Lab #5 Report: Monte Carlo Localization

Team #8

German Espinosa
Habeeb Salau
Izzie Torres
Angelina Zhang

6.141: RSS

April 6, 2022

1 Introduction [Angelina]

In this week's lab, we implement Monte Carlo Localization: we determine our robot's orientation and position in a known environment. Monte Carlo Localization predicts a set of particles using odometry data, and resamples the data using weights calculated with sensor data. A key feature is that Monte Carlo Localization models uncertainty in robot perception, such as wheel slip and sensor errors, with noise modeled as a random variable. In doing so, the robot's pose becomes a random variable and we use a probability distribution to estimate pose. In this lab, we use a map of the environment as well as sensor and odometry data to ultimately calculate our robot's pose with respect to the Stata basement.

In Labs 3 and 4, we have successfully implemented wall-following and visual servoing models. Our car is able to drive a fixed distance away from a wall, park in front of a colored cone, and follow a line. In Lab 5, we implement mostly new functions, but draw upon our wall-following model for testing purposes.

There are three main building blocks in our technical approach, the first one being a motion model. We subscribe to the car's ground-truth odometry data and apply random noise to provide estimates for its pose. The motion model outputs an array of particle data, containing information about the robot's coordinates with respect to the map, as well as its angle. The particle array is essentially a collection of predictions for the robot's orientation and position. The next module is a sensor model, which assigns likelihood weights to the motion model's particle data. The last module, a particle filter, combines outputs

from the motion and sensor models to implement Monte Carlo Localization. After initializing the robot's pose on a map, our model uses the previous two functions to updates particle positions upon receiving sensor or odometry data. After updating the particle positions, we take the average of the particle data and set that as our estimated robot pose. We then publish that pose and the corresponding transform with respect to the map.

With the conclusion of this lab, we achieve an internal model of the external world. In future sections, we further elaborate on our technical approach and experimental evaluation methods.

2 Technical Approach

2.1 Motion Model [German]

The motion model portion of our localization plan includes two main steps: computing the current pose of a particle and adding noise to the data. The use of noise allows us to make our code non-deterministic which is essential with regard to the overall localization algorithm. Throughout the next few sections, we shall cover the methods used to create a functional and efficient motion model:

2.1.1 Compute Current Pose

We are given several particles, each with a pose that can be described by: $x_k = [x, y, \theta]^T$. In addition, we can directly subscribe to the vehicle which gives us constant and accurate odometry data in the following format: $\Delta x^k = [dx, dy, d\theta]^T$. The first step in our process is to compose a transformation matrix which brings both the odometry and particle pose within the same frame of reference:

$$T_k^W = \begin{bmatrix} R_k^W & p^k \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} cos(x_k[2]) & -sin(x_k[2]) & x_k[0] \\ sin(x_k[2]) & cos(x_k[2]) & x_k[1] \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & x \\ sin(\theta) & cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}$$

Once we have our transformation matrix, we can simply multiply our odometry data by the transformation matrix to get our translational change:

$$x_W = T_k^W * \Delta x^k = \begin{bmatrix} cos(\theta) & -sin(\theta) & x_k[0] \\ sin(\theta) & cos(\theta) & x_k[1] \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} dx \\ dy \\ d\theta \end{bmatrix}$$

Since we can directly apply rotational change to our orientation, we only pull out the translational information from our new pose ($x_W[0]$ and $x_W[1]$). Therefore, we must apply one more operation before achieving our final pose:

$$x_W = \begin{bmatrix} x_W[0] \\ x_W[1] \\ x_k[2] + \Delta x^k[2] \end{bmatrix} = \begin{bmatrix} x_W[0] \\ x_W[1] \\ \theta + d\theta \end{bmatrix}$$

This process would be applied to all the particles passed in through the use of a simple for loop. Once we have all the particles updated with the car's odometry, we can add noise to our data.

2.1.2 Sample Motion Model Algorithm

Our initial approach to this problem made use of the pseudo-code found in 3 which was pulled from the Probabilistic Robotics textbook [1]. In order to do this, we applied the trigonometric operations defined in the algorithm and then applied noise that was sampled from a Gaussian distribution centered around zero with a variance proportional to the translational/rotational change. However, we found that this implementation caused a lot of problems and the variable noise was giving us unsatisfactory results. We knew that this approach could potentially work, but tuning four different noise parameters would require a large amount of time with very little benefit.

1: **Algorithm sample_motion_model_odometry**(u_t, x_{t-1})**:**

2: $\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
3: $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$
4: $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5: $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$
6: $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3\ \delta_{\text{trans}} + \alpha_4(\delta_{\text{rot1}} + \delta_{\text{rot2}}))$
7: $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

8: $x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$
9: $y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$
10: $\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

11: *return* $x_t = (x', y', \theta')^T$

Figure 1: Sample motion model algorithm from Probabilistic Robotics textbook [1]

Therefore, we took on a simpler approach and applied noise directly to the pose calculated above. Unsurprisingly, we used a Gaussian distribution but with a constant noise variable that we tuned. Our noise injection method can be summarized below with P_1 describing the particles and P_2 describing the noisy particles:

$$P_2 = P_1 + np.random.normal(mean = 0, variance = noise, shape = size(P_1))$$

This process only took place when we set the deterministic flag equal to True. After testing several noise parameters, we found that setting noise equal to .02 resulted in our robot's best performance which we shall describe in our Evaluation section.

2.2 Sensor Model [Habeeb]

The goal of the sensor model is to assign likelihoods to each of the proposed hypothesis particles in the localization algorithm. Particles with a higher probability are more likely to get sampled in the next iteration of the model. The sensor model allows the location hypothesis particles to continually adapt to sensor data in each iteration of Monte Carlo localization.

We implemented the sensor model in two steps. In the first, we precalculated a table of probabilities between sensor measured distances and actual distances. In the second, we used the table to evaluate the probability of each particle given the observed LIDAR data. The following is a breakdown of each of these steps.

2.2.1 Probability Precomputation

A sensor model $P(z_k^i \mid x_k, m)$ is a probability equation that defines how likely a given sensor reading z_k is given a static map m at time k. We modeled this as a combination of four potential cases.

1. p_{hit} models the probability that the LIDAR detects a known obstacle.

We model p_{hit} as a Gaussian distribution ($\sigma = 0.07$) centered around the ground truth distance of the object. The observable sensor reading was upper-bounded by z_{max} – the maximum range of the robot's LIDAR sensor. We broke the continuous gaussian space into discrete range values for our computation. The variable η in the equation below is the normalization factor applied to each discrete probability to ensure the total probability adds to one.

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

2. p_{short} models the probability of a short measurement due to an internal LIDAR reflection.

We model p_{short} as a linearly decreasing function in the distance away from the LIDAR sensor. We assumed a uniform distribution of obstacles. This means

that the LIDAR is more likely to hit objects the closer they are to the sensor.

$$p_{short} = \frac{2}{d} \begin{cases} 1 - \frac{z_k}{d} & 0 \leq z_k \leq z_k \text{ and } d \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

3. p_{max} models the probability of a missed measurement if the LIDAR doesn't bounce back to the sensor.

We model p_{max} as a delta function for sensor measurements of z_{max}. In our discrete ranges, this delta function is over one grid position of the table.

$$p_{max} = 1 \begin{cases} 1 \text{ if } z_k \leq z_{max} \\ 0 \text{ otherwise} \end{cases}$$

4. p_{rand} models the probability of a random unpredictable event. We model p_{rand} as a uniform value over all of the observable ranges.

$$p_{rand} = \begin{cases} \frac{1}{z_{max}} \text{if } 0 \leq z_k \leq z_{max} \\ 0 \text{ otherwise} \end{cases}$$

In each of these four cases, we precomputed a lookup table of probability values indexed by the measured value (z_k) and the actual value (d). The lookup table speeds up the robot's runtime by precomputing the probabilities once instead of several times during operation. We normalized the p_{hit} table independently with the η factor to account for changes in the probability sums when we moved from a continuous Gaussian range to a discrete range. Next, we combined each table and multiplied it by an α value to produce a weighted probability sum. We used the following weighting values: $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, $\alpha_{rand} = 0.12$. We re-normalized the entire table by the columns – d (distances) – to ensure the total probability for each likelihood equation summed to 1.

2.2.2 Particle Evaluation

We used the precomputed probability table to compute the likelihood of each particle given the observed scan. For each particle in the scan data, we use ray tracing to get a simulated distance from the particle to the wall across each angle of the LIDAR scan. These values serve as the actual distance – d –from the map to the hypothesis pose in our sensor model.

We combine this with the observed distance from the LIDAR sensor for each range in the LIDAR scan and look up the probability in our pre-computed table. The likelihood of a hypothesis pose is the product of the probabilities of each of the constituent range scans from the LIDAR.

2.3 Particle Filter [Angelina]

There are four main components to our particle filter. First, we need to initialize the particles. We accomplish this by subscribing to RViz's topic–this allows the user to set the car's pose using RViz's 2D Pose Estimate feature. We use 200 particles in our model; we found this to be the most optimal value that allowed for the benefits of a larger sample size (generalizable samples) but still allowed the model to run in real time. From the ground-truth x, y, θ values, we calculated 200 sample poses. We sampled a normal distribution with a variance of 0.05 meters to obtain the x, y values and a normal distribution with a variance of 0.1 radians to obtain θ. Through trial and error, we found that those variance values maximized accuracy in test cases and best offset human error, as the 2D pose estimate is not necessarily accurate to the robot's actual position and orientation.

Then, we use motion model's *evaluate* function to update particle data upon receiving odometry data from */odom*. Similarly, we obtain particle likelihoods through sensor model's *evaluate* function. We update particle data upon receiving sensor data from */scan* by sampling from the previous particle data, given each pose's likelihood.

After updating the particle data, we average the x, y, θ values. Since θ values are circular, we use *scipy.stats.circmean* for more accurate calculations. Then, we publish the average pose to */pf/pose/odom* and broadcast the corresponding transform with respect to the map (calculated using the $tf2\,TransformStamped$ library. We chose to calculate the average pose because it accounted for all the particle data and included noise from our samples, thus accounting for robot uncertainty.

Through particle filter, we have successfully implemented Monte Carlo Localization and can estimate our robot's position and orientation in the Stata basement.

3 Experimental Evaluation [Izzie]

The first level of testing began in simulation. First, we had to upload our solution to the Gradescope autograder. We found that the autograder returned high scores under all three conditions, as shown in Figure 2. This was determined to be sufficient to move on with our testing.

In simulation, we also ran the wall follower code with localization while showing the robot's estimated pose in RViz. This functioned as a mostly qualitative check. We found that the pose was always where we expected it to be in comparison to the simulated robot. Further, the point cloud of particles was always tightly concentrated near the robot. With this, we were ready to move on to testing on the robot.

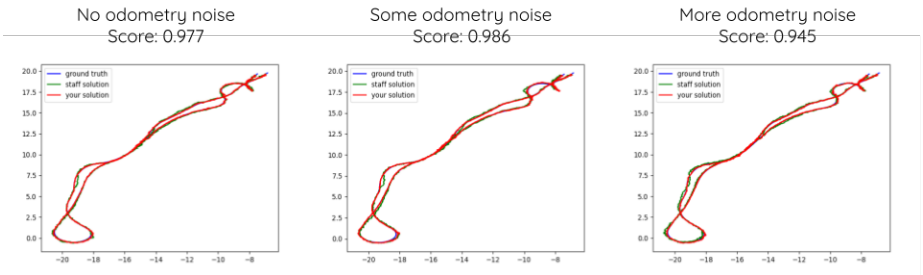

Figure 2: The code performed well according to the Gradescope autograder.

When running code on the robot, we found that we could not set the initial position of the robot in RViz. Thus, we decided to record bag files of the robot's movement, then play these locally, where we could set the initial position.
With this data, we calculated the average distance between each of the particles and the average position over time. This functioned similar to the variance of the particle position. In Figure 3, it is clear that this "variance" remained low throughout runs, proving our particles form a tight cluster as we run localization.

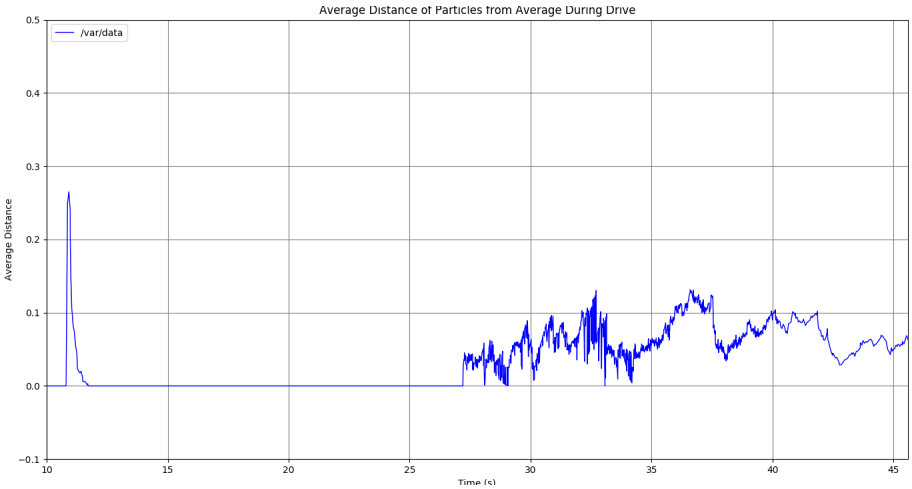

Figure 3: In a graph of the average distance of the particles from the average, the particles on average stayed within .2 m of the average (excluding a spike when the initial pose was set in the simulator).

Unfortunately, in the process of fixing our code to work on the robot, we inadvertently changed something that caused it to stop working as intended. This disfunctionality is reflected in the second plot generated. For the second plot, we

calculated the difference in position and orientation between the ground truth and our estimated pose. As can be seen in Figure 4, there is a large difference even before the robot starts moving, both in direction and distance. The difference in direction grows larger over time due to the difference in orientation (as can be seen in Figure 5, the robot thought it was moving in the opposite direction).

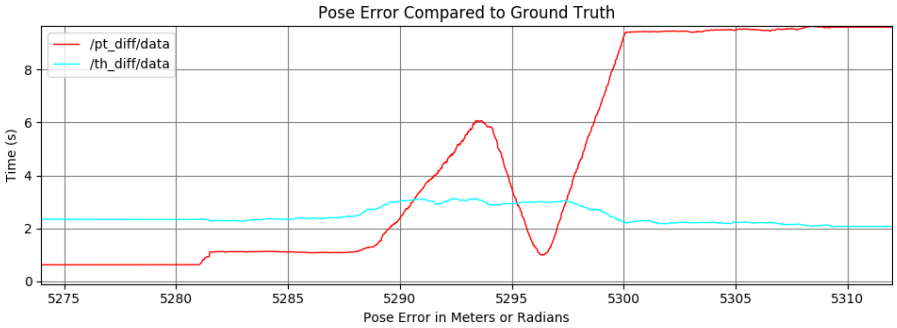

Figure 4: When running on the robot (and outside of simulation), there were clear large differences in position and orientation compared to the ground truth.

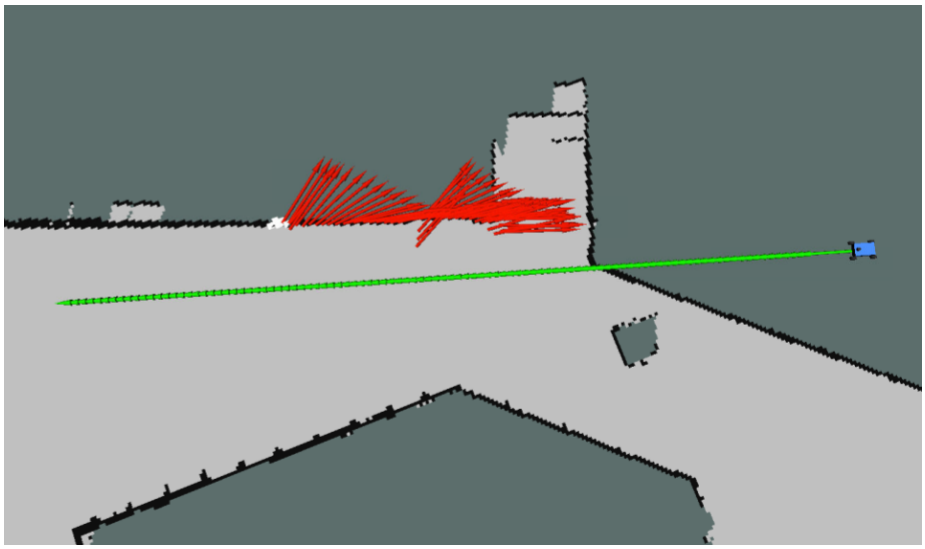

Figure 5: The green arrows represent what is published to the joystick, while the red represent the estimated pose. The white squares represent the particles.

Even if the chart in Figure 4 showed more favorable results, it might not be fully accurate. The "ground truth" was based on flawed joystick data, and suggests that the robot drives backwards through a wall. That said, Figure 5 shows the clear inconsistency in orientation and initial position, so both representations of the robot's movement are flawed.

4 Conclusion [German]

The team found that the optimal noise injection occurred with a variance of .02 and we found that the initial set of probability constants were sufficient. These values were found through the combination of both simulation and real-world testing. For the noise injection, we found our ideal noise parameter by comparing our cross track error from the online test suite. As for probability values, we tested our performance using a rosbag from a real-world run with the car and searched for the lowest overall error.

With regard to the motion model, we found that simpler is better. Ideally we would have liked to use the algorithm found in the probabilistic textbook; however, we realized that the variable deltas would result in too much noise which would ultimately confuse the robot in the long term. Therefore, we maintained a constant noise parameter for both transnational and rotational movements.

With regard to the sensor model, this code was the primary focus of most of our efforts in debugging, despite taking very little time to write initially. As the localization code is still imperfect, this will be the focus of our improvement efforts moving forward.

These two pieces combined in the particle filter worked smoothly together, despite the inaccuracies from the sensor model. That said, debugging the code was initially difficult, and some effort needs to be dedicated to cleaning up the code to make it more readable.

While the initial goals of the lab were met, we believe there is much room for improvement especially with the race competition in the near future. Over the next week, we will like to run our localization module directly on our robot. As previously mentioned, our test methods revolved around the use of rosbags and local testing. For the competition, we need to have this functionality prepared; otherwise, there is no point in creating the module itself. On a similar note, we need to update and tune parameters as soon as we have successfully managed to run the localization module while the robot drives.

In addition, we noticed that our localization module was very sensitive to speed increases. Throughout one of our tests, we used *teleop* to manually drive the car which resulted in quick changes in direction and speed which might have

affected the odometry readings. As a result, our initial pose was far from the initial direction of the car which pushed the car far from the intended path. Once we have managed to implement localization locally on the robot, this is an issue we will need to address due to the high speeds expected throughout the race.

Overall, our current implementation demonstrates a strong conceptual understanding of the task. There is always room for improvement and we understand how to effectively improve the performance of our vehicle. We expect to assign one person to localization improvement as we transition to the motion planning implementation throughout the coming weeks.

5 Lessons Learned

Presents individually authored self-reflections on technical, communication, and collaboration lessons you have learned in the course of this lab.

5.1 Angelina

This lab, I learned the importance of starting early. Since our group started working on the sensor model after spring break, we had very limited time to implement sensor model and particle filter, and also perform tests before the Wednesday briefing. As a result, we focused more on simulation and did not get the chance to tune our parameters for the real world.

5.2 Habeeb

The initial implementation of the sensor model didn't pass the unit test cases that the lab instructors created. I tried adding assert statements and print statements to debug our solution, but the test case output didn't include these debugging statements. I spent a lot of the lab time blindly trying to make tweaks to the code and run the test without having a strong objective function for what was improving the test-case score. This eventually took time away from progressing to the group's integration step. I learned to invest time into developing transparent test cases that work with the team's software development structure. If I had built some test cases that I could debug, our development could have been faster.

I also learned to create a planning buffer of about 1.5 days of work for the remainder of the labs. In this lab and the previous one, I observed that even with an efficient work plan we underestimated how long the integration and testing steps would take for the briefing and report. I'll use this planning buffer in the future to make sure we finish the lab while managing the team's stress levels.

5.3 German

I learned that it is important to communicate goals clearly. We planned to have all separate components for localization prepared by the end of spring break, but this did not happen. As a result, things piled up and we focused primarily on simulation and getting a presentation ready for the briefing. Unfortunately, my academic responsibilities forced me to place all my efforts throughout the first half of the week. Had I clearly communicated my schedule, there may have been an added motivation to have things ready on time. On the other hand, I found that there are several bugs related to the simulator that can be resolved by instructions from prior labs. Overall, there were several technical and communication challenges we faced that shined a light on things we can improve on.

5.4 Izzie

First, I learned that sometimes, if the code is outputting incorrect data, I should just move on to the next step. This is because I spent a lot of time trying to get the precomputed probability table to pass the unit tests, but when I went to office hours the TA said it was probably good enough, so I should just move on. If I had not spent so much time on it, I would have had more time to fix other problems with the robot. I further discovered how difficult it is to debug code I did not write. During the evaluation of the code, I discovered that we had broken the code somehow, and I could not figure out what had went wrong. It took a long time to understand what was happening in the code, let alone start to make guesses about what might be incorrect. In the future, I need to make sure that I understand what other people have written so that I am not caught confused at the last minute.

References

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. 1999.