Lab 5 Report: Applying Monte Carlo Localization

Team 7

Julianne Miana
Binh Pham
Liane Xu (Editor)
Michael Zeng

Robotics: Science and Systems

April 10, 2024

1 Introduction - Liane Xu

The purpose of the lab was to explore the concept of localization in robotics. Localization is the process of finding and tracking a robot's position and orientation within its operating environment. Our team achieved localization in the Stata Basement, both in simulation and in real life, by implementing the Monte Carlo Localization (MCL) algorithm.

Previously, we've used our robot's LIDAR sensor and camera feed to implement algorithms that keep the robot a set distance away from walls and cones. In other words, the robot was reacting to environmental information while it was being run.

This lab introduced the idea of incorporating information that is known ahead of time (the map of the Stata Basement), with information collected at time of running the robot (from its odometry sensors and lidar scans). Additionally, we also dealt with the challenge of being robust to noise when developing models of how our robot sees the world, because no sensors are one hundred percent accurate. The future implication is that, with our new localization module, we will be able to execute planned paths.

2 Technical Approach

2.1 Overview - Liane Xu

Monte Carlo localization (to be described in more depth in a later section) is a way of estimating our car's pose. It starts with considering a large number of potential poses (known as "particles") scattered around the map, then continuously iterating its estimate around the poses that are the most likely based on odometry and lidar information as the car moves around its environment.

We approached localization in three steps:

1. Building a **motion model** that transforms the poses of the particles based on car's wheel odometry information.
2. Building a **sensor model** estimates how probable each particle's scan is, based on the ground truth scan.
3. Combining motion model and sensor model in a **particle filter** that continuously gives its best estimate of the car's current pose.

2.2 Motion Model – Liane Xu

The motion model uses our robot's wheel odometry data to estimate the pose of our robot. Motor and steering commands give us the real-time linear and angular velocities of the car. Integrating these quantities over time, also known as dead-reckoning, gives us odometry information. In other words, we are able to obtain an estimate of the distance and direction our robot has traveled over that period of time $\Delta X = [dx, dy, d\theta]^T$.

However, odometry can be noisy for several reasons, including the wheel slipping or someone accidentally bumping the car. Therefore, in simulation we have to add noise in our motion model to reflect how the car's sensors might behave in real life. Every time we update the pose of a particle, we add some Gaussian noise in the x, y, and rotational components of the odometry information. On the real robot, we did not need to add this artificial odometry noise because it has real odometry noise.

Finally we composed the position of the particles at time *t-1* (with some noise added) with the noisy odometry information to get an estimate of the position of the particles at time t:

$$X^W_{particles,t} = X^W_{particles,t-1}\Delta X$$

2.3 Sensor Model - Julianne Miana

The sensor model computes the likelihood of each particle's pose. Laser scan range measurements are projected from each particle, and the probability of the

scan is calculated by comparing it to the ground truth laser scan as measured by the car.
To compute the probability of each range measurement, the model considers 4 cases:

1. p_{hit}, the probability of detecting a known obstacle on the map
2. p_{short}, the probability of a short measurement
3. p_{max}, the probability of a missed (extremely large) measurement
4. p_{rand}, the probability of a completely random measurement

Each case has a contribution scaled by a weight α towards the probability of a range measurement. All four α values sum to 1 to produce a probability distribution for the likelihood of each range measurement:

$$p_{particle} = \alpha_{hit} * p_{hit} + \alpha_{short} * p_{short} + \alpha_{max} * p_{max} + \alpha_{rand} * p_{rand}$$

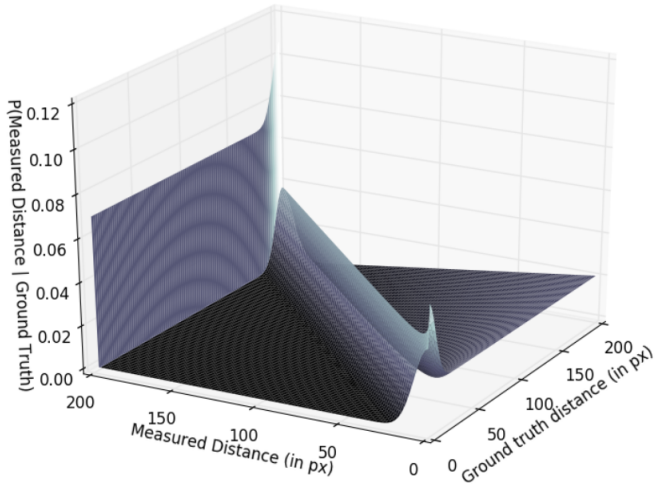

Figure 1: A plot of our precomputed sensor model.

In implementing the sensor model, we created a helper SensorModel class with two methods, `precompute_sensor_model` and `evaluate`, to calculate the likelihood of each particle. The `precompute_sensor_model` method precomputes a table of probabilities for each pair of discrete d (actual distance) and z (measured distance) values (over the range [0, 200]), which optimizes the runtime of our algorithm because we can use these hashed values repeatedly without needing to recalculate them every time we get a laser scan. Figure 1 shows a plot of this precomputed sensor model. The `evaluate` method then looks up range measurement probabilities and calculates their product using d and z values derived from particle and observation data.

2.4 Monte Carlo Localization - Binh Pham

Monte Carlo Localization integrates the Motion Model and Sensor Model to estimate the car's pose on the map. The algorithm is a particle filter. At a high level, we start with a lot of particles that could potentially be the car's pose and filter out particles that are least likely to be the car's pose. The filtered-out particles are replaced with newly generated particles that are more likely to be the car's actual position, and the process iterates.

We implemented Monte Carlo Localization as a ROS node called Particle Filter. To get the initial particles, we generate particles (200 in simulation and 100 on the car) normally distributed around an estimated position. The car's estimated pose is the average pose of these particles. The average pose is calculated by taking the arithmetic means of the particles' x and y positions and the circular mean of their angles. This estimated pose is published as both an odometry to /pf/pose/odom and a transform from /map to /base_link.

When the node receives odometry data (published to /odom in simulation and /vesc/odom on the car), it calls the Motion Model to update the poses of all the particles. However, the odometry data contains velocities, so the node needs to process the data before sending it to the Motion Model. To get the odometry that works with the Motion Model, we need to simply multiply the odometry values by the time in seconds since the last time we received odometry data (using `time.perf_counter()`). This gets us the raw change in x, y, and angle values since the last time the particle positions were updated. Finally, the particles' average pose is updated.

When the Particle Filter receives the LIDAR scan data, it uses the Sensor Model to evaluate the likelihood of each particle being the car's actual position. We normalized the likelihood of each particle (by dividing each particles' likelihood by the sum of all the particles' likelihoods) so we could use them as probabilities for random sampling. After sampling the particles with replacement, we added Gaussian noise to the x ($\sigma_x = 0.05$(car's speed)), y ($\sigma_y = 0.05$(car's speed)), and rotational components ($\sigma_\theta = \pi/30$). We tuned the noise standard deviations to be robust to "robot kidnapping", where the particles all converge to some wrong pose. Finally, the particles' average pose is updated.

Over time, the average pose of the particles converges to the car's true pose.

2.5 Performance Optimizations - Michael Zeng

The accuracy of our implementation is highly dependent on implementation details. Firstly, we found, based on qualitative observation, that the rate at which the algorithm can react to the odometry noise by filtering particles affected the impact the noisy would have on our position estimates. Therefore, we used NumPy to vectorize all array and matrix operations and down-sampled our LI-

DAR scans to 100 beams to reduce computational load without significantly compromising the usefulness of the LIDAR scans. Additionally, we found that the method of sampling had a strong impact on both the effectiveness of the sensor in converging to the correct position, and the rate at which the algorithm would converge. What we found worked best was sampling the particles with replacement without any nonlinear scaling of the particle probabilities; both of these methods serve to promote the chances of sampling only the highest probability particles. The byproduct of these two methods is that we would often sample very few distinct particles; so, to combat this, we also added independent Gaussian noise to each particle.

2.6 Localization On the Real Car - Binh Pham

Before our Monte Carlo Localization could work on the real car, we had to account for the odometry axes being flipped on the car. This was simply corrected by negating all the odometry data.

We also had hardware issues that prevented our Monte Carlo Localization from smoothly transferring to our physical car:

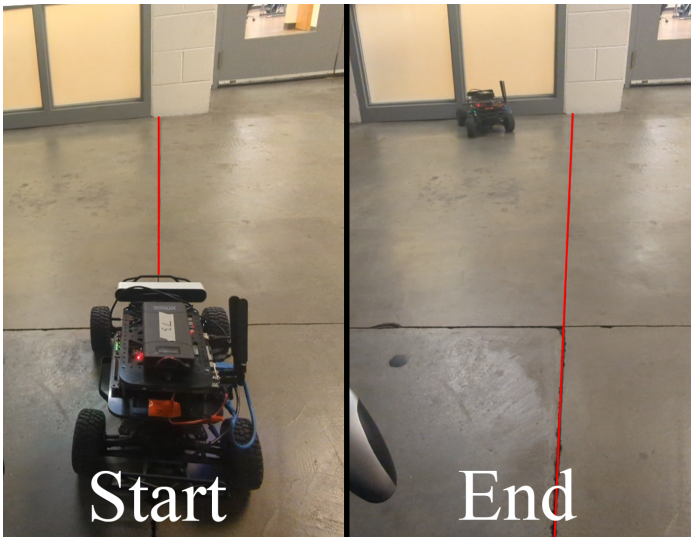

Figure 2: The starting and ending location of the car as it drives forward. The line marked in red is the path the car would have driven along if it drove straight.

First, as seen in Figure 2, our car curves left when driving forward (no steering angle command). To account for the car curving as it's supposed to drive straight, we added an anglular velocity of 0.08 radians/second to the angular velocity from the odometry, which we found experimentally.

After fixing the angle error, we noticed that the odometry was still not correct. It was consistently short in the x-direction on the real car. For example, when the car moved forward 1 meter, odometry would say that the car only moved about half of that distance. To account for this systematic error, we centered the distribution for x around $\mu_x = 0.05$ instead of 0. We may have to make this mean proportional to the car's speed for better localization in the future, but it worked well enough for the testing we did in this lab.

Finally, we slowed down our sensor model so that it only runs every 10 scans only while the car is moving. The goal of this change was to slow the rate of convergence. However, these changes were an attempt to solve an early issue where particles were converging very quickly to a wrong location, and they did not completely solve those issues. We may be able to have the sensor model run more frequently in the future for more accurate localization.

We also used mutex locks in the odometry and laser callbacks to ensure all writes to the particle filter array are thread-safe.

3 Experimental Evaluation

3.1 Simulation localization performance - Julianne Miana, Michael Zeng

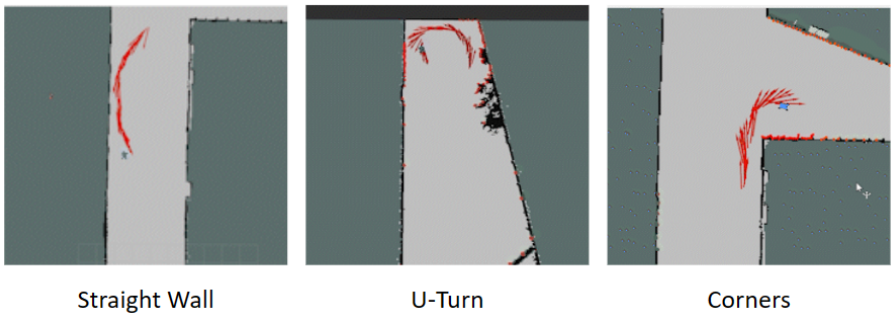

Figure 3: The general paths of the three tested scenarios in simulation.

To assess the performance of our localization program, we first conducted tests in simulation, with three test scenarios and three standard deviations of simulated odometry noise. The three scenarios (as seen in Figure 3) are:

1. driving along a **straight wall**,
2. making **U-turns**,

3. and rounding **corners**.

The three standard deviations tested are **2, 4, and 8 cm**, chosen to represent a realistic range of car performance on the real car.

For each scenario and standard deviation, we conducted 3 trials, totaling 27 trials overall.

3.1.1 Localization Accuracy

We first assessed our program's accuracy using a binary metric: did our localization converge (find the robot's ground truth position) or diverge (stop tracking the car's position altogether)? Following our 27 test trials, we found that our program converged in all cases, and therefore should be robust to odometry inaccuracies on the real car.

Additionally, we measured the average error of the pose $[x, y, \theta]$ computed by our program over each trial run to assess how closely our localization program could track our car. To measure the average error, we measured the accumulated difference between ground truth odometry values produced by the simulation and the average pose values from our particle filter, and divided by the number of times an average pose was produced throughout a testing scenario. We repeated this measurement across all test trials, and averaged the results for each scenario and standard deviation of simulated odometry noise:

Test Scenario	X Error (m)	Y Error (m)	θ Error (rad)
Straight Wall	2.33	0.36	0.07
U Turn	0.67	0.24	1.38
Corner	0.52	0.40	0.14

Table 1: Error values for 2 cm STD odometry noise.

Test Scenario	X Error (m)	Y Error (m)	θ Error (rad)
Straight Wall	2.15	0.24	0.10
U Turn	0.67	0.26	1.56
Corner	0.89	0.39	0.90

Table 2: Error values for 4 cm STD odometry noise.

Test Scenario	X Error (m)	Y Error (m)	θ Error (rad)
Straight Wall	1.31	0.17	0.08
U Turn	0.69	0.38	1.28
Corner	0.48	0.28	0.81

Table 3: Error values for 8 cm STD odometry noise.

Figure 4: The straight wall case lacks distinguishable features that would help the car localize itself more accurately.

We found that most errors were low across the three scenarios and the three standard deviations with two exceptions to these low average errors: the average X-error in our straight wall case and average angle error in our U-turn case. We believe that the first exception is due to the lack of distinguishable features along the first section of the straight wall, as seen in Figure 4, and from the subsequent periodic wall "spikes" that make it difficult for the sensor model to determine the car's x-position. The second exception is caused by a normalization bug in our angle error computations that resulted in frequent error spikes when the car would drive upward on the map. This is a bug in our testing code and is not representative of the performance of our algorithm, although a closer analysis of the test data does show that the angle error was higher during the U-turn tests even with the normalization bug removed; this is likely due to the frequent swerves in the U-turn trial, which can cause dramatic shifts in the sensor model's estimated probabilities.

3.1.2 Noise Resistance

Based on the testing data, we found the localization to be quite resistant to simulated odometry noise, with a weak trend of lower error with more noise. This shows that our sensor model is performing very well and able to remove particles that deviate from the ground truth. The increase in performance with higher odometry noise might also result from the increased spread of the par-

ticles, and that sometimes these spreads are able to steer the position estimate back toward the ground truth when it deviates.

3.1.3 Convergence Rate

Lastly, we measured the convergence rate of our localization to assess how quickly our program can converge towards our car's ground truth pose. To do this, we ran one trial with each of the simulated odometry noise values where we initialized the robot and particles (in a normal distribution around the robot) in the center of an enclosed area of the map, and computed the variance in x, y, and θ of the particles and plotted these values in real time. We measured the time it took for these variance values to decrease from the initial normal distribution to their stable values, which was approximately 0.20 seconds, or 10 loop cycles, regardless of the level of odometry noise. This high rate of convergence could minimize lost time for the robot before it an begin completing meaningful tasks like navigation in future labs.

3.2 Real car localization performance - Liane Xu

To test the performance of our localization algorithm on the real car, we visualized the particles and their average pose on Rviz while a teammate teleoperated the car.

3.2.1 Qualitative measurements

We tested the car's ability to localize under different starting conditions, including facing a column, parallel to a wall in a long corridor, and perpendicular to a wall. We found that the the robot could localize itself well when facing the column because it is feature-rich. In contrast, when starting parallel to the wall in a long corridor, there was more error in the x direction because there were no features along the wall that made any particular particle facing in the same or opposite direction of the car better than the other.

We also tried a variety of driving "styles", including zigzagging and following a straight line. We found the motion model to be responsive to the the odd shapes that were traced out by the car, though when we got close to walls, the particles would drift off of the map over time, which is the robot kidnapping problem.

Lastly, we tried a variety of loops, including the large orange circle outside the RSS classroom, and a loop from the orange circle to the door at the entrance of the Stata basement nearest to the classroom and back to the orange circle.

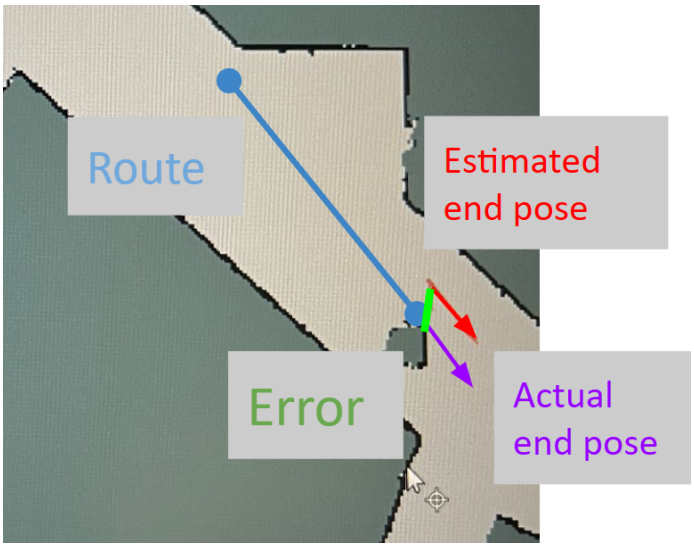

Figure 5: The experimental setup for quantitative evaluation of our localization on the real car. The blue line is the route of the car, starting from the top left and moving towards the bottom right. The red arrow is the estimated final pose, the purple arrow is the actual final pose of the car, and the green line is the error/distance between the estimated and actual final pose.

3.2.2 Quantitative measurements

To measure how our estimated pose drifts over time, we selected a 5.08 meter (200 inch) straight route in the Stata basement that started at a corner between walls and ended at the column near the RSS classroom. We used the *Measure* tool in Rviz to estimate the distance error between the car's inferred position and the car's actual position, using the a corner and the column as landmark features. Figure 5 shows this experimental setup.

The average error across 4 trials was 0.6m (see Table 4 below). Qualitatively, there was more error in the x-direction than in the y-direction. The inferred x-position of the car always falls short of the actual x-position of the car, which indicates that we should further tune the noise distribution of the x-position of particles.

Trial number	Error (m)
1	0.71
2	0.49
3	0.45
4	0.73
Average	0.6

Table 4: Real car localization test results.

4 Conclusion - Julianne Miana

In this lab, our team was able to build and implement a robust localization program using the Monte Carlo localization method. This method consists of two components: a motion model to update current pose particles and a sensor model to compute the likelihoods of potential pose particles of the car at each time step. Together, these models filter for and update the most likely pose of our car.

To optimize the performance of our localizer, our team built a motion model that was resistant to odometric noise and a computationally efficient sensor model that helped track particles with the highest probabilities. These optimized methods resulted in low average error between average pose and ground truth, quick responses to noise, quick convergence to the car's ground truth pose, and consistent and smooth tracking. The transition from simulation to the physical car required tuning; however, after these minor modifications, we continued to see consistent and accurate tracking, as demonstrated by low average errors in real-world tests.

Although we are satisfied with our car's localization performance in simulation and on the physical car, we aim to improve it by filtering out-of-bounds particles and preventing out-of-bounds initialization when starting close to a wall. Furthermore, we aim to combine our localization program with mapping and path-planning techniques to progress towards a fully autonomous car.

5 Lessons Learned

5.1 Julianne Miana

This was my first time learning about Monte Carlo Localization and I thought that implementing it was a challenging but informative process. I was already familiar with most of the math involved; however, building a working localization program that implemented this math was a steep learning curve. Regardless, I enjoyed learning how to perform the theoretical work and then applying that knowledge to a physical system. I also learned more about ROS and how to manage nodes and packages to run simulations. On the communication and

collaboration side, an important lesson for us was documenting what parts of our program we were unsure of to more quickly debug errors/issues in our implementation.

5.2 Binh Pham

This lab was particularly difficult. At the beginning, I did not have a grasp on how Monte Carlo Localization worked. I particularly did not understand the resampling step of MCL. As a result, the way I implemented that step was very questionable. It worked fine in simulation, but when transferring to the real car it did not work at all. Thankfully, Liane came up with a simpler and better resampling step that transferred well to the real car after some tuning. Once again, the power of collaboration prevails, and my weaknesses and oversights are covered.

5.3 Liane Xu

I enjoyed learning about localization in the context of robotics. Previously, I knew about the Monte Carlo algorithm through Computational Cognitive Science so making the connection between two seemingly different fields was cool – how are robots like humans and how are humans like robots? This lab also taught me a lot about working with ROS, both in managing nodes and publishing and subscribing to topics. Finally, I found this lab to be the toughest out of all the labs. It seemed like the README told us step-by-step what to do so there would be no creative aspect, but in reality, there was much debugging and figuring out to do.

5.4 Michael Zeng

Monte Carlo Localization was definitely a harder subject compared to the topics of previous labs, so I learned that it's incredibly important to ensure a firm grasp on the algorithm before attempting to get too far into the code. Additionally, we found that documenting code well, particularly parts that we are unsure about at time of writing, can save a lot of time later when it comes time to debug the code; in this lab, there were a lot of uncertain parts that ended up causing very difficult bugs later in the lab. Finally, I enjoyed seeing an application of probability in robotics, as this is the first time for me applying may of the principles from probability class in an actual robot.