Lab 5 Report: Localization

Team 17

Devin Mccabe
Anish Ravichandran
Arnold Su
Anirudh Valiveru

6.4200 - Robotics: Science and Systems

April 15, 2023

1 Introduction - Arnold Su

The goal of this lab is to implement localization: determining where the mobile robot is located with respect to its environment. Localization is an important component to have on an autonomous car because it can allow other functions such as path planning where knowing the location of the is essential. We present two algorithms in solving localization, Monte Carlo Localization (MCL) and Simultaneous Localization and Mapping (SLAM) with Google Cartographer. MCL takes a predetermined map and uses "particles" to guess the location of the robot, where each particle is assigned a probabilistic "likelihood" for how accurate it is. MCL utilizes both odometry and LiDAR data to analyze and improve the particles. As implied by its name, contrary to MCL, SLAM has no predetermined map and does both localization and mapping simultaneously. In Section 2, we discuss the approach in our implementation these two algorithms. In Section 3, both algorithms are evaluated in both simulation and on the autonomous car. The end goal is to produce an accurate localization algorithm in simulation based on a Gradescope autograder, and working on the racecar moving around in the Stata basement.

2 Technical Approach - Arnold Su

This section presents our approach for developing and implementing two localization algorithms, MCL and SLAM. Each algorithm has its own initial conditions and resources, which guide our approach in developing and implementing them in ROS. In the end, both algorithms produce a pose representing the estimated position of the robot.

2.1 Monte Carlo Localization (MCL) - Arnold Su

At a high-level, Monte Carlo Localization estimates the robot's pose by generating multiple random guesses, each called a "particle". The algorithm would then assess how accurate each particle would be, and uses that information to produce an estimated pose.

2.1.1 Initial Set Up and Resources

There are three core pieces of information we need to implement the MCL algorithm: odometry data, LiDAR scans, and a predetermined map. Both the odometry and LiDAR scans are obtained and published by the physical car. Three maps were supplied as files: "basement_fixed", "building_31", and "stata_basement", as shown in Figure 1.

2.1.2 Technical Approach

Our initial set up of odometry data, LiDAR scan data, and maps are the exact inputs needed in the MCL algorithm. In MCL everything is centered around the particles and their respective likelihoods. And so given our data, to iteratively improve the particle's poses MCL combines both movement (odometry) and exteroceptive observation (LiDAR scans). This can be mapped into two modules: the motion model and the sensor model. The motion model uses odometry data to add movement to the particles by matching the car's movement. The motion model is important because if the particles never move, it will use the same observational data which stifles the particles ability to improve their estimation. The sensor model uses LiDAR scan data and a predetermined map to determine the likelihood of each particle, in addition to resampling particles based on their likelihoods. The sensor model is crucial because comparing LiDAR scan data and the map is the only way to assess how accurate a particle is. With a metric for determining how accurate a particle is and by resampling based on that metric, the algorithm is able to more efficiently estimate the pose with high accuracy because it "discards" bad particles and produces more good ones. Figure 2 diagrams the algorithm and its modules. Below each module is discussed in more detail.

Initializing Particles

With the given software resources like RViz we can easily guess or estimate the initial pose (this process is explained more in section 2.1.3 below). Therefore, to initialize the particles, we randomly distribute the particles around the initial pose using a Gaussian distribution. Additionally, we initialize each particles likelihood to be uniform across all particles. Intuitively, the more guesses we have, the better chance one of them is correct. So it is important to randomly distribute the initial particles because if they are all the same, we effectively have a single particle. Moreover, we use a Gaussian distribution centered around the initial pose because we would like our initial guesses to be "close" to our initial

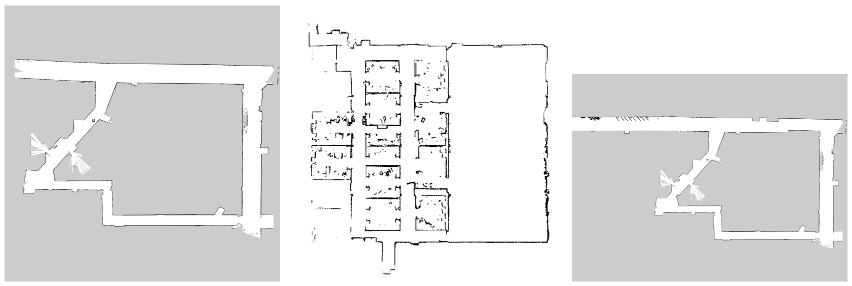
Figure 1: The maps given: basement_fixed, building_31, and stata_basement from left to right.

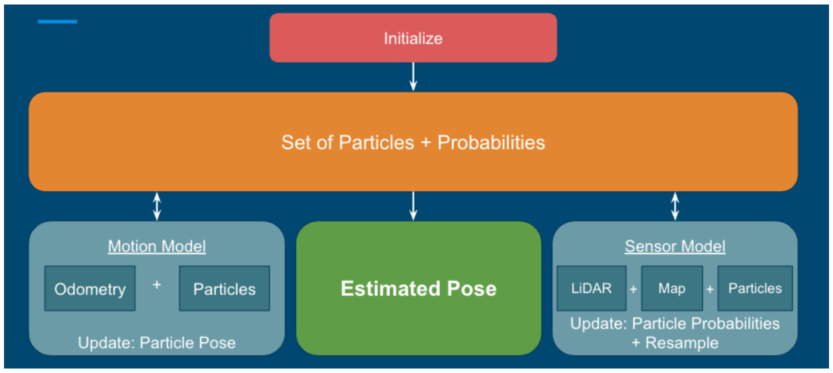

Figure 2: Overview of the MCL Algorithm

pose while accounting for inaccuracies that initial pose estimate may be.

Motion Model

The motion models incorporates the motion of the racecar into the particles. To do so, it takes in odometry data in the form of $\Delta x, \Delta y$, and $\Delta\theta$ representing the change in x, y coordinates (position), and θ (orientation). We are also given the x, y, and θ of each particle on the map. The goal now is to find the new x, y, and θ of each particle on the map given the $\Delta x, \Delta y$, and $\Delta\theta$ of the car. This problem can be framed as a problem about coordinate frames and transformations. We define the following:

- Let w be the world coordinate frame representing the map.
- Let p and c be points on the map (in the world frame), whose poses can be represented as p_p^w and p_c^w respectively.
- Define another coordinate frame p whose origin and orientation are defined by the pose p_p^w. This coordinate frame represents the particle/robot.
- Let p_c^p represent the point c in the coordinate frame of p, and R_p^w be the rotation matrix from coordinate frame p to w.

Now if we let p be the initial pose of the particle, and c be the final pose of the particle, then p_c^p is exactly the odometry data for the particle. So we are given p_p^w and p_c^p, and want to solve for p_c^w. As learned from lecture, this can be done with the equation

$$p_c^w = R_p^w p_c^p + p_p^w$$

The process is shown in Figure 3. To add robustness and tolerance to faulty data, our motion model also incorporates a small amount of Gaussian distributed noise to the odometry data. In other words, for each particle we use $p_c^p = \begin{bmatrix} \Delta x + \epsilon_x \\ \Delta y + \epsilon_y \\ \Delta\theta + \epsilon_\theta \end{bmatrix}$ where ϵ_x, ϵ_y, and ϵ_θ and independently identically distributed from a Gaussian.

Sensor Model

The sensor model incorporates the LiDAR scans of the racecar and compares them with ground truth distances determined by the given map and produces a likelihood for each particle. Translating into mathematical notation, the likelihood for each particle k is equivalent to $p(z_k|x_k, m)$ where z_k is the LiDAR scan readings, x_k is the position of particle k, and m is the given map. Additionally, let us define $p(z_k^i|x_k, m)$ as the likelihood of range measurement i of the scan. As described in lecture, the likelihood of a scan is computed as the product of the likelihoods of each of those n range measurements in the scan, or

$$p(z_k|x_k, m) = \prod_{i=1}^{n} p(z_k^i|x_k, m)$$

As described by the lab assignment, we determine each likelihood as

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

Where $\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1$. Below is a table defining and describing each term (hit, short, max, rand).

Case	Description	How we model	Equation
Hit	Probability of detecting a known obstacle in the map	A Gaussian distribution centered around ground truth distance	$\mathrm{p}_{hit}(z_k^i\|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(z_k^i - d)^2}{2\sigma^2}) & \text{if } 0 \leq z_k^i \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$
Short	Probability of a short measurement	Decreasing function as ray is further from robot	$\mathrm{p}_{short}(z_k^i\|x_k, m) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^i}{d} & \text{if } 0 \leq z_k^i \leq d \text{ and } d \neq 0 \\ 0 & \text{otherwise} \end{cases}$
Max	Probability of very large or missed measurement	Large spike at the maximal range	$\mathrm{p}_{max}(z_k^i\|x_k, m) = \begin{cases} \frac{1}{\epsilon} & \text{if } z_{max} - \epsilon \leq z_k^i \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$
Rand	Probability of a completely random measurement	Small uniform value	$\mathrm{p}_{hit}(z_k^i\|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if } 0 \leq z_k^i \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$

where d represents the ground truth distance determined by map m and pose x_k of range measurement i. Note that all α's, η, σ, z_{max}, and ϵ are all parameterizable.

The other functionality of the sensor model is to resample the particles based on their likelihoods. This process will iteratively improve our particles by focusing them around positions that are most likely to be correct. A graphic of this process, obtained from https://en.wikipedia.org/wiki/Monte_Carlo_localization, is shown in Figure 4. After resampling, we add a small amount of Gaussian distributed noise centered around 0. This is to prevent the particles from resampling too aggressively, where a "cloud" of particles quickly reduces down to a single particle. Overly-aggressive resampling causes a lot of random error and makes the algorithm much more dependent on chance.

Estimating Pose

To produce a single estimated pose with our particles and likelihoods, we simply produce a weighted mean for the x and y coordinates and weighted circular mean for the θ coordinate. In other words, given n particles each where particle i has pose x_i, y_i, θ_i and likelihood p_i, our final estimated pose will be given as

$$x = \frac{1}{n}\sum_{i=1}^{n} p_i x_i$$

$$y = \frac{1}{n}\sum_{i=1}^{n} p_i y_i$$

$$\theta = atan2(\frac{1}{n}\sum_{i=1}^{n} p_i sin(x_i), \frac{1}{n}\sum_{i=1}^{n} p_i cos(x_i))$$

2.1.3 ROS Implementation

To implement MCL in ROS, we split up the four modules of the algorithm (initalizing particles, motion model, sensor model, and estimating pose) into three files: `particle_filter.py`, `motion_model.py`, and `sensor_model.py`. `motion_model.py` and `sensor_model.py` will handle the functionality of the motion model and sensor model respectively, while initalizing particles and estimating pose will all be handled inside `particle_filter.py`. `particle_filter.py` will be the only ROS node, which will subscribe to topics to receive both odometry and LiDAR scan data, while publishing the estimated pose. Therefore, `motion_model.py` and `sensor_model.py` are utilized in callback functions for the odometry and scan data respectively. Note that the callback functions are threaded using python Locks to avoid concurrency (and potential bugs) between the callback functions. The reason for this set up was so that there was a centralized node (`particle_filter.py`) to store, update, and analyze the particles, and adding `motion_model.py` and `sensor_model.py` help modularize the implementation for testing and bug safety purposes. A full diagram of this setup is shown in Figure 5, along with an rqt graph in Figure 6. In the remainder of this section, the implementation of each module will be discussed in more detail. Note that all functionality and data will be written and stored using numpy to get performance (particularly latency) at a adequate level.

Initalizing Particles

To implement the algorithm as described in Section 2.1.2 - Initializing Particles, we require obtaining an initial pose. This is done by subscribing to another topic, "/initalpose". This is so we can utilize RViz. In RViz, by setting a pose at our desired start location, it will be published to "/intialpose" and picked up by `particle_filter.py`. While this method works seamlessly on simulation, there are more steps required for it to work on the racecar. When RViz publishes to "/initialpose", it is unable to be picked up by the node running on the racecar. So we utilize two given python scripts, `initialpose_sub.py` and `initialpose_pub.py`, to communicate the message from RViz onto the racecar.

Motion Model

The algorithm described in Section 2.1.2 - Motion Model was able to be implemented into python with numpy without much issue.

Sensor Model

To compute the quantity $p(z_k|x_k, m)$ as described in Section 2.1.2 - Sensor Model every time a new scan (z_k) is received is too computationally demanding and causes the script to run too slow to function well. To get around this problem, we precompute a probability table that gives a $p(z_k|x_k, m)$ for a given scan z_k and ground truth distance d given the pose x_k and map m. Note that obtaining the ground truth distance uses a ray tracing program given to us. The probability table will be two dimensional and indexed by z_k and d as integers from 0 to z_{max}. Notice this means that z_{max} is determined by the size of the table. Additionally, because now our probabilities have been discretized, extra steps to normalize values must be added on top of the equations described in Section 2.1.2 - Sensor Model. By having a precomputed lookup table, obtaining $p(z_k|x_k, m)$ will now take near instant time and have much better performance.

Additionally, there is some preprocessing that must be done to the data before calculating likelihoods. For one, the number of LiDAR scans is not necessarily the same as the number of beams (number of angle measurements for ground truth distance) we obtain per particle. Therefore, the scans must be downsampled appropriately by approximately matching each angle measurement to have an equivalent number of scans as the number of beams. Both scans and ground truth distances must be converted from meters to pixels, rounded into integers and clipped to fit in the range $[0, z_{max}]$ so that it can be used to index our precomputed probability table. Once all likelihoods have been obtained, they are normalized such that they sum to 1. Finally, resampling particles was a straight forward process using numpy functions and the newly computed likelihoods.

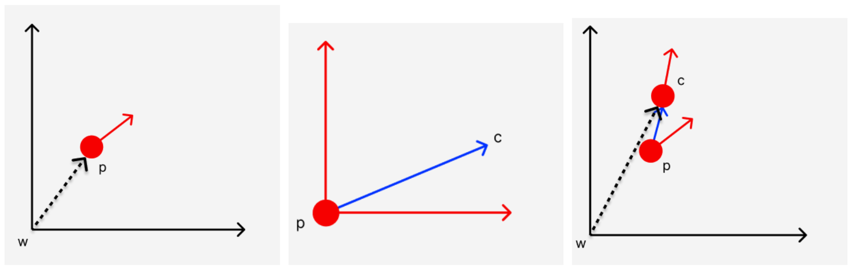

Figure 3: Diagrams displaying the initial position p in frame w, new position c in frame p, and position c in frame w from left to right.

Figure 4: Graphic displaying the density of particles, the likelihoods of each particle, and newly resampled particles using the likelihoods from left to right.

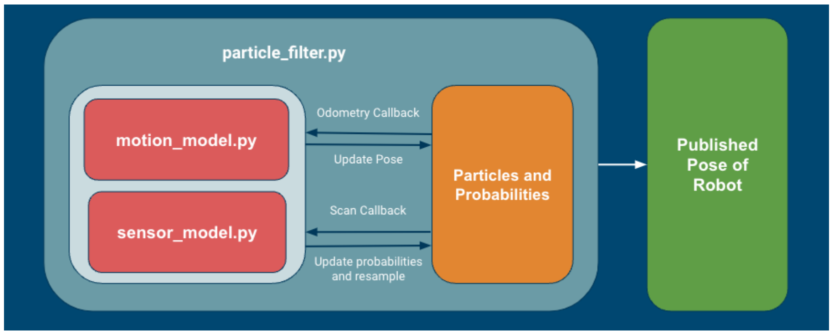

Figure 5: Diagram of the ROS implementation of MCL.

Estimating Pose

The algorithm described in Section 2.1.2 was able to be implemented into python with numpy without much issue.

2.2 Simultaneous Localization and Mapping (SLAM) with Google Cartographer - Anirudh Valiveru

An important part of the work that we did over the week was in integrating ROS Cartographer to enable simultaneous localization and mapping on our car. Simultaneous Localization and Mapping, or SLAM, is an algorithm that allows a robot to construct a map of its surrounding, given both sensor data of its surrounding and odometry data used to localize the robot. The localization aspect of SLAM is similar to localization when the map is known as we implemented in this lab; however, the additional constraint of needing to generate the map requires the SLAM algorithm to integrate laser scan data as well, matching features of the environment as it slowly changes to get a better understanding of the 3D nature of its environment and its place within it.

Cartographer is a ROS package developed by Google as a modular SLAM algorithm that allows robotics engineers to integrate SLAM into their own applications. While the technical details of SLAM are outside of the scope of this paper, the figure below from Cartographer's documentation demonstrates that SLAM is a multifaceted system with various components that work in tandem to construct a map accurate to the ground truth.

Since our system only used odometry data (published to 'vesc/odom') and laser scan data (published to '/scan'), our system's RQT graph representing all nodes available is simpler than the system diagram above, but still publishes topics that represent scanned submaps, constraints, feature matching data, an estimated trajectory, and landmark poses to generate the occupancy grid that finally represents our scanned map.

After installing Cartographer, we tested the package by running the program on both the demo bag and in real time on the robot. When running the experiment on our robot, we found that odometric drift was a big factor in determining the shape of the generated map. Although loop closure is generally used in SLAM algorithms to account for this drift, we were unable to make a full loop around the Stata basement due to connectivity issues with our router. Screenshots from both the demo and the car (Figures 9 and 10) are included below.

Enabling cartographer with our car was an incredibly valuable learning experience because it taught our team valuable lessons in integrating third-party software, especially when working with out-of-date documentation, disparate

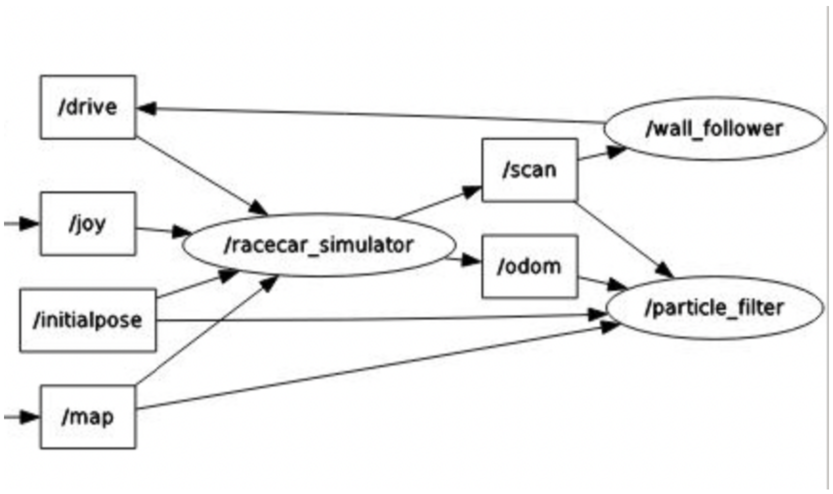

Figure 6: The rqt graph of the localization implementation in ROS.

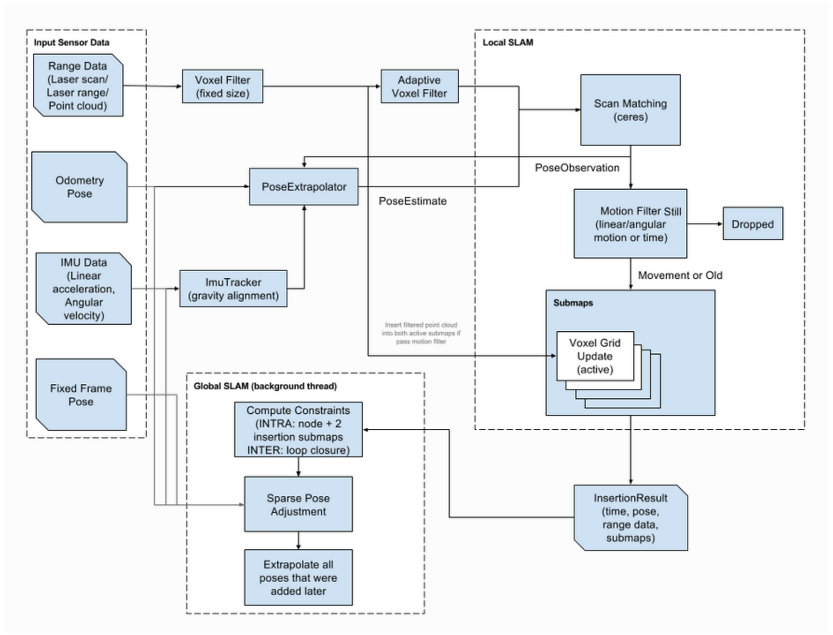
Input Sensor Data
Range Data (Laser scan/ Laser range/ Point cloud)
Odometry Pose
IMU Data (Linear acceleration, Angular velocity)
Fixed Frame Pose
Voxel Filter (fixed size)
Adaptive Voxel Filter
PoseExtrapolator
PoseEstimate
ImuTracker (gravity alignment)
Local SLAM
Scan Matching (ceres)
PoseObservation
Motion Filter Still (linear/angular motion or time)
Dropped
Movement or Old
Submaps
Voxel Grid Update (active)
Insert filtered point cloud into both active submaps if pass motion filter
Global SLAM (background thread)
Compute Constraints (INTRA: node + 2 insertion submaps INTER: loop closure)
Sparse Pose Adjustment
Extrapolate all poses that were added later
InsertionResult (time, pose, range data, submaps)

Figure 7: System diagram of sensors and modules in Cartographer.

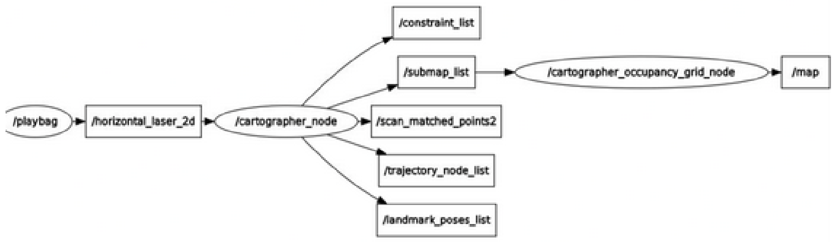

Figure 8: RQT graph representing Cartographer running in simulation.

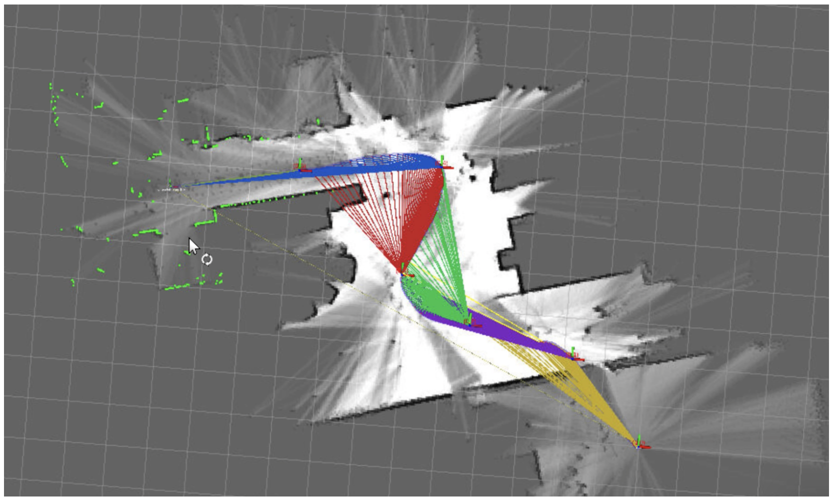

Figure 9: Cartographer map generated from demonstration bag (REVO LDS).

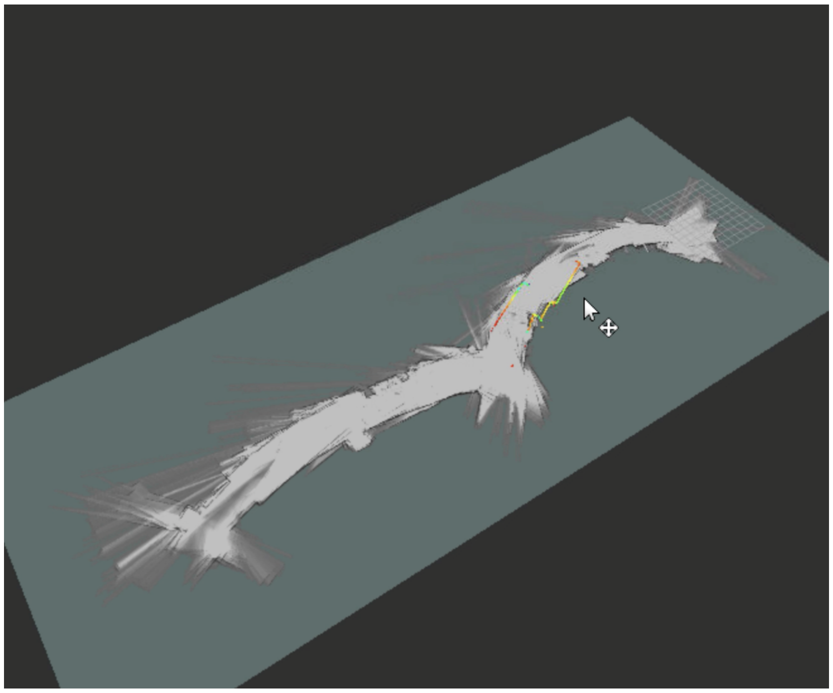

Figure 10: Corner of Stata Basement in Cartographer.

tools ranging from inconsistent Docker setups to apt-get and rosdep, and automating repetitive tasks with shell scripts. Due to some error in our Docker setup, running Cartographer in simulation proved to be a valuable lesson in patience, since we needed to reinstall the package each time Docker was restarted. While it was a difficult process to get to work, the lessons gained in understanding tools like Bazel, RVIZ, and rosdep were invaluable.

3 Experimental Evaluation

3.1 Simulation Based Evaluation - Devin McCabe

RViz Visualization

RViz visualization software was used to visualize the effectiveness and functionality of our particle filter and pose estimation. The simulation and RViz were used to determine if the particle filter and models were working, the estimated rate of convergence of the algorithm, the impact of adding in noise, and the error associated with the estimated pose produced. The pose of each particle was published and then visualized as a pose array, allowing the viewer to get a relative indication of how quickly the particles converged to a single pose. The ground truth pose was then visualized alongside the pose of the particles. The two poses could then easily be compared to determine performance as shown below in Figure 9.

Performance Improvement with Added Noise

Noise was added in the initialization, motion model, and sensor model sub-parts of our pose estimation as described in Section 2. The added noise resulted in better long-term performance, where the estimated pose error stayed low over time. The relative time for the particles used for estimating the pose to single in on a central location could be seen using the visualization of our simulated race car. With no noise added, the particles converged to a singular pose estimate quickly. As shown in Figure 10, the estimated pose error was minimized in approximately 0.1 seconds.

Settling to a pose quickly normally would be a good characteristic, as the car could be located as fast as possible. The model with no noise converging quickly was instead a cause for greater error down the line. The now singular pose would become off from the ground truth pose and the error would continue to build. This led to the pose estimate becoming entirely ”lost” during some tests. As shown in Figure 11, the error continues to rise and become a worse estimate.

Noise was then added to the motion and sensor model components to keep the

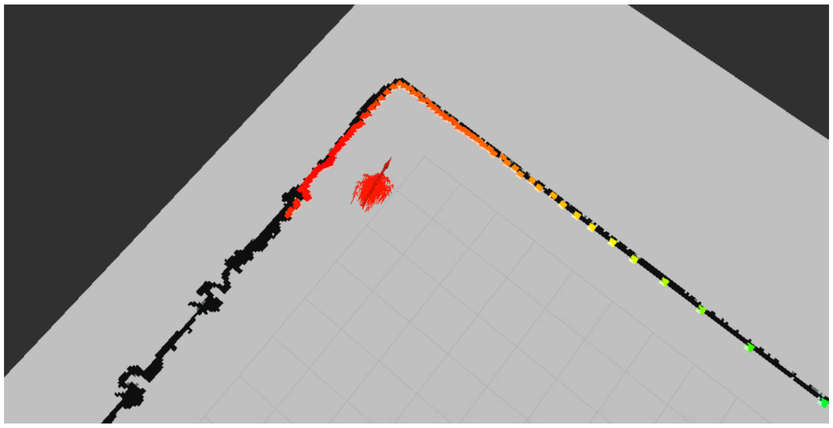

Figure 11: The long arrow represents the ground truth pose of the car within the simulator. The cloud of particles is represented by the smaller arrows surrounding the pose.

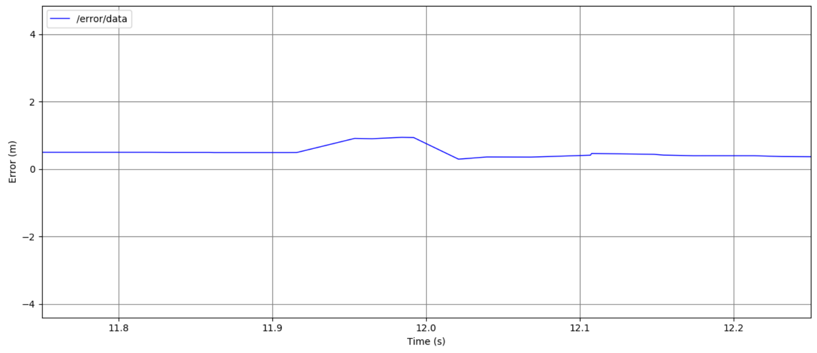

Figure 12: The graph above displays the error of the estimated pose. The error drops to its lowest value after initialization in approximately 0.1 seconds.

error from continuing to increase while estimating the pose for a long run. Using Gaussian noise to update the particles each time the pose is estimated showed favorable performance characteristics. While keeping a short settling time to the minimal error for each test, the error did not have the same problem of increasing over time, unlike the no-noise tests. The convergence rate stayed at approximately 0.1 seconds, keeping consistent with the performance without added noise. The long-term error stayed at a consistently low value of approximately 0.5 meters. Figure 12 below displays the long-term behavior of the error graph.

The use case for the particle filter requires that the estimated pose stays at a consistent error for long runs. The estimated pose getting disconnected from the ground truth pose makes it so that the estimation stays consistently close to the car's position. Some amount of error will inherently come with the estimated pose. The large and increasing error will make the pose estimate become irrecoverably far from the ground truth. This could be seen in the testing of the particle filter. The simulated runs showed a large average deviation from ground truth when there was no noise added. Figure 13 shows what happens without the noise added for long runs. The error continues to increase as the estimated pose converged to an incorrect location. The average deviation for that specific test was 15.6 meters while with noise added, the deviation was 0.277 meters.

3.2 Robot Performance Evaluation - Anish Ravichandran

Evaluating localization on the robot presented a new set of challenges. There were difficulties setting the initial pose estimate on the racecar due to the one-way nature of the communication between racecar and Docker command lines. Furthermore, metrics that were straightforward to evaluate in simulation were not so in a real environment. For example, in simulation, one can generate convergence rates by calculating error from the difference in localization pose estimate and car coordinates, however in the real world, there was no simple method to measure the car's ground truth position at any given time of the run. Some proposals were to have a marked surface with a grid of fixed lengths or to chart a predetermined path for the car to drive which was already measured with a ruler, however each of these methods did not guarantee a significant amount of accuracy for the effort it would take to implement them. Ultimately, our team decided to rely on qualitative data, with conclusions strengthened by multiple tests.

A video of the working localization program on the racecar is embedded on our team's webpage. Upon visual evaluation, it is clear that the robot can quickly locate itself based on the particle filter and sensor model, and our program is robust enough to not be affected by people walking within several feet of the robot, along with other obstructions in the hallways that are not accounted for in the preloaded map.

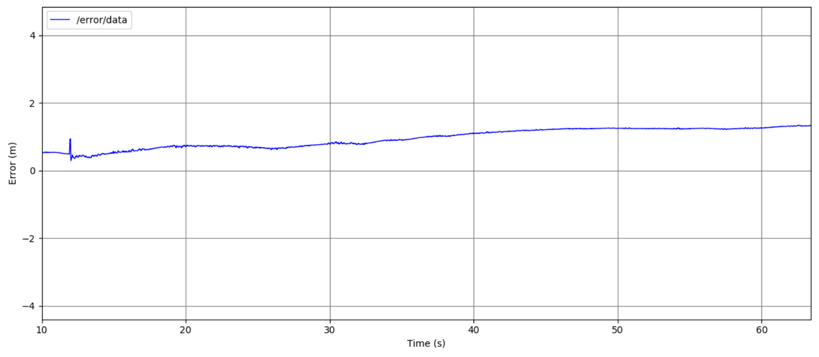

Figure 13: The graph above displays the error of the estimated pose. The error continues to rise over time and does not settle to a value.

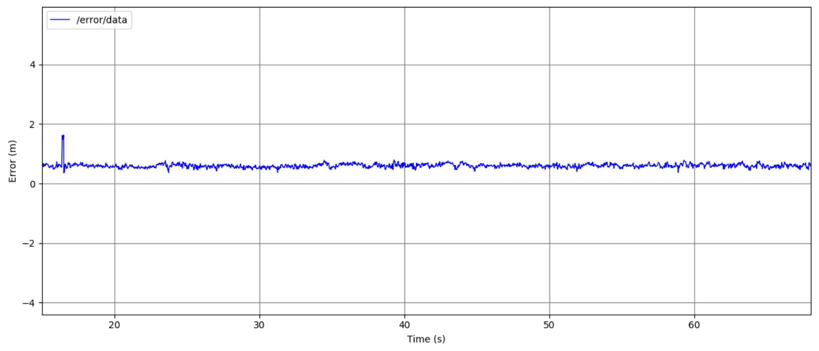

Figure 14: The graph above displays the error of the estimated pose. The error settles to approximately 0.5 meters and does not increase over time.

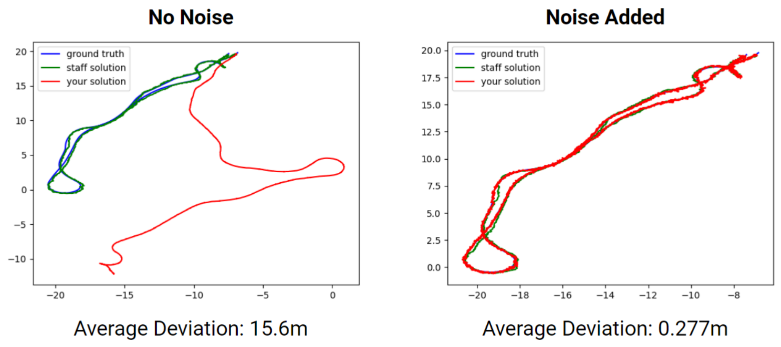

Figure 15: The two maps above display the estimated pose in red compared to the ground truth in blue and an acceptable solution in green. The average deviation was 15.6 meters for no noise while it was 0.277 meters when noise was added.

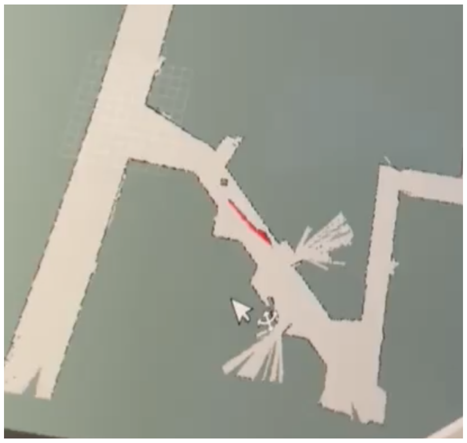

Figure 16: An RViz visualization of the robot's path in red by adding the particle positions in real time. Note how the particle path thins in a long hallway.

4 Conclusion - Anirudh Valiveru

As our team's first introduction to implementing probabilistic robotic algorithms for localization, this lab provided us with many learning opportunities in engineering systems with multiple parts. Since this lab explicitly required us to think about how probabilistic algorithms can be used to help understand a robot's pose, much of the lab was spent understanding the equations behind these algorithms while gaining an intuition for how initialization and resampling parameters affect the effectiveness of our approach. Some of our insights, such as the effect that resampling noise has on the particles' convergence, or the effect of motion model noise on our model's stability, are only learned through explicit experimentation with our system.

There also were a couple of minor technical challenges that we needed to overcome to achieve our lab objective. The first is learning how to write thread-safe code using locks. Since both our motion model and our sensor model have callbacks that independently update a global array of particles for the particle filter, explicit multithreading functionality is necessary to ensure that important data isn't overwritten before it can be incorporated into the localization algorithm. The tool that we found most useful in Python's `multithreading` library is the `Lock` class, which let us enforce sequential constraints on our asynchronous callbacks. The second is integrating third-party packages for part E of our lab to enable SLAM with our robot. There were many changes that we had to make to get cartographer to run; for example, we needed to change the `BUILD` file's name to `build.bazel` while installing Protobuf in order to build the project.

Aside from this, the primary non-technical lesson that we learned over the past few weeks was in time and resource management. While we had been given the opportunity to spend more time on the lab through our extension, trying to plan our progress around spring break was difficult, especially with a tight looming deadline. Resilience was another important lesson; even though we had planned to record our data using rosbags, for example, we ran into some problems with clock skew that rendered our recorded bags unusable. While we were eventually able to generate the necessary data for our report, these technical difficulties taught us a valuable lesson about being thorough and deliberate with the tools at hand.

Some future improvements that we would implement, given time, would include integrating wall following and/or line following, as well as perhaps trying to different source of input data to generate our localization. By expanding the scope of our lab, we may be able to provide a compelling demonstration of our car's localization capabilities while understanding the tradeoffs of using certain data more strongly.

5 Lessons Learned

Arnold Su - For the technical portion of this lab, I feel I learned a lot about working with numpy and discovering a lot of techniques and functions to manipulate arrays of data. Moreover, implementing locks to avoid interleaving between functions was something I never done before either. There was a also a lot of struggle with debugging subtle errors within ros, and learning to work around ROS's limitations to find ways to debug (like implementing a way to print information on first or second callbacks and RViz) was a big lesson for me. From a communication perspective, I learned it was hard to describe multiple moving parts as this lab had many different files and modules working very closely together. I found that using diagrams was very helpful in this aspect. Finally, on the collaboration aspect this lab taught me that it was difficult to smoothly get stuff done on time and not rushing last minute no matter how early in advance certain things are accomplished, which was both frustrating but also a learning point for the future.

Anirudh Valiveru - As the person responsible for the Cartographer/SLAM section of this lab, the main lessons that I took away were certainly those pertaining to the difficulty of integrating third-party libraries into my work. Using cartographer with the Docker container was especially difficult because of various inconsistent behavior pertaining to the installation of packages like `abseilcpp`. I also learned the importance of double-checking different aspects of our multifaceted system before being certain that everything would work. In particular, our team had run into a major issue with the rosbag files that we recorded, as it turns out that our robot was experiencing clock skew while those files were being recorded. This rendered this data useless, but while we were eventually able to get good visualizations for both localization and SLAM, being more thorough and double checking our data before calling it a day may have saved us hours of debugging over the week. For collaboration, this lab taught me the importance of parallelizing my work; since I had to install Cartographer several times, parallel processing with my responsibilities allowed me to support my team in aspects like debugging and data collection while troubleshooting cartographer.

Devin McCabe - I found this lab a great example of the use of simulation to aid in adjusting our algorithm. Looking at the particle pose array in RViz made it very simple to see what was really happening. It would have been very difficult to see that the convergence was too quick when the code was only tried on the robot. The error graph was helpful as well. It was the only non-visualization tool other than the auto grader tester itself to determine how well our particle filter worked. In addition, this was a difficult lab to manage our time on. It was a large and difficult challenge that we had multiple weeks to complete. We ended up having to continue to communicate and make sure we were all set with respect to our final deadlines.

Anish Ravichandran - Learning to collaborate with multiple different and separate sections was a major lesson learned for me. Because we are in a tough part of the semester, we learned how to communicate our workloads a week in advance, allowing us to either offload our work or plan to complete it ahead of time. In the technical sphere, I learned how especially for robotics, it is easier to record rosbags and videos all in one sitting rather than spreading them among multiple days. More often than not, I ran into one-off errors that caused me additional hours of setup each time I tried recording tests. In the future, I plan to clear my schedule and coordinate with my team so that we can perform testing all in one sitting.