Lab #5 Report: Monte Carlo Localization

Team #1

Kyri Chen
Alex Ellison
Ashley Holton
Jordan Ren
Laura Rosado (editor)

6.141: RSS

April 5, 2022

1 Introduction (Ashley)

One of the most important functionalities any robot trying to move through the world must have is the ability to know where it is located. This may seem like an easy problem at first, but it can prove to be incredibly complex and is still an area of active research today. The problem mainly stems from the fact that reference points are needed to anchor oneself in the world, as well as the need for a way to measure how one is moving over space and time. There is also the problem that rears its head in many robotics problem - that being noise and uncertainty. One must be able to integrate the reality of noisy sensor data into any framework of localization.

The goal of this lab is to create a framework that can determine the robot's position at any time, given a map of the environment it is in. We aim to achieve this goal by breaking up the problem into two parts: first, being able to determine how we are moving, and second, being able to use lidar data to give us information about the environment around the robot. This will be achieved by implementing the method of Monte Carlo localization. The two parts of the problem will be encapsulated in the motion model and sensor model, with a particle filter that will incorporate the information gained by both models in order to give a prediction of the robot's estimated pose. We will first test our localization framework in simulation before then transferring it over to the robot, making any necessary changes to increase robustness of the framework along the way.

2 Technical Approach

In the following subsections, we will outline the multiple pieces we developed in our localization system to output the location of our robot in a given map of the environment. To start we will share a high level overview of how each piece was integrated together to achieve our localization goal, then we will step through each subcomponent of the system in more detail.

2.1 Particle Filter (Alex)

This section will go over the components that make up the particle filter and how they integrate to serve as a localization method for our robot. Before going into each of the steps we took to complete the filter, however, we will first clarify the goals for the system. The idea behind the particle filter is to randomly spread particles over a known map, then update their poses utilizing a motion and sensor model as we receive more data in order to localize the position of a robot in the environment. In other words, we want to pinpoint the location of our robot in a known map. To do this, we split our algorithm into four main steps which are outlined in more detail below.

1. **Motion Model** - As mentioned earlier, we started with a set of particles randomly generated by first inputting an estimated robot pose on our map, and then spawning 200 particles with Gaussian noise around that pose. The random particle generation process gives us a point cloud around some location on the map with estimated x and y locations and angular bearing. From here, the first step in our system utilized a motion model to update these particles based off odometry data from the robot to move one time step into the future while also adding noise to eliminate all particles moving in the same way deterministically (this will be explored further when we discuss resampling).

2. **Sensor Model** - The next step in the process was to utilize our sensor model which took into account lidar data to compute the likelihood that our perceived measurements were taken from each of the particles we updated in the last step. These likelihoods represented a distribution of which particles were most likely to be correct.

3. **Resampling Filter** - From the probabilities calculated in the previous step, we then resampled our particles by drawing from the original particles, weighting the likelihood one is redrawn by the probability that it is where the sensor measurement is from. Ideally, this functioned to remove outlier points with small probabilities of being correct and condensed our cluster of particles around what we expect to be correct. Visually, we were able to see the condensation of our particles as desired.

4. **Publish Average Pose** - Finally, from our resampled particles, we took an average of the x, y, and theta values and returned the average pose. This is what we expected the location of our robot to be on the map.

Now that we have outlined how each of these pieces are integrated together, we will go into more detail about the two main sub-components, the motion model and the sensor model.

2.2 Sub-components

2.2.1 Motion Model (Laura)

The first main component of particle filter is the motion model, which is able to predict the future position of particles based on the instantaneous linear and angular velocity reported by the odometry data. The racecar is modeled as a rigid body moving in a 2D plane, which decreases the complexity of the necessary calculations. When deployed as a component of the particle filter, the motion model is able to handle an $N \times 3$ matrix of particle positions. For simplicity, we will present our calculations as if we are predicting the position of a single particle.

From the given particle position at some time k, $\boldsymbol{x_k} = [x_k, y_k, \theta_k]'$, a transform matrix is constructed of the form

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R}_k & \boldsymbol{p}_k \\ 0 & 1 \end{bmatrix}$$

where $\boldsymbol{R}$ is a two-dimensional rotation matrix of the form

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{bmatrix}$$

and $\boldsymbol{p}$ is a position vector.

$$\boldsymbol{p} = \begin{bmatrix} x_k \\ y_k \end{bmatrix}$$

By multiplying the transform matrix by the odometry data, $\boldsymbol{\Delta x} = [dx, dy, d\theta]'$, we obtain the position of the particle at time $k+1$.

$$\boldsymbol{T} \times \boldsymbol{\Delta x} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \\ 1 \end{bmatrix}$$

To obtain the value of θ_{k+1}, we simply sum the current θ and $d\theta$, which is given as the last entry in the odometry data.

$$\theta_{k+1} = \theta_k + d\theta$$

Putting it all together, we find the predicted, or updated, position of the particle in the next time step.

$$\boldsymbol{x}_{k+1} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix}$$

This process can be applied to N number of particle positions at any given time step. In practice, the method of collecting and reporting position data will be imperfect due to sensor uncertainties and other hardware factors. To account for this, noise is added to the final positions according to a normal distribution. By "fighting noise with noise" we have a more robust model to track the robot's motion.

2.2.2 Sensor Model (Kyri)

The second component of the particle filter methodology is the sensor model. The sensor model, denoted $p(z_k|x_k, m)$, defines how likely it is to record a given sensor reading z_k from a hypothesis position x_k in a known, static map m at time k. While the motion model is used to spread the hypothesis positions apart, the sensor model is used to filter out particles that deviate too far from where the robot truly is in the map. Map distances from each particle are calculated and compared with ground-truth sensor measurements, and particles are only kept if the two values are consistent.

When using a laser scanner, the likelihood of a sensor reading is given by the product of the likelihoods of each individual range measurement:

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

For each range measurement i, the likelihood is composed of four distributions (detailed in the Appendix), each weighted by the parameters $\alpha_{hit}, \alpha_{short}, \alpha_{max}, \alpha_{rand}$:

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

Since computation speed is critical in the ROS implementation of MCL, we precomputed a lookup table to enable quickly finding the probabilities of measuring any discrete range $z_k^{(i)}$ for all ground-truth ranges d. Indexing the lookup table is much faster than computing the probabilities after each laser scan measurement. Borrowing the values of the hyperparameters from Part A of the lab, we calculated all the values in the table using $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, $\alpha_{rand} = 0.12$, and $\sigma_{hit} = 8.0$. After populating the table, we normalized the probabilities across each value of d to ensure that they summed to 1.

Next, we move on to calculating the likelihood of each particle in the map. In the lab, we were given a C++ implementation of ray casting, or calculating simulated laser scans from each particle, to quickly generate ground-truth distances to be used to compute likelihoods. However, before we're able to calculate the likelihoods, we must do some pre-processing.

First, we need to convert the distances in our laser scan observations and the ray casting results from meters to pixels. This brings our measurements from real life into the scale of the Stata basement map that is given to use. Next, in order to guarantee that all measurement distances can be indexed in the pre-computed lookup distances, we round all measurements to the nearest integer and clip all measurements to fall between 0 and the maximum distance z_{max}. Lastly, since there are many range measurements in the laser scan message and all likelihoods are less than 1, including every single range measurement will drive the final scan likelihood extremely close to 0 while also increasing the computation time. Therefore, we will randomly downsample a smaller number of range measurements to include in each particle's likelihood calculation.

Once we have the likelihood of all the particles, we wish to squash the likelihood distribution across particle locations. To examine the intuition behind this, we examine Figure 1.

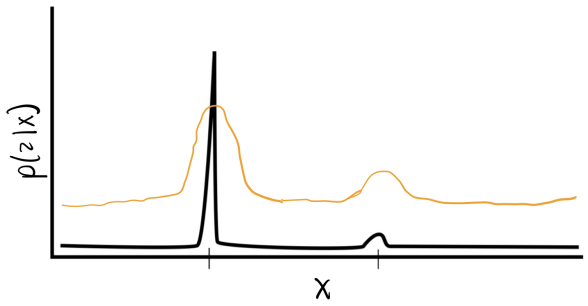
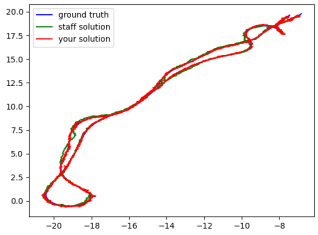


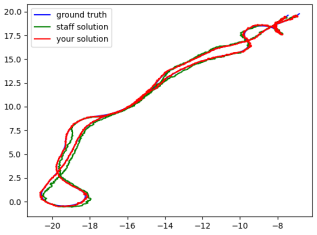
Figure 1: True distribution in black, squashed distribution in orange

The black line in the figure represents the sensor model of a very accurate sensor - the likelihood of a particle having rayscan data that matches the sensor measurement is highly peaked around the true location of the robot. However, if we imagine the motion model distributes the particles uniformly across all possible locations, then it is very unlikely that very many (if any) of the particles will fall under the slim peak of the model. It is actually more likely that a particle will fall under the second, wider peak. After the resampling step, it is possible that very few particles surrounding the first peak are successfully sampled, which would throw off future iterations of the localization algorithm.

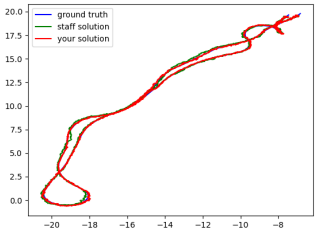
To fix this problem, we can "squash" the distribution model by taking all of the probabilities to some power $\frac{1}{p}$ less than 1. Doing this essentially decreases the "confidence" we have in the sensor, as p observations are now needed to obtain the likelihood distribution that 1 observation previously did. This gives us the orange line in the figure, and it is easy to see that more particles will now fall under the first, correct peak. This squashing makes sure enough particles



(a) No odometry noise. Time-average deviation of 0.268 meters from the ground truth.



(b) Some odometry noise. Time-average deviation of 0.267 meters from the ground truth.



(c) More odometry noise. Time-average deviation of 0.267 meters from the ground truth.

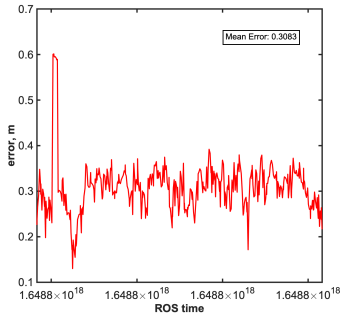
Figure 2: Graphs generated from submitting our code to the Gradescope autograder. Across all scenarios, our particle filter had an average deviation of $0.065cm$ from the staff solution.

are sampled from the true location of the robot and makes the algorithm much more robust.

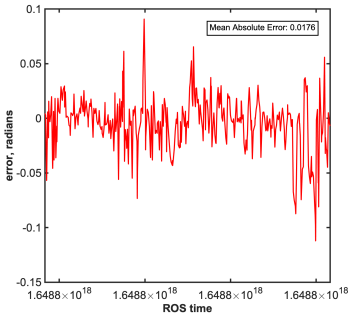
3 Experimental Evaluation (Jordan)

To quantify the performance of our particle filter, we first uploaded our code base to the course-provided autograder, which evaluates how well our particle filter tracks position while moving along a set trajectory. This performance was then compared to the ground truth as well as the staff solution. The results are shown in Figure 2.

The paths generated by the staff solution, our solution, and the actual path are juxtaposed in each graph. Our particle filter was tested under three different circumstances: receiving odometry data with no noise, some noise, and more noise. Our particle filter resulted in a time-average deviation of 0.268 meters, 0.267 meters, and 0.265 meters from the trajectory for the respective scenarios,



(a) Euclidean distance from predicted position to actual position.



(b) Difference in theta between actual and predicted measurements.

Figure 3: Error graphs collected while running our particle filter and wall-following algorithm through a simulated Stata basement environment. This course consisted of one turn.

compared to the staff solutions of 0.217 meters, 0.190 meters, and 0.199 meters. This performance gave us a baseline for what to expect while collecting further performance metrics in simulation.

3.1 Simulation Performance

When moving the particle filter onto the simulation we observed a similar performance to what we had on the server tests. As shown in Figure 3, the estimated pose had an average positional deviation from the car's actual position of 0.3083 meters, only 4 cm greater than the deviation reported from the autograder. The orientation of the car had an average deviation of 0.0176 radians. While the autograder did not report an equivalent error, this means on average the car thinks it is rotated by 1 degree from where it actually is. Relative to the size of the car, this is a small error that we deemed satisfactory.

3.2 Hardware Performance

In order to adjust the particle filter to work on hardware there were several changes that had to be made. First, since we had the Velodyne lidar the scans were collected in two time steps to create a full scan. Next, we needed to shift the scan data so that it is in the frame of the car, which required a $\frac{\pi}{3}$ shift of the ranges. Finally, we had to account for the extra noise that is always present on hardware by tuning our original noise parameters. Since there is no easy way to know the car's actual position and orientation relative to the map, we were only able to qualitatively asses the localization performance. From what we observed the predicted pose of the car matched the car's actual position on the map to a similar degree as in simulation.

4 Conclusion (Ashley)

In this lab, we successfully implemented a particle filter that utilizes both a motion model and a sensor model in order to accurately determine our robot's position, given a map of the environment it is in. Using the odometry and lidar data from the robot, we were able to make accurate estimates of our robot position. By adding noise and uncertainty to the data as we process it, we make our localization framework robust to the noisy data that is collected in real-world situations. We also created code that was efficient such that it could be run at high frequencies in order to provide the most up-to-date position estimates.

While our localization framework provided good position estimates of our robot, especially in simulation, there is still room for improvement, especially for when it is deployed on the robot. Though our observations of the car's localization seemed accurate, there was likely still a larger amount of error than on simulation. More work could be done to investigate why we see such a change when we bring our framework to the hardware. This could mean investigating more or different applications of noise and uncertainty, or more integration tests to see if the way the different pieces of our framework are interacting is causing any of the error we see.

5 Appendix (Kyri)

The likelihood formula in Section 2.2.2 is composed of four distributions that arise from a few cases:

1. Probability of detecting a known obstacle in the map.
2. Probability of a short measurement. Maybe due to internal lidar reflections, hitting parts of the vehicle itself, or other unknown obstacles.
3. Probability of a very large (aka missed) measurement. Usually due to lidar beams that hit an object with strange reflective properties and did not bounce back to the sensor.
4. Probability of a completely random measurement.

In the first case, if the measured range is $z_k^{(i)}$ and the ground truth range is determined (via ray casting on the map m from pose x_k) to be d then we have that:

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

In the second case, the likelihood is

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

The third case is represented by a large spike in probability at the maximal range value. When we discretize the distances in the lookup table, this can be represented as:

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

Lastly, the fourth case is represented by a small uniform value:

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

6 Lessons Learned

6.1 Kyri

Technically, I feel like this was the most challenging lab we've done this year. Beyond understanding the intuition and mathematics behind the MCL algorithm, there were a ton of optimizations, hacks, and tuning that needed to be done in order to get everything working. One standout topic was working more efficiently with numpy arrays instead of Python lists or for-loops. Also, since this lab had a lot of moving parts that relied on one another, our team really had to emphasize communication and collaboration as we pieced together each other's code. I think we did a great job of this, and we all fully understood the entire lab despite only having expertise over the implementation of parts of it. Since the majority of the lab took place over spring break and a swim meet that 4/5 of our group participated in, we could've potentially done a better job of pacing out our workload, but we still figured out everything efficiently once we returned.

6.2 Ashley

I learned a lot about localization through this lab, along with all the considerations you have to put into a task like that. I also feel like this lab highlighted the importance of having an execution plan for how to approach a project like this, since a lot of the work we did was very last-minute and we probably would've benefited more from having a more concrete game-plan going into it.

6.3 Laura

This lab was daunting at first, but it was very rewarding to see all the components come together in the end. While the high-level concepts were relatively simple to understand, integrating it together proved to be challenging. It seems that there was always a hardware quirk standing in our way of debugging on the racecar. Evaluating our performance was also confusing – we achieved an acceptable score on the autograder, but struggled to see similar performance in simulation. A little more guidance on common issues would have been helpful in

that case. I think we did well communicating and working together as a team; of course, starting earlier would have helped improve our final performance, but all things considered I'm proud of the work we did.

6.4 Jordan

This lab has been the most conceptually difficult but most rewarding lab we have done yet. From a technical standpoint I feel as if getting a deep understanding of the material before coding was extremely helpful, as it made debugging much more simple. I also learned that making good use of numpy arrays is necessary for computing time-sensitive information. In terms of communication I think that we could have created a better plan to get work done efficiently and on time. This would have made the past week less stressful and could have resulted in better performance on our robot.

6.5 Alex

This week was a lesson in self-reflection and knowing when to ask for help. Up until this point I have felt fully prepared to get by with each of the labs without any substantial help from instructors and TAs outside of lecture, but this week while working on the particle filter I reached a point where I had no clue what else to try in order to get our code to work and spent far too long trying to debug myself before swallowing my pride and getting help from a TA. After getting help, we found that it was a problem of efficiency due to us publishing test information and logging. Looking back, I think I learned a valuable lesson in communicating with TAs more regularly and effectively. On the technical side, I learned a lot about utilizing numpy for more efficiently manipulating data. This proved to be pivotal to our success in making our system work.