Lab # 5 Report: Localization
Using Monte Carlo Localization to determine a robot's orientation and position in a known environment

Team 3

Wing Lam (Nicole) Chan
Shanti Mickens
Magnus-Tryggvi Kosoko-Thoroddsen
Sheng Huang
Jessica Zhang

RSS

April 14, 2023

1 Introduction - Nicole, Shanti

In Lab 5, our team implemented Monte Carlo Localization (MCL) to determine our racecar's orientation and position in a known environment. In previous labs, we implemented wall following and line following to allow our racecar to move around autonomously. From there, localization was a key next step since as our racecar drives around by following a wall or a marked line on the ground, it needed to be able to locate and keep track of its real-time position and orientation in the world for navigation purposes.

MCL starts with a set number of particles, where each represents a possible state or hypothesis of where the racecar is and its orientation. Then, as the racecar moves around and senses its environment, the algorithm continuously updates the particles to predict the racecar's new state. Ultimately, the particles converge toward the racecar's actual position. We tested our MCL algorithm in simulation and real life by comparing the racecar's pose estimation with the ground truth position. The goal for this lab was for the racecar to be able to predict its location and orientation accurately, within 0.2m, and efficiently, outputting position estimates at a rate of at least 20Hz.

2 Technical Approach - Sheng, Jessica, Magnus

To implement MCL, there are two key sub-modules. First is the sensor model, which uses the known map and LIDAR data to update the particles probabilities or likelihood of being correct. Second is the motion model, which uses odometry data from the racecar's encoders to update the particles positions. Lastly, the MCL algorithm resamples the particles and averages the particles' poses, before finally publishing that transform as the estimate of the racecar's pose. This section will give an overview of the motion model, the sensor model, and then how these come together to form the overall MCL system or particle filter.

2.1 Motion Model

The motion model mainly works with "poses," one-dimensional vectors containing a relative x-position, y-position, and heading of the robot (represented in meters, meters, and θ respectively).

$$x_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} \tag{1}$$

The motion model takes in a particle's old pose, x^m_{k-1}, and the odometry data, and returns the particle's updated pose, x^m_k. However, the odometry data is given with respect to the robot's frame while the particle pose needs to be with respect to the map's frame. So, the odometry data first needs to be converted into the map frame, which can be done using the corresponding rotation matrix:

$$R^m_r = \begin{pmatrix} cos(\theta_k) & -sin(\theta_k) & 0 \\ cos(\theta_k) & sin(\theta_k) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2}$$

The odometry data with respect to the map can then be calculated by multiplying the rotation matrix with the odometry data with respect to the robot.

$$\Delta X = \begin{pmatrix} \Delta x_m \\ \Delta y_m \\ \Delta \theta_m \end{pmatrix} = \begin{pmatrix} cos(\theta_k) & -sin(\theta_k) & 0 \\ cos(\theta_k) & sin(\theta_k) & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \Delta x_r \\ \Delta y_r \\ \Delta \theta_r \end{pmatrix} \tag{3}$$

However, Monte Carlo Localization also needs some randomization in order to work, which can be achieved by adding noise to the odometry data. We decided to model the noise $\tilde{X}$ with a Gaussian distribution using 0 as the mean and 0.05, 0.05, and 0.076 as the standard deviations for the x, y, and θ components respectively. These standard deviations were decided upon empirically through testing in the simulator. Finally, the motion model adds together the old particle position, the odometry data with respect to the map frame, and the randomly sampled noise to get the new particle position.

$$x^m_k = x^m_{k-1} + \Delta X + \tilde{X} \tag{4}$$

2.2 Sensor Model

The sensor model takes in an array of particles, a LIDAR scan, and a map, and returns an array of the likelihood for each of the given particles, where likelihood is defined as the probability that the particle's pose is equal to the car's current pose. To model the likelihood of a particle, the sensor model first performs ray casting, which models a LIDAR scan from the particle's pose, and computes the likelihood of each measurement in the modeled scan by comparing it with the actual scan from the car's LIDAR sensor.

The likelihood model for each measurement considers several cases:

1. The probability of detecting a known obstacle on the map. This is modeled as follows, where z_k is the measured distance, and d is the distance determined from ray casting:

$$p_{hit}(z_k, d) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(z_k - d)^2}{2\sigma^2}) & \text{if } 0 \leq z_k \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

2. The probability of a short measurement, possibly due to hitting unknown obstacles or internal LIDAR reflections. This is modeled using:

$$p_{short}(z_k, d) = \frac{2}{d} \begin{cases} 1 - \frac{z_k}{d} & \text{if } 0 \leq z_k \leq d \text{ and } d \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

3. The probability of a very large measurement, which is usually due to a missed measurement when LIDAR beams do not bounce back to the sensor. This is modeled by:

$$p_{max}(z_k, d) = \begin{cases} \frac{1}{\epsilon} & \text{if } z_{max} - \epsilon \leq z_k \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

4. The probability of a completely random measurement, which is modeled by:

$$p_{rand}(z_k, d) = \begin{cases} \frac{1}{z_{max}} & \text{if } 0 \leq z_k \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

To compute the final likelihood of a measurement, the sensor model combines the four distributions using a weighted average, where the weights are defined by the parameters α_{hit}, α_{short}, α_{max}, and α_{rand}, which we have set to 0.74, 0.07, 0.07, and 0.12 respectively.

$$p(z_k, d) = \alpha_{hit} * p_{hit}(z_k, d) + \alpha_{short} * p_{short}(z_k, d) + \alpha_{max} * p_{max}(z_k, d) + \alpha_{rand} * p_{rand}(z_k, d) \tag{9}$$

However, computing these probabilities for each measurement every time is slow, especially since each ray casting returns an array of several measurements, and the given particle array will also need to be large. So, we built a precomputation table of the probabilities for quick lookup after getting results from ray casting.

2.2.1 Precomputing Probability Table

To build a precomputed probability table, first the LIDAR ranges need to be discretized. We used a table width of 201 as we found that it gave enough granularity to get good results. Notably, the distributions listed in equations 5-8 can be used with any scale, so we decided to convert all the LIDAR measurements from meters to pixels for easier calculations.

To fill in the table, the sensor model loops through all distances from 0 to 200 for both z_k and d, and calculates p_{hit}, p_{short}, p_{max}, and p_{rand} as defined in equations 5-8. But before we can find the final $p(z_k, d)$ values, the p_{hit} values need to be normalized across the d values. Equation 5 is intended for use over continuous ranges, however, since the range has been discretized, we need to normalize the distribution so all the probabilities sum to 1. After normalizing the p_{hit} values, $p(z_k, d)$ can be calculated using equation 9. Finally the $p(z_k, d)$ values are normalized across the d values so they sum to 1.0, before being saved in the table.

2.2.2 Evaluating Particles

After precomputing the probability table, evaluating the probabilities for an array of particles becomes very simple and fast. First, the particles are passed into the ray casting function, which simulates a LIDAR scan from each particle's position by shooting beams at set angle increments and returning the distance when a beam hits an obstacle on the given map. Next, the actual LIDAR scan from the car's sensor needs to be downsampled to match the number of beams that ray casting returns. This is done by sampling every $1/num\_beams\_per\_particle$ measurement of the real LIDAR scan. Finally, the measurements from both the model LIDAR scans and the actual LIDAR scan need to be converted from meters to pixels, to match the scale used in the precomputation table. Now, for every beam in a particle's scan, the sensor model can simply look up the probability of $p(\text{model LIDAR scan measurement}, \text{actual LIDAR scan measurement})$ in the probability table. To get the probability for a particle, the sensor model multiplies together the probabilities for every beam of the particle's modeled scan. Finally, we have decided to squash all the resulting probabilities by raising them to the power of $\frac{1}{2.2}$ to make the distribution less peaked. The sensor model repeats this process for every particle, and then returns the array of probabilities for each particle.

2.3 Particle Filter

The particle filter combines the motion model and sensor model to implement Monte Carlo Localization (MCL). It keeps an array of particles that get updated on every iteration of MCL, getting updates from the

motion model as the car moves, and updates from the sensor model to resample the most probable particles based on the current laser scan. The filtering process is as follows: for a set of particles initialized at the beginning of localization,

1. Randomly distribute the particles around the initialization position.
2. Whenever the robot gets new odometry readings, update the particles using the motion model. Whenever the robot gets new LIDAR readings, update the laser scan saved in the sensor model.
3. Use the sensor model to get new probabilities for each particle.
4. Normalize the probabilities from the sensor model and resample the particles based on the normalized probabilities.
5. Average all the particles, using the arithemtic average for the x and y components, and the circular average for the angle. Publish this average as the estimate of the car's location.
6. Repeat steps 2-5.

The filtering process works in real-time, updating, and resampling the most probable particles as the car moves through the map. Even with just 200 particles, the average from these particles is able to estimate the car's position on the map pretty well.

The final piece of our MCL algorithm is deciding how to initialize the particles. We have implemented 2 different ways of initializing the car's estimated position on the map. First we have set an optional ROS param for the car's initial position on the map, which is set to the default of [0, 0, 0], which is roughly the center of the intersection in the stata basement with the orange circle, facing southwest. We have also set up the option to initialize the car's position using the 2D pose feature, where the initialpose_pub.py script can be used to publish an [x, y, theta] that the car will use to initalize its position.

3 Experimental Evaluation - Magnus, Nicole

3.1 Testing Procedure in Simulation

In order to test our particle filter in simulation, we ran the localization code alongside our wall-follower and parking simulator code in RVIZ. The goal is for the car to generate the correct pose as it follows the cone in simulation, which demonstrates that it is able to localize as it moves.

3.2 Results in Simulation

First, we ran our previous wall-follower code from Lab 3 simultaneously with the particle filter. A video of this simultaneous performance is included in our briefing and at this link. In Fig. 1 as well as in the video, we see that the car is able to accurately determine its pose as it follows the wall. So, we know that localization is accurate even as the racecar is moving.

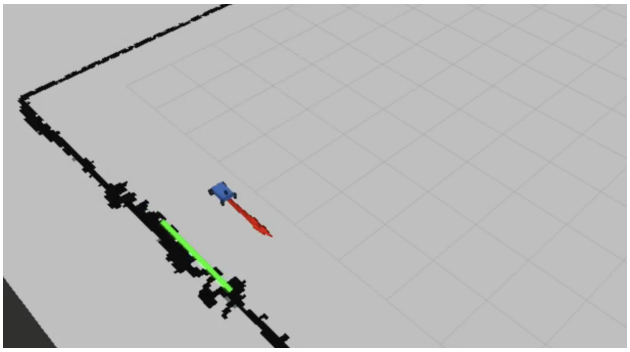

Fig. 1. The green line represents the car's current linear regression of the wall from the wall-follower procedure while the red arrow marker represents the car's estimated pose. We can see that it accurately estimated the car's position and heading.

In another testing scenario, we ran our previous parking controller code from Lab 4 simultaneously with the particle filter. A video of this performance is included in our briefing and at this link. In Fig. 2 and the linked video, we see that as the racecar approaches and turns towards the cone marker, the red marker correctly estimates the car's pose consistently.

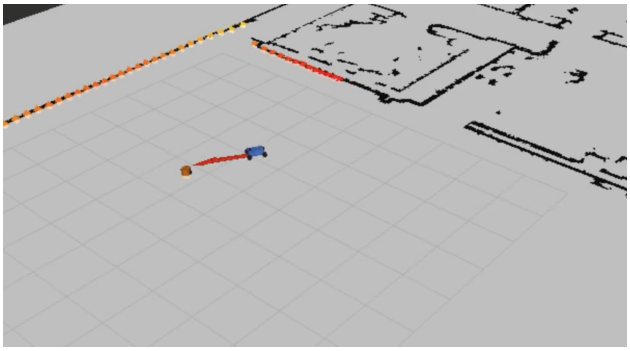

Fig. 2. The localization marker correctly estimates the racecar's pose as the it parks in front of the cone in simulation.

To provide numerical results, we also calculated the error of the pose estimate from the particle filter with the ground truth of the car's location. We first found the ground truth transform and converted the quaternion coordinates into euler coordinates. Then, we calculated the error and published it as a topic for measurement. Fig. 3 is a snapshot of visualizing the racecar parking in front of the cone while a plot of the published estimation error is generated in the bottom right corner. For convenience, Fig. 4 is a close-up of that plot. As we can see, there is almost no error in the x-position and heading estimates and an acceptable amount of error in the y-position estimates.

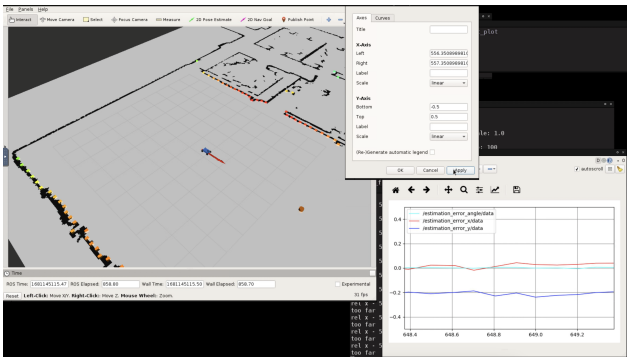

Fig. 3. We generated error plots by comparing the racecar's true pose with the pose from MCL.

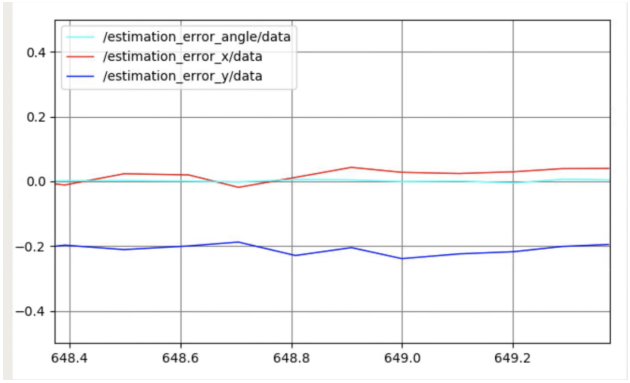

Fig. 4. The error plot shows little to no error in the racecar's pose estimation in x and y position as well as heading angle.

3.3 Testing Procedure on Racecar

In order to test the localization on the racecar, we placed the car in the Stata Basement in accordance with the map given in lecture or lab instructions. We determined the origin of the world frame as an actual location in the basement corridor, then began our tests by placing the car in that location. We ran the localization code as one of our teammates drove the racecar manually with the controller. In RVIZ, we were able to visualize the car's movements as well as its estimated pose as it is moving in the environment. We present our findings in the next section.

3.4 Results on Racecar Localization Performance

Through qualitative analysis from the videos provided here and here, the car can sufficiently locate itself in the Stata Basement given a map. As shown in Fig. 5, the side camera view is for personal confirmation that the car is moving and locating itself in real time. The odometry arrow attached to the car is the actual estimate from the particle filter's simultaneous processing. One thing to note is that when moving backwards, the *LaserScan* data is centered on the estimated odometry of the car, but it does not line up with the actual map, which is indicative of lag. There may be edge cases where backwards motion is not correctly represented in the motion model.

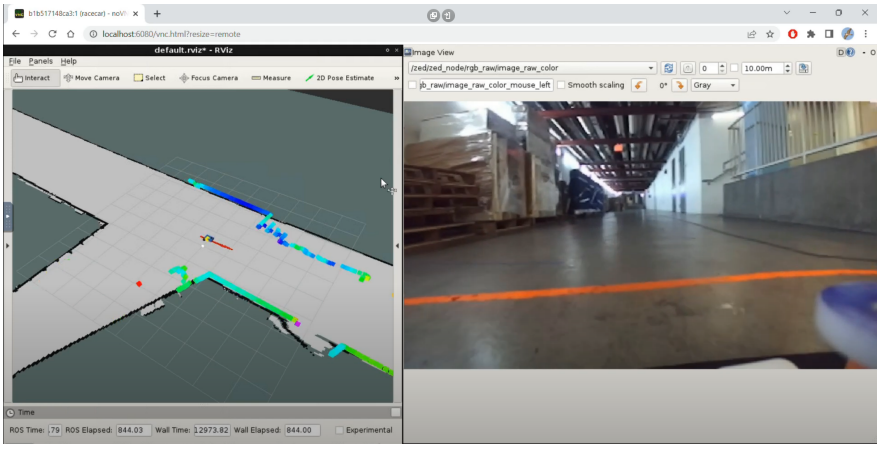

Fig. 5. The real-life localization view shows the map with the car's estimated odometry on the left hand side, and the ZED camera view on the right hand side, confirming localization is sufficient with the implemented particle filter.

4 Conclusion - Shanti

Over the course of Lab 5, we have successfully implemented an MCL algorithm to allow our racecar to track its orientation and position in a known environment as it drives around in both simulation and on the racecar. In the future, we would like to do more work to increase the odometry publish rate from our MCL implementation to allow for more accurate and real-time tracking of the position of our racecar, especially at fast speeds, up to 10m/s, which will be used in the final challenge. In addition, we want to investigate our motion model further to debug the lag we noted in the estimated odometry of the racecar when it is traveling backward. In Lab 5, we also learned about the importance of time management and dividing up tasks to ensure work gets done on time.

In the next design phase, we will be implementing path planning to output a trajectory from our current position to an end goal position and a pure pursuit controller to output driving commands to steer our racecar along that trajectory to the end goal position.

5 Lessons Learned

Below we each included our own self-reflection on the technical, communication, and collaboration lessons we learned in the course of this lab.

5.1 Nicole

I learned about all the intricate parts that go into a working particle filter and how delicate the models can be. Also, it was cool to see localization working in real life after learning about the mathematical methods behind its implementation. Although we ran into hardware issues, we were able to compile the results for the report despite not being included in the briefing.

5.2 Shanti

I learned about all the components of Monte Carlo Localization and then was able to implement it with my team. More specifically, I got to learn about the importance of thinking through what frames data is defined between and how crucial visualizing your data is for debugging. In addition, I learned how to include hyperlinks in a Latex document, and I noticed how with larger labs, it's very helpful to divide and conquer by having some team members focus on different submodules of the overall lab.

5.3 Magnus

Learning about the high-level strategy and low-level data transformations necessary for Monte Carlo Localization gave me a stronger understanding of what considerations are necessary for localization, i.e. proper reference frames, frame transformations, particle initialization and distribution, etc. This learning process also highlighted the importance of splitting up modules between team members, but also ensuring all of us had an adequate understanding of what the code was doing, as well as the higher-level concepts of the sensor and motion models, as well as the particle filter.

5.4 Sheng

I have learned the steps of how the robot think about each particle and assign each particle a probability based on its distance using the different p_value calculations that was given in the lab.

It was also interesting to code up the transformation matrix for the robot to transform the odometry data to the actual robot's position with relative to world frame and see it work in real life.

5.5 Jessica

I learned about all the steps of the Monte Carlo Localization and how adding randomness can help a model learn things about the real world. I also learned how to integrate several modules together, as we had seperate publishers/subscribers for the motion model, sensor model, localization, and the transforms needed to visualize everything in rviz.