sensor Model - Akila

The sensor model is a sub-component of the overall model that is used to assign likelihoods to the particles distributed by the motion model. More specifically, these likelihoods represent how likely it is to record a sensor reading from a hypothesized location. Good hypotheses will have higher likelihoods of being sampled in future iterations.

This likelihood is a combination of various cases that could be encountered including:
1) The probability of detecting a knowing obstacle in the scene
2) The probability of a short measurement (ex. reflections, unknown obstacles, etc)
3) The probability of a large (or missed) measurement
4) The probability of a random measurement

The sensor model is very complicated and requires a lot of computations. For a robot that is moving continuously and handling several particles, it would be extremely inefficient to keep recomputing the values at every time step. Additionally, many of the values will likely repeat. Therefore, we pre-computed a table of probabilities according to the formula prescribed above over a discretized range of values representing different positions in space. Reading off the probability reduces the time complexity and makes the robot's response to data quicker for the real-time decisions it is making. Figure 1 below is a graph of the distribution formed by the probability table.

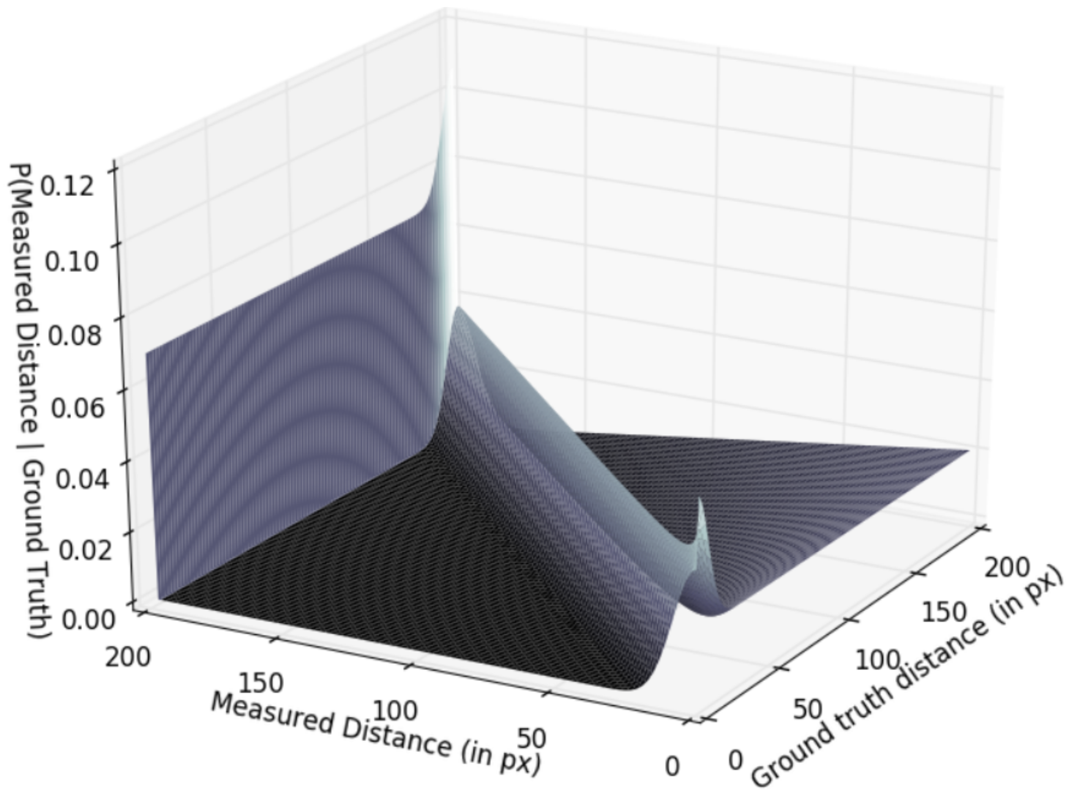

Figure 1: Probability Table Distribution of Ground Truth vs Measured Distances

The spike across the diagonal indicates the likelihoods of ground truth and measured distances that are very close to each other (i.e. good hypotheses) while the flatter regions represent pairs of values that are bad estimates. Using this table, we evaluated sample ground truth values with measurements to test the effectiveness of the sensor model before integrating it into the complete MCL algorithm.

2.3 Particle Filter - Noah

The particle filter is the CPU of our localization implementation. It synchronizes all the other components of our model to produce a complete and functioning system. The main idea behind a particle filter is to stochastically calculate the pose of our robot via the use of particles. Particles, essentially, are candidate poses for our robot. Each one has a corresponding pose and weight associated with it, where the weight represents our belief- or the likelihood- that the particle is the true pose of our robot. Given a map and lidar scan, we can probabilistically calculate our weights. The pose, on the other hand, is updated based on odometry data. Overall, our filter is comprised of four main parts: initializing our robot's pose, evaluating odometry data, evaluating lidar scan data, and updating the estimated pose of our robot. In our implementation, we decided to use 100 particles, as it gave us the best combination of speed and accuracy.

The first part of our particle filter is used each time the pose of our robot is initialized. When this happens, the first thing we do is disassemble our pose message into its position and quaternion component. Using this, we can then instantiate all of our particles to be at this exact pose. The reason we do this is that we assume the starting point is known. We only have error in our sensor measurements, so that is where we choose to model the probabilistic noise. Thus, we initialize all of our particles to be at the known starting pose of our robot.

The second part of our particle filter utilizes data from our odometry sensors to update the poses of our particles. When odometry data is published, we use the linear and angular velocities along with the change in time to calculate the changes in position and orientation:

$$dx = \dot{x} \cdot \Delta t$$

$$dy = \dot{y} \cdot \Delta t$$

$$d\theta = \dot{\theta} \cdot \Delta t$$

Where θ represents the angle of the robot with respect to the z-axis, and x & y follow the conventions we use in 6.141. This data is then sent to the motion model, which uses the changes in position in conjunction with stochastic noise to update the positions of each particle (see Section 2.1 for more details). This gives us a great transient response to the particle positions. Because the noise accumulates- however- our pose belief drifts, or deviates, each time we use the motion model, giving us a wider cloud of particles.

In the third component of our particle filter, we use laser scan data to update the weights of our particles. We chose to only consider 1 out of every 10 lidar scans received in order to make our particle filter more computationally efficient. On each considered scan, our first step is to downsample the data to only 100 rays in order to further reduce the processing requirements. Then, we feed the downsampled scan to our sensor model, which updates our particle weights (see Section 2.2 for more details). Once we have our updated weights, we resample our particles. In effect, this discards the ones with lower weight and narrows our pose belief by making our cloud of particles more compact. Finally, our last step is to model the uncertainty in our lidar scan by adding noise to our new set of particles. All in all, this component is great at mitigating the steady state error that accumulates from the drifting in our odometry module.

The final part of our particle filter computes the estimated pose of our robot. To do this, we calculate the unweighted average of our particles. This module is run anytime the particles are updated.

2.4 Simulation Implementation - Hoang

Once the motion model, sensor model, and particle filter worked independently, they were implemented into a simulation. The code was first put into simulation for initial testing. This was to avoid any unexpected reactions, like crashing, from the robot after translating the theoretical ideas into an implementation. Otherwise, this would be an expensive and time-consuming issue. In this simulation, simulate.launch, localize.launch, and wall_follower.launch were used to run the simulated car.

First the odometry message is run through the particle filter. This returns the new poses of particles which are then published using PoseArray. This helps visualize how the particles are spread, updated and resampled. An example of this can be seen in Figure 2.

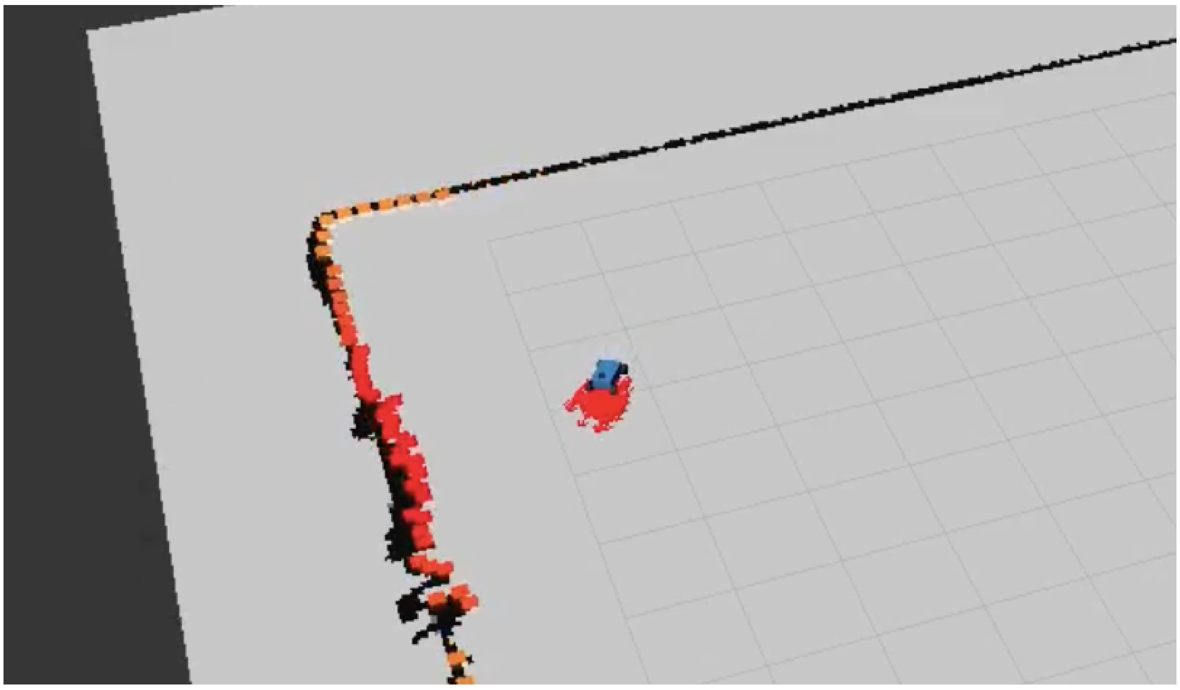

Figure 2: Particles Spreading in Simulation

In addition to returning the new poses of the spread particles, the particle filter also returns its best estimate of how the robot is oriented and where it is located within the simulated environment. This is published to the topic "/pf/pose/odom". The estimated pose can be viewed in Rviz, as shown in Figure 3, and can be compared to the actual direction of the simulated car by observation.

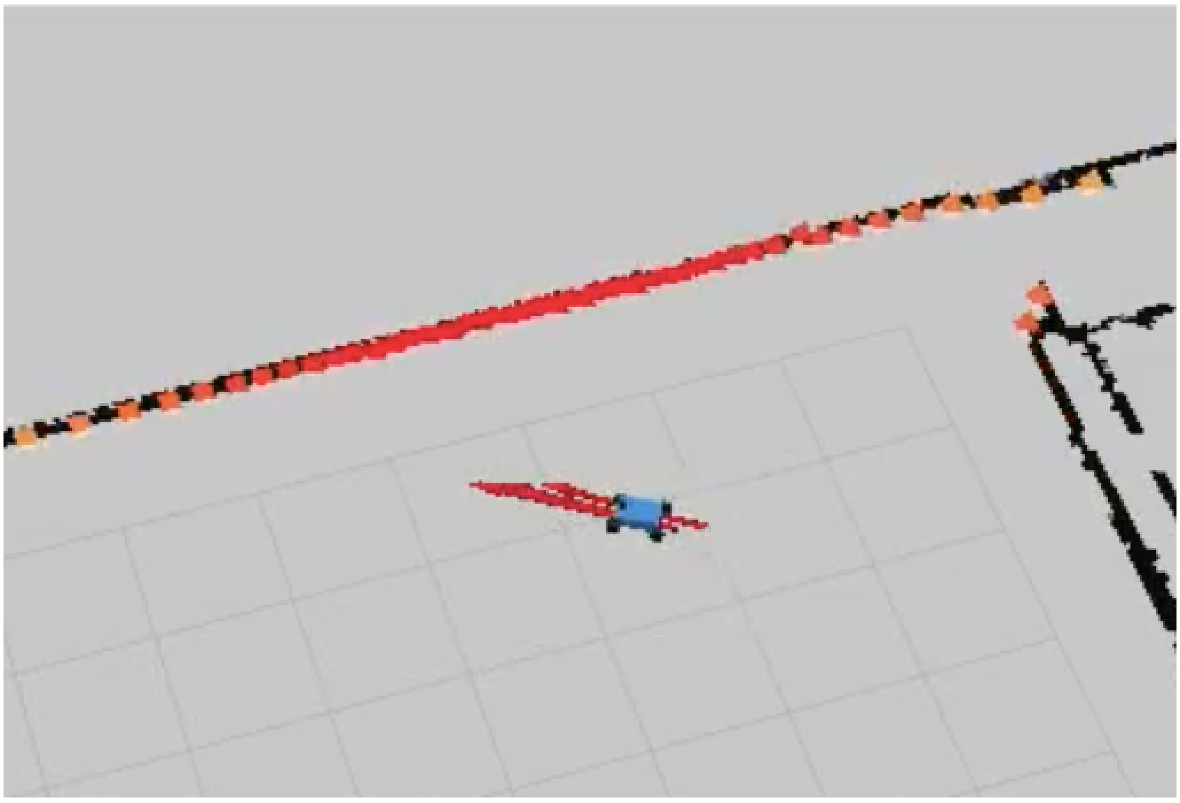

Figure 3: Estimated Pose and Orientation as Visualized in Simulation

Furthermore, since this car is running in simulation, the "ground truth" pose, or where the car actually is, can be obtained through an odometry message. This can then be used to calculate the distance and rotation error from the estimated pose in order to see how well the particle filter works. The distance error (or distance difference) is calculated as $\sqrt{(x_{ground} - x_{estimated})^2 + (y_{ground} - y_{estimated}^2)}$. The rotation error (or rotation difference) is calculated as $|\theta_{ground} - \theta_{estimated}|$. This is crucial because if the simulated car did not drive properly, the code would be unfit for implementation onto the actual robot. The details of how well the simulated car performed will be examined more in the Evaluation section, but the need for the simulated car to be able to orient itself was necessary for it to be able to navigate the Stata basement in the next step.

2.5 Robot Implementation - Hoang

The overall robot implementation was fairly simple. After the simulation code was tested, the Stata basement map was added to simulate.launch with the "map_server" package. Then the code was uploaded to the robot.

Since computation takes time and the robot had to be able to react fast, the frequency of the LIDAR scan message was reduced from 50 Hz to 10 Hz. In addition to reducing the frequency, each LIDAR scan message only processed 100 beams (equally spaced) instead of a full scan (about 900 beams).

In addition to computation time issues, the Velodyne LIDAR sensor published incomplete scans every time step. This meant that the robot could not see all of its surroundings at once. To fix this issue, a node called "scan_correction" was added to subscribe to "/scan". This combined 2 scans from 2 different time steps and published a corrected scan message to "/scan_correction".

Furthermore, ROS callbacks are not necessarily thread safe. Since both odom_callback and lidar_callback try to update particle, it is possible for them to have conflict, which would result in an incorrect estimation. Using threading with the Semaphore library when updating particles in odom_callback and lidar_callback helped ensure that particles were updated completely during each callback.

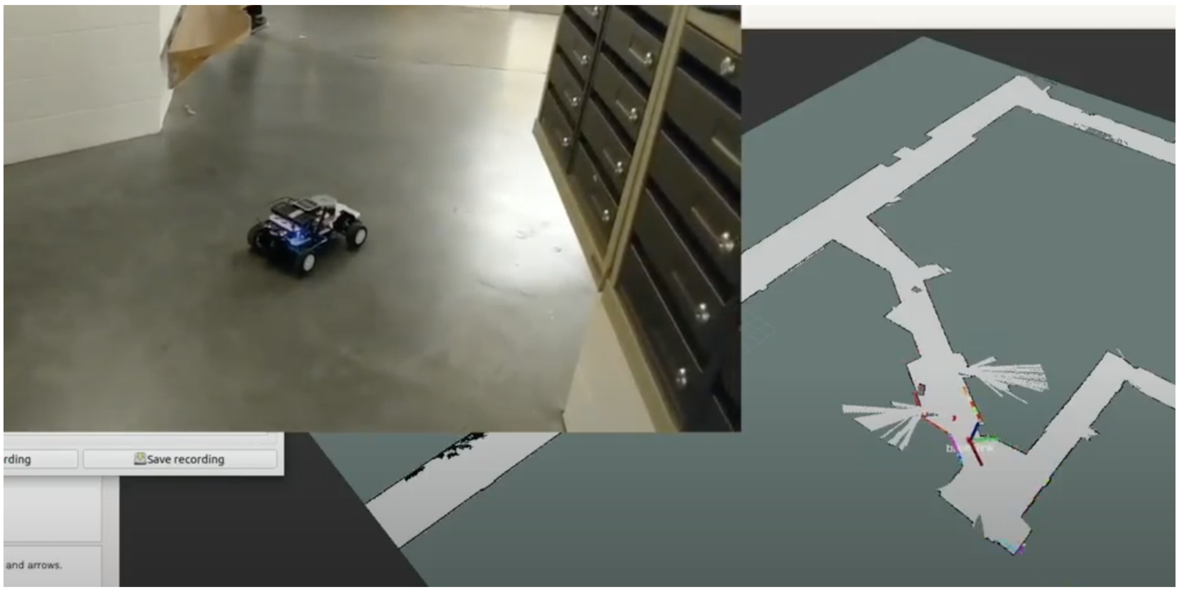

Figure 4: Robot in Simulation (bottom right) and Reality (top left)

3 Evaluation

3.1 Simulation Implementation - Noah

Our MCL implementation worked very well in simulation. For these tests, we quantitatively evaluated our model in two ways: with the wall follower simulation and on gradescope.

Using the wall follower simulation was ideal because it contains all of the components we need to quantitatively test our particle filter; the robot autonomously moves around the map, we can retrieve the simulated odometry and LIDAR data, and we know the ground truth pose. With this in mind, we ran the simulation and plotted both the distance and rotation error (in radians) - calculated as the difference between our particle filter position/distance and the actual position/distance - over a 400s interval. This can be seen below in Figure 5.

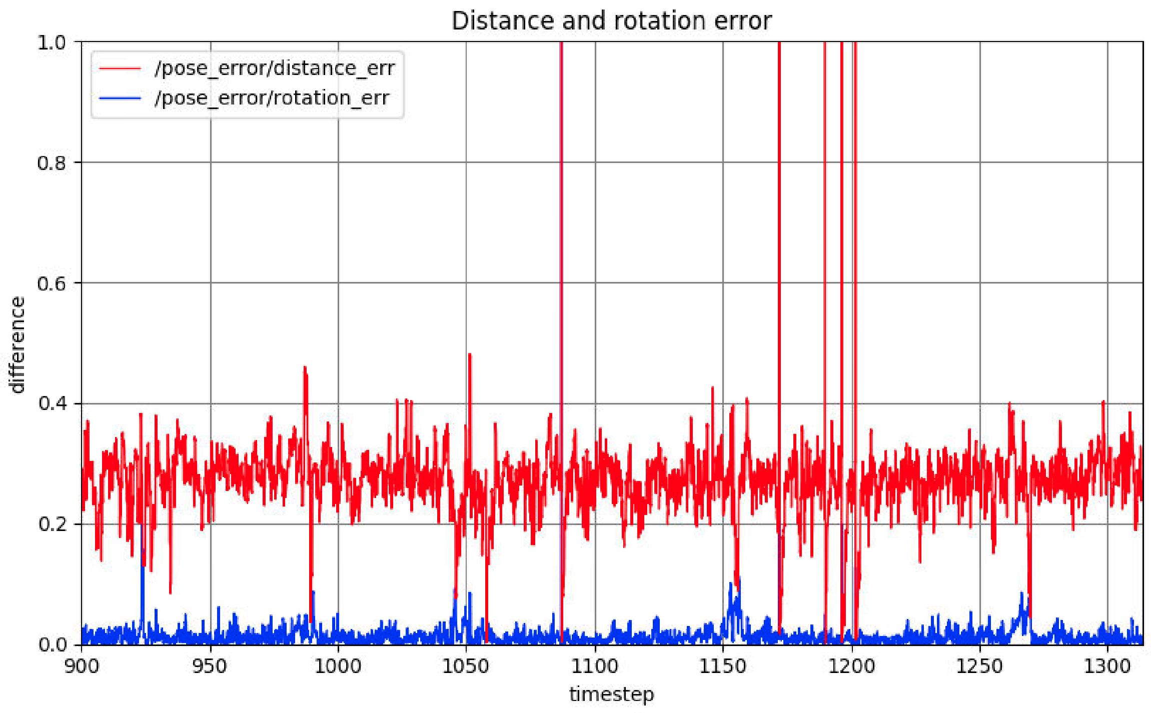

Figure 5: Robot Simulation Error

In this graph, the spikes occur at times when we reinitialize the pose of our robot. Overall, our particle filter remained around 30 cm away from the actual position and maintained a very small rotation error. This isn't perfect, but it is a great estimate given the computational trade-offs made and the fact that our model is stochastic in nature.

In gradescope, our model was tested and compared with the staff solution for three different cases. The first case was a simulation with no noise in our sensors, seen in Figure 6 below.

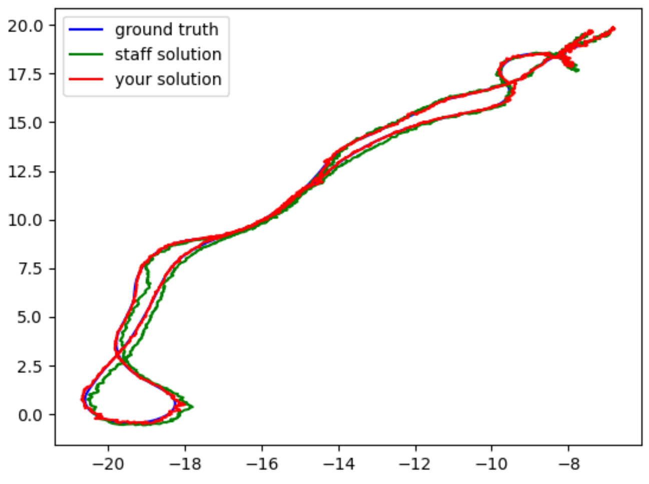

Figure 6: Gradescope Evaluation with No Noise

Here, you can see that our model closely followed the staff solutions. However, our time-average deviation from the trajectory was 0.06 m - 3 times less than the staff's!

The second scenario had some sensor noise. You can see this in Figure 7 below.

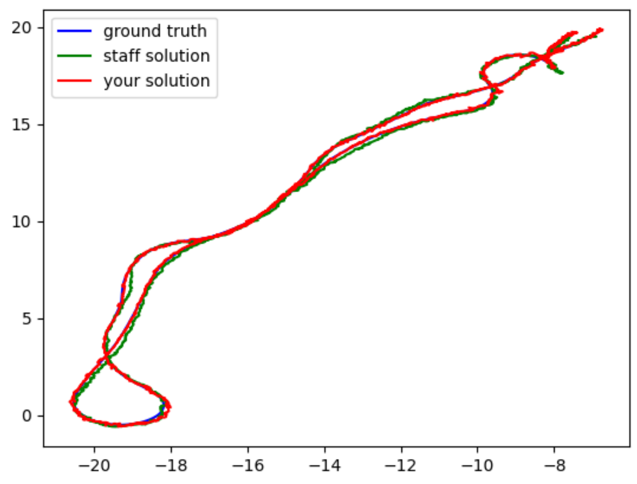

Figure 7: Gradescope Evaluation with Some Noise

Again, our robot followed a very similar trajectory to the staff's, and our time-average deviation from the true trajectory was a mere 0.06 m. This also was also 3 time's less than the staff's!

The last test on gradescope evaluated our particle filter when there is a lot of noise in the sensor data. You can see our results in Figure 8 below.

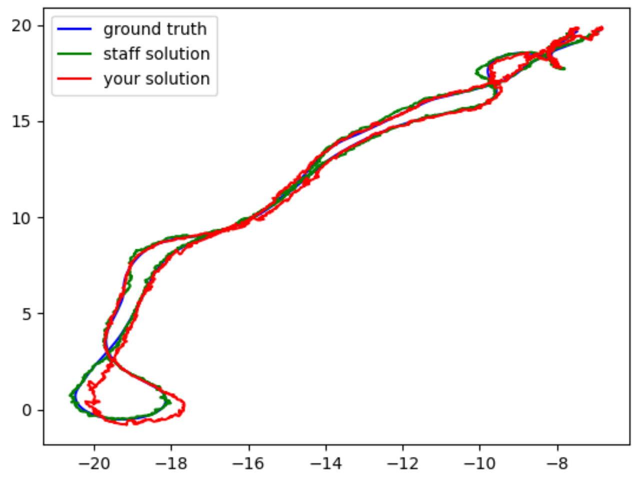

Figure 8: Gradescope Evaluation with a Lot of Noise

For the most part, our solution followed a similar trajectory to the staff's. However, they deviate when the robot goes around the loop in the lower left corner of the figure. Regardless, our time-average deviation from the true trajectory with a lot of noise was still only 0.19 m - identical to the staff's.

3.2 Robot Implementation - Grace and Akila

The robot worked well by observation. It successfully navigated through the State basement hallways and did not hit any walls. It was difficult to evaluate the actual robot quantitatively because there was no way of establishing a "correct" orientation of the robot in reality.

The main issue was that the Velodyne LIDAR only returns incomplete scans of the robot environment. This was problematic because it made it much more difficult to implement the Monte Carlo localization algorithm in the robot. The current resolution is to overlay previous scans with current scans in order to generate a complete scan of the surroundings. However, this issue also severely limited the speed at which the robot could drive at since it takes extra time to be able to process previous data as well. This needs to be fixed because the final challenge will include racing around the Johnson track, so being able to drive fast will be important in order to be competitive. The team's long term solution will be to search for a better software solution (than overlaying lidar scans) in the next lab or to replace the LIDAR sensor with a better one.

A video of our robot working along with the simulation can be found at this link.

4 Conclusion - Akila

The goals of Lab 5 were to implement and test the Monte Carlo Localization algorithm in both simulation and on the physical robot. We began by implementing the motion and sensor model and then integrating it into the particle filter. This particle filter was tested in simulation and then fine-tuned after transferring it onto the robot. The performance was assessed both qualitatively and quantitatively in the simulation and by observation on the robot. A good run was assessed by determining the similarity between the robot run, a ground truth path and the staff solution.

All of these assessments yielded positive results, specifically that our implementation of the MCL algorithm performed to our expectations and achieved the desired goals. While we did run into issues with our LIDAR not returning full scans for usage by the robot we were able to circumvent the issue and arrive at decent results. Future steps would be to better account for this LIDAR issue and continue testing the robot to speed it up for the final races.

5 Lessons Learned

5.1 Akila

This was quite a complex lab and I realized the importance of dividing and conquering between the group members. This also meant we required a lot of communication to complete tasks if people ran into issues. There were several instances where people ran into technical issues and it was easy to get help and move forward in the lab.

From a technical perspective, I learned about how to apply Monte Carlo localization to a real-life robotics scenario. Additionally, the process we were asked to follow through the lab (breaking it into smaller components) was a helpful learning for future technical tasks that can seem extremely daunting at first.

5.2 Grace

In this lab, I learned a lot about the value of being on a team and learned to appreciate how team members can support each other. I really appreciated how the rest of the team helped me to complete the parts that I had said I would do when I needed help, whether it was due to needing help with learning the material or just needing to manage different responsibilities.

In the technical portion, I got to familiar myself more with odometry and pose transformations, which we initially explored in Lab 2. Learning about Monte Carlo localization also exposed me to how complex seemingly simple autonomy problems are. This localization also taught me the importance of needing noise in data, which I used to always think was best filtered out.

In terms of the communication aspects, I learned the importance of letting the team know whenever there were setbacks or delays so that other people could help before the delay snowballed too far. It also was very helpful with reducing stress.

5.3 Hoang

In this lab, I learned that things are very different between real hardware and simulations. At first, our code ran well in simulation, but it was completely off on the car. I learned that we need to have time buffer for doing project because there are unexpected events that might happen. Also, I learned to support my team and other teams when they need help. I appreciate the opportunity to learn and grow together.

In the technical portion, I learned about the process of localization in the robot, which was fascinating. I learned how motion model, sensor model, and particle filter works, which deepened my understanding about MCL localization.

For the communication aspects, I learned that it is very important to communicate with my team. When some incidents come up, the team should be updated so that everyone can help and plan accordingly.

5.4 Noah

One of my main takeaways from this lab is the power of teamwork. This lab was very convoluted, so- at first- it was very beneficial to divvy up the work. However, to turn our individual work into a functioning system, we all needed to work together and effectively communicate in order to integrate our work.

Something else that I learned about was Monte Carlo localization. After the lecture on this, I left feeling I only had a surface level understanding of how MCL works. However, having done most of my work on the particle filter component of Lab 5, I came to understand this algorithm at a deeper, intuitive level.

At the lab meeting on Wednesday, I realized a few different things I can do to become a better presenter as well. For starters, I can include more visualizations on my slides. This week, my slide contained all text, and I've come to realize that this is very dull and disengages the audience. Visuals allow for a higher level understanding of what I'm presenting, and it can also engage the audience. Another improvement I can make while presenting is talking slower. This week, I noticed that I talk very fast and sometimes stutter. Instead, if I were to slow things down, I will come off as more comfortable and confident, and it will also give the audience more time to digest the material.