Lab #5 Report: Monte Carlo Localization

Team #11

Quincy Johnson
Aileen Ma
Dylan Goff
Matt Jens
Anika Cheerla
6.141/16.405

April 5, 2022

1 Introduction (Aileen)

Humans may find the task of contextualizing their state relative to their environment to be a trivial matter, but this is far from the case for a robot. With the help of landmarks, signs, and estimations, a human could reasonably determine where they are given nothing but a map. Given this information, a robot can also use a variety of techniques to determine its state on a map; although the process is much more complex than that of the human thought process. In this lab, we explore one such way to implement an algorithm for a robot to determine its state given a map.

The objective for lab 5 is to implement Monte Carlo Localization (MCL) such that the robot is able to identify its position and orientation given a known map. Within this algorithm, the robot will generate a cluster of hypothesized states at every timestep, prune states that do not match the data from the sensors and gradually converge the average hypothesized state to the actual state of the robot. We need to ensure that our implementation works both in simulation and on the physical robot itself. In simulation, we run localization while adding randomness to model real noise. On the robot itself, we tune parameters and address edge cases that do not appear in simulation.

There are many challenges to consider, including accounting for noisy sensor data, preventing concurrent modification from LIDAR callbacks, publishing at a sufficiently high rate, and optimizing code to run efficiently. Overcoming these challenges is necessary because localization is a critical piece in constructing an autonomous robot as it informs decisions that a robot must make to successfully interact with its environment.

2 Technical Approach

Monte Carlo Localization is a method to estimate the location of a robot given a known map of the area it is enclosed in and a set of *particles*. Each particle is a hypothesis of the state of the

robot (in our case, a state is a location and orientation). At a high level, Monte Carlo localization operates by producing particles, guesses for where the robot may be and finding the particle that best corresponds with the current position of the car. While a robot is moving, the algorithm is continued by producing more particles using knowledge of how the current set of particles should move in space and time based on the dynamics of the robot.

In this section, we will outline the various parts comprising the MCL algorithm: the motion model, the sensor model, and the particle filter. We further simplify our implementation by assuming that we are given a roughly known initial position for the robot instead of assuming global localization (also known as the kidnapped robot problem). This simplification reduces the state space over which our particles may be initially distributed and limits the number of iterations of resampling needed to converge to a rough approximation of where the robot is.

The MCL algorithm follows six primary steps.

1. A cluster of particles is initialized around the initial robot estimate to represent the hypothesized robot state.
2. Using the odometry values, forward propagate the state of each particle. In simulation, we also add noise to make the particles spread out in various directions. This step comprises the motion model.
3. For each particle, compute the probability that the state of the particle corresponds with the actual LIDAR scan at a given timestep by looking up values from a discretized table. For particles with a higher probability, assign a higher weight for future resampling purposes. This step comprises the sensor model.
4. Resample from the particle distribution, assigning a higher weight to the particles with a more likely state.
5. Calculate the average pose of the resampled particles, taking into account both position and orientation.
6. Repeat steps 2 through 5 for every time step.

2.1 Motion Model (Anika and Matt)

The motion model takes in the odometry reading $[dx, dy, d\theta]$ from dead-reckoning integration of motor and steering commands and applies it to each particle. The motion model is then used to update the position and orientation of each particle at a specified time step.

Mathematically, the motion model f, takes in the old pose of the car $\mathbf{x}_{k-1}$ and odometry data $\Delta\mathbf{x}$, and returns the current position of the car $\mathbf{x}_k$. As defined in our instructions,

$$\mathbf{x}_k = [x, y, \theta]^T = f(\mathbf{x}_{k-1}, \Delta\mathbf{x})$$

In our code, first we read in the odometry data. If we are testing in simulation, we also add noise to generate particles with a randomly generated normal distribution of positions and orientations. Then we transform the odometry data into the world frame using the rotation map below.

$$R = \begin{bmatrix} \cos(d\theta_n) & \sin(d\theta_n) & 0 \\ -\sin(d\theta_n) & \cos(d\theta_n) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, we add the transformed odometry data $[dx, dy, d\theta]_W$ to the particles from the previous time step.

$$x_{particles} = \begin{bmatrix} x_{p-1,0} & y_{p-1,0} & \theta_{p-1,0} \\ x_{p-1,1} & y_{p-1,1} & \theta_{p-1,1} \\ \vdots & \vdots & \vdots \\ x_{p-1,n} & y_{p-1,n} & \theta_{p-1,n} \end{bmatrix} + \begin{bmatrix} dx_0 & dy_0 & d\theta_0 \\ dx_1 & dy_1 & d\theta_1 \\ \vdots & \vdots & \vdots \\ dx_n & dy_n & d\theta_n \end{bmatrix} \begin{bmatrix} \cos(d\theta_n) & \sin(d\theta_n) & 0 \\ -\sin(d\theta_n) & \cos(d\theta_n) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Every time the particle filter resamples, it calls motion model, and each of these steps is executed, which is why we utilized numpy arrays instead of for-loops in our implementation. Fast updates are key to the accuracy of the model.

2.2 Sensor Model (Aileen)

The sensor model provides a probability that a particular sensor reading z_k would occur from a hypothesis position x_k. Once we have received the updated position and orientation of the particles from the motion model, we use the sensor model to weight the particle distribution for future resampling. For each of our particles, we use the sensor model to provide a probability that sensor reading z_k would be seen from the position and orientation of that particle. Given that a sensor reading contains many range measurements within the scan, we compare each individual range measurement with the range measurement given by the hypothesized ground truth from the particle and multiply across. That is, for any given map m,

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

where we are finding the probability of a given sensor reading z_k from hypothesized particle state x_k across i range measurements for a given scan. The probability of a given sensor reading takes into account a few cases: the probability that the sensor reading has accurately detected the obstacle (p_{hit}), the probability that the sensor reading falls short of the actual measurement (p_{short}), the probability that the sensor reading overestimates the actual measurement (p_{max}), and the probability that the sensor reading returns some completely random value for no particular reason (p_{rand}). We can express all of these with the following equations:

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

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

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

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

Given some experimentally determined weighting coefficients for each of these four cases $\alpha_{hit}, \alpha_{short}, \alpha_{max}, \alpha_{rand}$ where $\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1$, we can thus express our probability of reading $z_k^{(i)}$ given a particle state x_k as

$$\begin{aligned} p(z_k^{(i)}|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m) + \\ \alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m) \end{aligned} \tag{2}$$

We experimentally determine values of $\alpha_{hit} = 0.74, \alpha_{short} = 0.07, \alpha_{max} = 0.07, \alpha_{rand} = 0.5$. Additionally, we have that $z_{max} = 5$.

The equations above represent the calculations for continuous values returned by the LIDAR. However, performing the calculation above is very costly for each particle. For 100 scan measurements across 200 particles, calculated between 20 to 50 times a second, we may reach upward of one million computations a second. As a result, it is much more computationally efficient to precompute a table with discretized values for z_{max} and d, and to perform lookups within the table. For the purposes of discretization, we convert lidar scan measurements from meters to pixels. An example with five discretized values is shown below.

Note that values across the diagonal have the highest probability due to the initial alpha values we have chosen. Here, $\alpha_{hit} = 0.74$ represents a weighting that suggests a particular sensor reading is likely to correspond with the ground truth of the state itself. As such, values along the diagonal, where $z_{max} = d$, will have the highest probability. Also, note that columns (actual distance values) are normalized to 1. This is because for a given ground truth, the distribution of sensor readings must form a complete set of possibilities for adding up to the given.

Once we have precomputed the sensor model table, we can systematically plug each forward-propagated particle of the motion model into the table and compute the probability of the particle new state given the LIDAR scan observation. Given a set of probabilities corresponding to the new states, we have a weighting that we can apply to the particles for future resampling. However, before we do so, we squash the weights by raising each probability to the power $\frac{1}{2.2}$. In doing so, we prevent peaks from becoming too sharp and forcing resampling to converge to a very limited sample space. The effects of squashing are shown below in Figure 2.

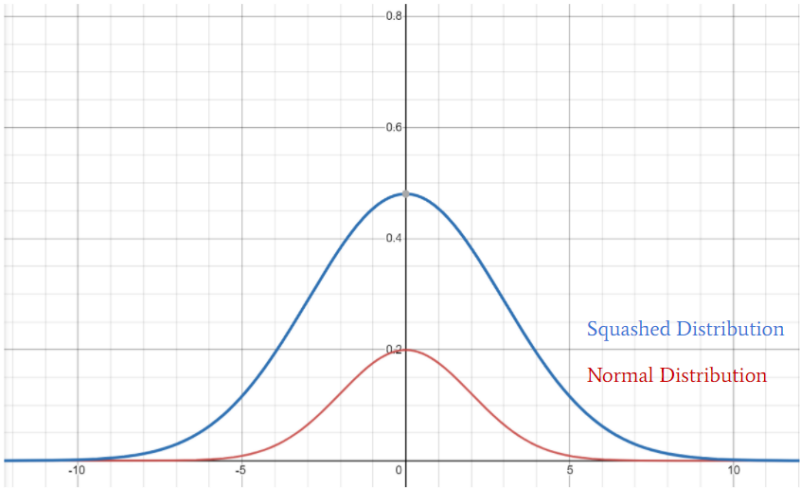


Figure 1: Comparing a normal distribution with a squashed distribution

2.3 Particle Filter (Quincy and Matt)

We construct a particle filter that uses the MCL algorithm to allow the robot to accurately estimate its location in a known environment.

2.3.1 Initialize Particles

Under the assumption that we are given the rough initial position, we initialize 200 particles around the estimated initial position. While a larger number of particles may allow for faster convergence to the true position of the robot by providing a larger pool of candidate states, it is also costly to forward propagate and compute probabilities for a larger number of particles. By choosing 200 particles, we achieve a balance between efficiency and accuracy.

2.3.2 Update Particle Probabilities

Once the particles are initialized, the robot will begin to continuously update the probability of the robot having certain particle poses. The first step to this process occurs when the robot's odometry system publishes an update. With this update, we obtain the robot's current 'twist' value as reported by the odometry which represents the robot's current angular and linear velocity in free 3-D space. Since the robot is grounded in the 2-D plane, only the linear x,y and angular z components of the 'twist' are needed. Using the 'twist', we can calculate a fairly accurate representation of the change in odometry Δx since the last 'twist' update. Δx and the particles are then passed into the motion model to recalculate the new particles' poses. The last step in updating the particles' probabilities occurs when the robot receives new LIDAR scan data. We pass this newly received data into the sensor model and update the probabilistic weights for each particle to the weights returned by the sensor model.

2.3.3 Re-sample Particles

With the particles' newly assigned probabilistic weights given by the sensor model, we can now resample the particles to provide a new distribution of particles weighted towards areas of high probability. This is done by first normalizing the weights between 0 and 1 to accurately represent the probability of a particular particle being resampled, and then updating the particles based on the probabilistic distribution represented by these normalized weights.

2.3.4 Publish Transform

Every time our particles are updated, either from forward propagation via the motion model or resampling via the sensor model, we determine the "average" particle pose. This is done by first taking the mean of the x and y position components of the pose and then calculating the circular mean of the orientation component. In order to calculate the circular mean of the orientation of the the pose, we use the formula:

$$\bar{\alpha} = atan2(\sum_{j=1}^{n} sin\alpha_j, \sum_{j=1}^{n} cos\alpha_j)$$

This formula gives us the average orientation of the pose. We then publish this transform as a new odometry message representing the current estimate of the robot's odometry in space.

2.3.5 Challenges in Implementation

A big challenge with implementation involves concurrent modification of the particles. Each subscriber runs in its own thread, but particles is shared between threads. This means that particles may be modified by one subscriber while another subscriber is trying to perform a computation on it. In order to prevent such a situation from happening, we use the multi-threading technique of locks to only allow one subscriber thread to modify particles at any particular time. By encapsulating any modification of particles with a lock, we guarantee atomicity for any change to the shared object.

3 Experimental Evaluation (Dylan and Anika)

In order to verify the validity of our implementation of Monte Carlo Localization (MCL), we tested our code both in simulation and on hardware. In this section, we discuss the results obtained from these experiments both qualitatively and quantitatively and compare them to the expected behavior and values.

3.1 Simulation Results (Dylan)

Before running our code on hardware, it is important to test on simulation in order to simplify algorithm tests, prevent unexpected damage to the robot, and to isolate hardware issues from software issues. In doing this, we tested several scenarios and modules:

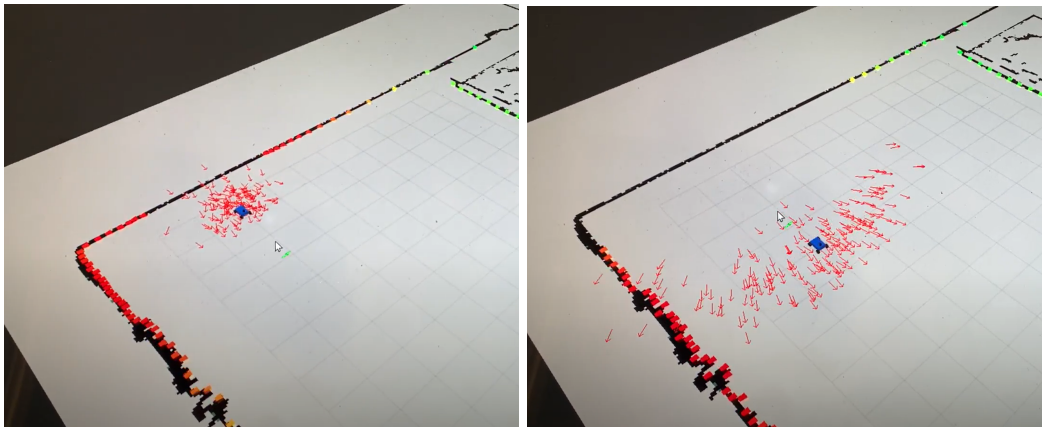
1. If we do not resample particles, does motion model behave as expected?
2. Does MCL converge and maintain correct values for position when the robot is stationary?

3. How robust is MCL under the influences of noise?

4. Does MCL work when the robot is moving around?

3.1.1 Motion Model Test (Dylan)

To answer the first of these bullet points, we temporarily disabled resampling in order to verify qualitatively that our motion model module was working correctly. When resampling is disabled, we expect the motion model to continuously spread particles out in an arc. This behavior is due to MCL not being called to reweight and resample the particles, meaning that the particles will not converge to the robot's location. Instead, when resampling is disabled, we see that the particles are initialized with random poses around the robot and are given equal probability weights. This causes them to move outwards from the car; as the car moves, this behavior appears as a curved "wave" of particles moving forward with the car. We verified this behavior in simulation and the result, which matches expectations, is shown in figures 2a and 2b.



(a) Particle spread at initialization. Note that the particles are initialized with random poses around the racecar.

(b) Particle spread after car moves forward. Note the arc-like spreading behavior of the particles; this is expected in the absence of resampling

Note that the sensor model module and MCL as a whole are not tested in this scenario, as sensor model is responsible for weighting the given particle distribution for future resampling. However, we can test our sensor model and overall MCL implementation by enabling sampling and running tests provided by the course staff.

3.1.2 Stationary Robot Test (Dylan)

After verifying our motion model and sensor model approaches through the previous approach and through the unit tests provided by course staff, we tested to make sure that our MCL code as a whole worked in simulation. We began by testing that the MCL code could give a stable position estimate while the car was stationary. The result of this experiment is shown below in Figure 3.

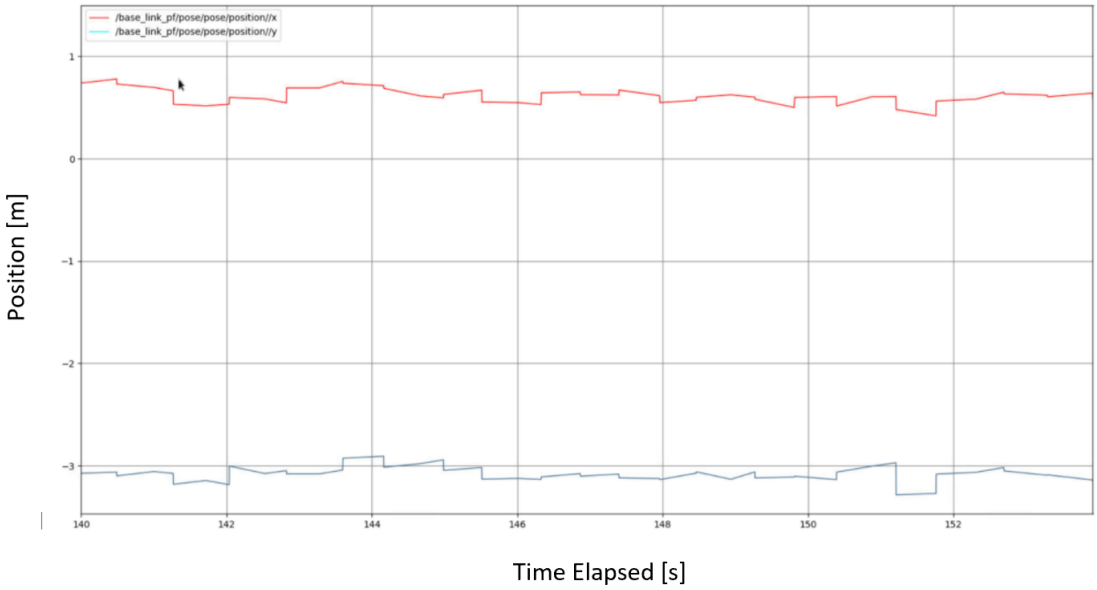


Figure 3: The results from implementing MCL on a stationary robot in simulation. The x position is red and the y position is blue. These values are the mean pose values of all Both the X and Y position appear centered around fixed values (about 0.75 and -3.2, respectively), indicating that the MCL algorithm is giving a stable estimate for the robot's position. Note that the square wave-like pattern is caused by randomly sampling for particle poses and subsequently resampling.

As we can see, the MCL algorithm gives a relatively steady estimate for the robot's X and Y coordinates. The values in Figure 3 are calculated from the mean pose of the particles. The values appear stable, indicating that the MCL algorithm is giving a stable estimate for the robot's pose. The square wave-like pattern in Figure 3 is due to randomly sampling for particles (causing the particle distribution to expand around the robot) and subsequently resampling (causing the particle distribution to contract around the robot) at each time step. This results in the mean pose of the particles changing when random sampling and resampling occur, giving the observed square wave pattern.

3.1.3 Noisy Odometry and Trajectory Following (Anika, Dylan)

With our MCL implementation verified to be effective, we tested our algorithm on the tests provided by staff. As shown in Figure 4, these tests measured our algorithm's performance in following a trajectory with no noise, some noise, and a lot of noise in the odometry readings.

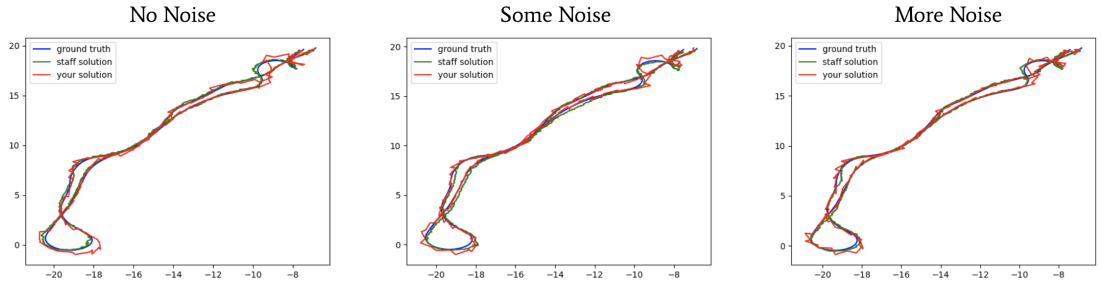


Figure 4: The results from implementing MCL on the staff tests.

Our MCL implementation had a score of 0.93 similarity with the staff code in the test with no noise, 0.95 similarity in the case with some noise, and 0.96 in the case with more noise. Clearly, our algorithm does better with noisier data. Fine tuning the amount of noise we add in the motion model could help our algorithm perform better in all three cases; however its performance is fairly high as is.

3.2 Hardware Results (Dylan)

While verifying our implementation in simulation is useful, the most important piece of the puzzle is getting the MCL algorithm working on the actual robot. To test MCL on the robot, we first recorded rosbag data of the robot moving (manually driven) in a sinusoidal wave-like motion along a wall. We then ran our MCL code on the rosbag data obtained from this to verify that the mean pose of the partcicles over time matched the trajectory of the robot. Figure 5 shows the plotted mean pose obtained from the particles over the course of the rosbag.

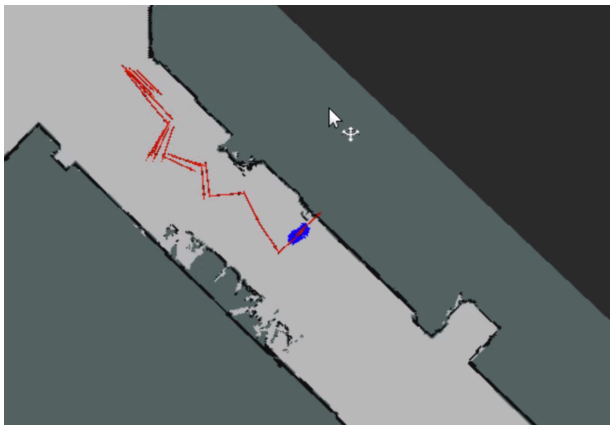


Figure 5: The results from implementing MCL on rosbag recorded from hardware.

While the result is not a perfect sinusoidal wave, we can clearly see that the mean pose estimates

calculated by the MCL code match up with the actual movement of the robot, thus verifying that our MCL implementation is effective on hardware. A video showing the robot being driven in real life as well as a video showing the implementation of the MCL code on the rosbag data can be found in the appendix.

4 Conclusion (Aileen)

We achieved the primary objective of this lab by implementing MCL to work in both simulation and on a physical robot. In order to do so, we separated MCL into three components: the motion model, the sensor model, and the particle filter.
First, we built a motion model and incorporated randomness in simulation to account for noise in actual data. Second, we developed a sensor model that would precompute a discretized sensor table for efficient look-ups and squash probabilities of particles passed into the table. Finally, we combined both of these in the particle filter, where we paid close attention to concurrent modification errors from the callback function. After testing all of this in simulation, we were then able to integrate with the hardware implementation to see the robot moving with localization in the Stata basement.

Given the non-modular nature of this lab, we turned to a few other resources in order to collaborate more effectively. In particular, we made use of the Live Share feature in Visual Studio Code that allowed us to modify code in real time in a collaborative work space. This gave each member the opportunity to address issues and ask questions in real time, furthering our knowledge of the algorithm and code as a whole.

Moving forward, there remain a couple of areas of improvement which we would like to build upon in the final project. First, we would like to tune the parameters of the sensor model to achieve higher efficiency in converging to the accurate robot state. Second, we would like to experiment with various number of particles initialized in order to determine the optimal balance between a sufficiently large number of particles to produce a good sampling space and efficient code for every time step. Third, we would like to address areas in our code where optimization may potentially allow us to publish at a faster rate and become more responsive to changes in the environment. Fourth, we would like to find a way to address the issue of picking an average mode in a multi-modal distribution (as is the case for choosing an average pose).

5 Lessons Learned

5.1 Aileen

While some labs are modular in their inherent structure (such as the visualization lab), this lab did not have a defined sense of structure. Each member worked on every part of the implementation and this resulted in a broad familiarity with the Monte Carlo Localization algorithm. While this is the optimal way to approach the lab in terms of maximizing knowledge per person, it was not the most efficient way of completing the lab. It would have been useful to define roles and modules early on, especially to parallelize tasks and prevent members of the team from blocking one another.

5.2 Matt

This lab was rigorous and challenged me to learn quickly and adapt to a new collaboration style. From the technical side, localization was a difficult topic for me to grasp but was fascinating at the same time. I found myself stuck many times as I worked with my team to map out the MCL algorithm into working code, but by leaning on each other and the time-generous TAs, we were able to complete many of the lab's challenges. To reflect on my communication and collaboration, I felt that the whole team is much more comfortable with each other, so there was a bit less structure in our meetings, which was both a good and bad thing for productivity and effectiveness. This lack of structure was also caused by the lab having less of a modular set up. We often found ourselves working on the same problem all at once, which sometimes slowed progress.

5.3 Anika

As my teammates have mentioned, Lab 5 was challenging in both its technical and communication components. Our team is quite comfortable working with each other and we've learn effective ways to divvy up work in modular labs. However, as this lab could not really be split up into multiple components that different people could work on. Therefore, we initially struggled with a way to make good collaboration happen. Eventually, we used a tool called LiveShare which helped us work on the same code together. The lab was also very technically challenging as the topic of localization was new to all of us. Truly understanding MCL, and how the motion and sensor models fit in with MCL, took quite a bit of time. However, understanding localization and how complex localization can be was very fruitful.

5.4 Quincy

This lab was the most challenging yet. This lab was logistically difficult as the lab was less modular outside of the motion and sensor models. This lead to work/productivity being bottle-necked as a result of everyone waiting on the same sections of a lab to be completed. We also ran into a plethora of technical difficulties that required a big team effort to resolve. Overall, the lab was extremely informative and also further reinforced the importance of teamwork and asking for help when necessary.

5.5 Dylan

This lab was challenging for us both technically and logistically. From the technical side, I learned plenty about MCL in general along with some new coding tips. However, the entire time it felt like an uphill battle, which I feel has challenged me to adapt my way of thinking in order to help debug and puzzle through the assignment. From a logistical standpoint, I feel that we struggled at first after trying to divvy up tasks, only to find that this lab required a lot of intercommunication and teamwork. This resulted in us changing up our workflow to account for the inability to really work on different tasks in parallel. While this was challenging, I feel that this has increased our cohesiveness as a team and taught us about adapting how we work both individually and as a team to different scenarios.

6 Appendix

YouTube links to the videos for MCL running on data from a rosbag recorded on hardware can be found at the following links:

Recording of robot: https://youtu.be/l0n1wyS9GUo

MCL running on rosbag data obtained from robot: https://www.youtube.com/shorts/3slq5SXMDAI