Lab 5 Report: Localization

Team 8

Xavier Bell
Nicolaniello Buono
Mary Foxen
Elise Harvey
Eli Scharf

6.4200

April 11, 2024

1 Introduction

(Eli)
Robot localization is a fundamental but difficult task in autonomous robotics. It requires a confident understanding of your surroundings with many unknown factors. However, if done successfully, understanding your robot's location offers a great deal of freedom in autonomizing your robot. Localization enables robots to orient themselves within their environments and plan actions to reach a target. In this lab, we are building on previous work to add a Monte Carlo localization algorithm to our robot. This will allow the robot to know its relative location on a given map.

In previous labs, our team worked to autonomize the robot using cameras and LiDAR data. This lab requires more advanced techniques. Given a starting location and map, this lab focuses on orienting our robot on the map. By understanding the robot's location on the map, we can optimize the decisions the robot makes to reach its desired state.

The primary goal of this lab is to implement a Monte Carlo localization algorithm. The algorithm breaks down into two parts: the motion model and the sensor model. These parts are combined and together they make the Monte Carlo algorithm. Using these methods, the robot can independently determine its position and optimal path.

The technical challenge is to accurately estimate the robot's location. Uncertainty in scan data, the environment, and odometry will pose a large challenge. Monte Carlo localization, a probabilistic approach, handles uncertainty well and is robust to noise.

Monte Carlo localization leverages the use of multiple hypotheses about the robot's position based on the LiDAR data. These hypotheses are generated using the motion model and odometry data from the robot's current movements. The sensor model assigns respective probabilities to each hypothesis based on the LiDAR data. The hypotheses are sampled based on their probabilities and one hypothesis is chosen as the robot's estimated location. This approach allowed us to maintain a distribution of likely poses. It provides a flexible and robust solution to the localization problem. Throughout this lab, our team takes steps toward developing a fully autonomous robot.

2 Technical Approach

Localizing a robot is a surprisingly difficult task in the field of robotics. There is a great deal of uncertainty in the robot's perception and motion control. Modeling the environmental variability, dynamic obstacles, and noise is also a difficult challenge.

Monte Carlo localization is a robust algorithm that takes the robot's LiDAR scan data, odometry data, and a map of the environment. It uses this information to predict the robot's location and pose within the map. To track our accuracy, we visualized the error between our pose estimate and ground truth in simulation. By iterating over the non-deterministic parameters, we were able to improve the accuracy of our model. Monte Carlo localization requires two main components: the motion and sensor models. The motion and sensor components are then combined in the filter model to complete the algorithm. Through this approach, our team implemented a successful Monte Carlo localization algorithm.

2.1 Motion Model

(Nico)
The motion model uses both odometry data and a collection of potential poses, represented as particles, as its inputs. Wheel encoders capture the robot's movement by tracking the motion of the wheels and directional shifts (left/right and forward/reverse motions). The odometry data are the main source of information for predicting the robot's future pose. The particle data are an Nx3 array. There are N particles, all of which have an x, y, and θ component. These can be thought of as the pose guesses for the robot.

The particles are all in the global map reference frame and the odometry data are in the robot's reference frame. To use the odometry data and particle data,

it must first be converted to a single reference frame. Without one common reference frame, our computation would lead to inconsistencies and divergences in the localization process. Our team used a rotation matrix (Figure 1) to convert the odometry data to the reference frame.

cos(Θ)	-sin(Θ)	0
sin(Θ)	cos(Θ)	0
0	0	1

Figure 1: Rotation Matrix: Convert from Robot's frame to map frame

(Xavier)
In a common reference frame, the odometry data are scaled by the difference in time between now and the last measurement. We are only interested in the most recent time step of data to see how the robot has moved. We apply this change in wheel speed/angle data to update each particle in the particle matrix. Because our odometry is imperfect (attritbuted to encoder noise, wheel slippage, or sensor calibration), normal Gaussian noise is also added to each particle. We changed the standard deviation to optimize our accuracy. The addition of noise allows the model to account for uncertainties and imperfections in the robot's motion and measurements. This reflects the fact that the accuracy of dead reckoning localization (using just odometry) decays over time. Randomized noise simulates the potential errors in the many factors that affect the robot's motion. This noise helps the algorithm account for errors by creating different potential outcomes for several different errors (in different directions). Without noise, the algorithm would be deterministic and assume its readings were perfect. This is an unrealistic assumption.

The updated particles are roughly localized around ground truth, but have also spread out. These particles become the new particle matrix. The sensor model takes these particles and the LiDAR scan as inputs to determine the likelihood that the robot is at each particle.

2.2 Sensor Model

(Xavier)
The sensor model takes the Nx3 particle matrix (estimated poses) and the robot's observed LiDAR scan data as inputs. The model uses the LiDAR data to assign probabilities to each of the particles in the matrix. It does this by modeling the chances of a correct, missed, short, and random measurement based on measured distance and true distance. These probabilities reflect the alignment of the LiDAR scan with the map at each pose. To assist with computation, a precomputed probability lookup table was used and the LiDAR data

was downselected.

First, the robot performs ray-casting to generate a hypothetical LiDAR scan for each particle. This unique LiDAR scan is what the actual LiDAR scan would return if the robot were at that specific pose. A LiDAR scan from the robot usually consists of over 1,000 unique readings and the particle matrix has N possible poses.

Given an expected scan, the likelihood scan evaluates the similarity between the expected scan and the real sensor data. Greater similarity between the two scans causes the Sensor model to assign a greater probability to the given particle. Large discrepancies cause the algorithm to assign the particle with a low probability.

Determining the probability for 1000 beams for N particles can be computationally intensive. To combat this, we create the ray-casted LiDAR scans with only 100 readings. The robot's ground truth LiDAR scan is downselected to 100 beams. A smaller scan meant our algorithm needed to compare fewer points for each iteration. To alleviate the computational burden, our group created a lookup table of precomputed values. The table included a discrete range of possible LiDAR values with particle beam distance and ground truth distance to obstacle as inputs. The following sections will detail how these probabilities are computed. Finally, our group used the Numpy library in Python which has been optimized to perform data-related computations.

(Eli) The likelihood scan function is computed as the product of the likelihoods of each of n range measurements in the scan, where $p(z_k^{(i)}|x_k, m)$ is the likelihood of range measurement i. This is represented mathematically:

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

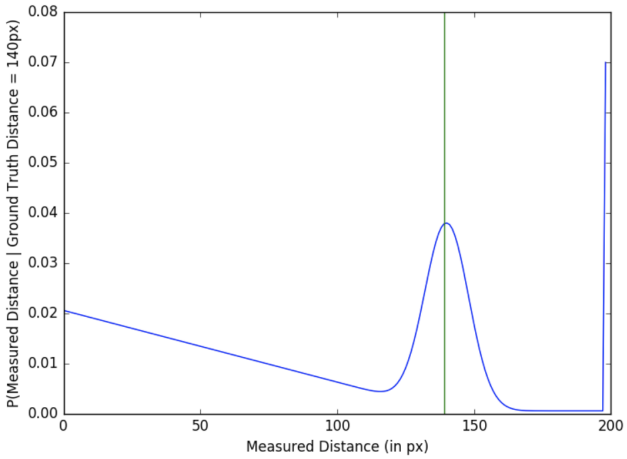

Figure 2: Likelihood Function in 2D. The function plots the probability of a LiDAR-measured distance given ground truth. Source: https://github.com/mit-rss/localization

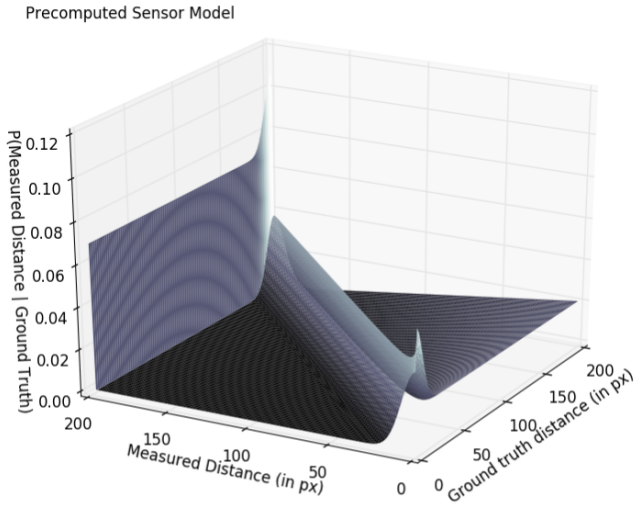

Figure 3: Likelihood function in 3D. Shows measured distance, ground truth distance, and likelihood values. Source: https://github.com/mit-rss/localization

For each measurement, several components are combined to find $p(z_k^{(i)}|x_k, m)$. The first component is the probability of detecting a known obstacle on the map. The probability of detecting the known obstacle is represented as a Gaussian centered around the ground truth. In Figure 2 this is represented as Gaussian

centered around the green line. If the measured distance exactly equals the ground truth the probability will be greatest. The first component can be represented mathematically as shown below. η is a normalization constant such that the Gaussian integrates to 1 on the interval $[0, z_{max}]$.

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

The second component represents the LiDAR hitting an unknown obstacle, such as a person or another robot. This is modeled as a downward-sloping line. It is more likely for the LiDAR beam to hit an obstacle if it is closer to the robot. The constant d is predetermined.

$$p_{short}(z_k^{(i)}|x_k, m) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} & \text{if} \quad 0 \leq z_k^{(i)} \leq d \text{ and } d \neq 0 \\ \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The third scenario is a missed measurement where the LiDAR beam never returns to the sensor. This happens if a surface has odd reflective values. in Figure 2, this is the large spike in probability at high measured distances. The third scenario is calculated as follows:

$$\max p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{\epsilon} & \text{if} \quad z_{max} - \epsilon \leq z_k^{(i)} \leq z_{max} \\ \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The final scenario takes into account a random LiDAR scan being returned. For this case, we use a small uniform value.

$$p_{rand}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_k^{(i)} \leq z_{max} \\ \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

To calculate the full likelihood function we take the weighted sum of all four scenarios.

$$\begin{aligned} p(z_k^{(i)}|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m) \\ + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m) \\ + \alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m) \\ + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m) \end{aligned} \tag{6}$$

(Xavier) We must ensure that summing all the alphas together equals one. With this statement true we have a probability distribution as seen in Figure 2. We

use this final equation to populate our lookup table making sure to normalize correctly along each column (ground truth distance).
With a complete likelihood function, we can use it and estimated poses (particles) from the motion model to compare each ray-cast estimated pose scan to the actual robot LiDAR scan. We use the similarity between the two scans to assign a probability to each particle. These probabilities are then used in the particle filter to finally determine the robot's position.

2.3 Particle Filter

(Mary)
After finalizing the motion and sensor models, we tied our Monte Carlo Localization algorithm together with a particle filter. At a high level, the particle filter works by using the motion model to update particle positions when odometry data are received and by using the sensor model to compute the particle probabilities when sensor data are received. In addition, we calculate the average of the computed particles and publish the resulting transform each time.

First, we initialized N data points at the robot's initial location in the simulation. At each time step after this, the particles were updated with the current odometry data and noise. In this way, each update to the robot's position results in a field of particles spanning outwards from the robot.

When the motion model is called, the average x, y, and theta values of the particles are computed and published to the odometry topic. The motion model is called each time the odometry topic sends new data. For theta values specifically, we used the circular mean. We used this approach because of the geometric nature of the particles. To calculate the circular mean we take the cosine of each angle and the sine of each angle and sum each array created. We then calculate the arctan of the cosine and sine angle sums. In other words, the function treats angles as vectors in a circle, computes their x and y values and finds the angle of the resultant vector. This yields the mean direction of the angle. The mean particle is then converted into a transform and published as a TransformStamped message.

When the sensor model function is called, it returns a list of probabilities. The sensor model is called each time the LiDAR scan sends a new message. The particle filter takes the list of probabilities and their respective particles. It samples the particles based on their probabilities and returns a new set of N particles. This new set is sampled based on the probabilities from sensor model with replacement. Then, we sample one more particle. This particle is the pose estimate, and this pose is then published to the odometry topic. The predicted particle is then converted into a TransformStamped message and published. This way, we update our particles each time the sensor model reassigns probabilities to the previous set of particles.

2.4 ROS Implementation

(Mary)
In this lab, the only node was Particle Filter. Particle filter subscribed to these topics: /map, /scan, /odom, /initialpose. The topic /map sent the initial map data for the Stata's basement. The topic /scan contained the LiDAR data as a scan message. The robot sent its odometry data over the /odom topic and /initialpose was the initial pose of the robot. The node published to /base link pf, /base link, and /pf/pose/odom, /particles, and a topic we created to show the path particle filter predicted. In simulation, we used /base link pf, but on the real robot we used /base link, but both took the same data. For those two publishers, we sent a TransformStamped message. For /pf/pose/odom, we sent the current estimate for the robot's odometry as an odometry message. When moving from the simulation to the robot we changed the orientation. The robot has positive x pointing behind the car and positive y pointing to the right. This differs from simulation where it assumes positive x is forward and positive y is left (relative to the car). On the robot, we updated the code by applying a coefficient of -1 whenever we received odometry data.

3 Evaluation

(Elise)
To evaluate our algorithm, we tested it in simulation and on the real robot. In simulation, we compared our estimate pose with the ground truth, which we calculated. We did this by tracking our starting position's x, y, and theta values. We then updated these values in the same way we did with the motion model but without the added noise (deterministically). We updated this ground truth value whenever we received odometry data to track the location of the car.

3.1 Using Localization in Simulation

In simulation, we changed noise parameters by publishing a visual for all of the particles. We updated the standard deviations for x, y, and theta values by selectively changing each and seeing how well the path was plotted. We then published error messages between the ground truth x and y and the values that our particle filter mapped. This difference is shown below in Figure 4. Combined with a visual of our path being successfully updated, we considered this a solid working product.

Due to the noise, we will never be able to achieve an error of zero. Additionally, since the particles' locations and assigned probabilities update so quickly, the error jumps around multiple times per second. Lastly, considering our path plotted in Figure 5 follows the actual path of the car well, we accepted the average error of approximately 0.3 meters.

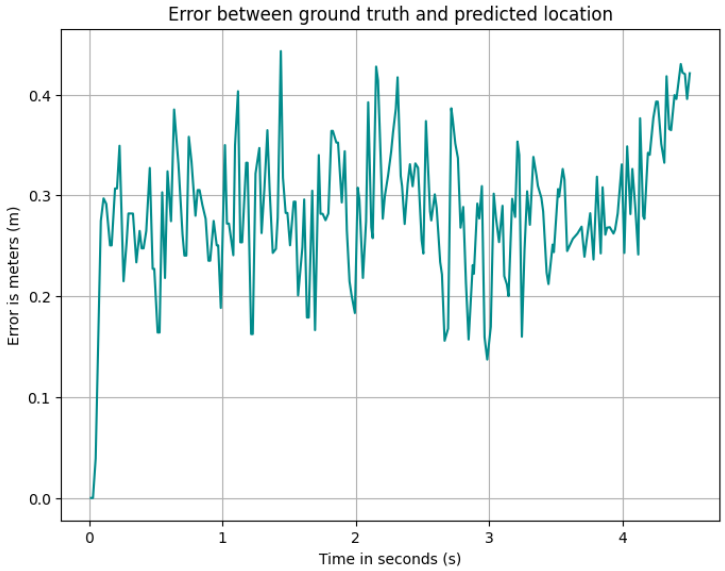

Figure 4: Error over time for simulation run of particle filter locally. Error calculated compares ground truth to estimated location. Fluctuation in error showcases the randomness in the model

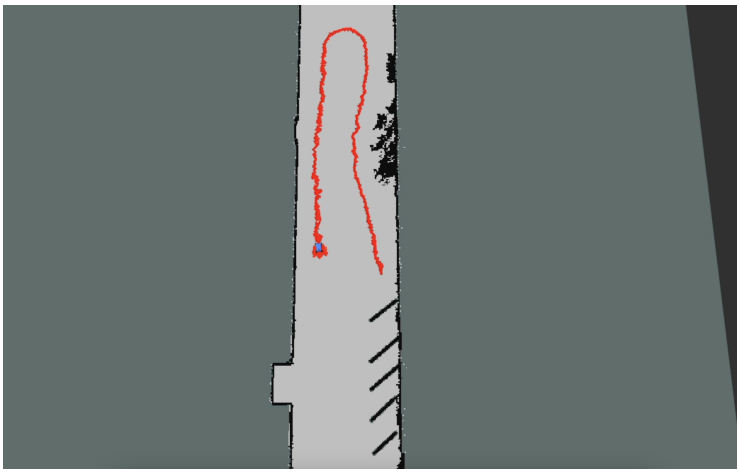

Figure 5: Plotted path by particle filter when running our wall follower in simulation. The red bubble around the car is a visualization of the randomized particles.

3.2 Using Localization in Robot

To test our implementation on the robot, we had to change a few parameters in our particle filter and update how to odometry data was read. The simulation assumes positive x is forward and positive y is left (relative to the car). When using the odometry data learned from the car, it has positive x pointing behind the car and positive y pointing to the right. Thus, we were able to handle this by applying a coefficient of -1 whenever we received odometry data from the car.

Then, we were able to run particle filter by physically placing the car on the ground in the Stata basement, estimating the 2D Pose estimate in our simulated map, and driving randomly with the car. We included a side-by-side of the real video of the car driving in the basement with what particle filter mapped in our briefing slides. We found our mapping to be pretty accurate, with some jumps due to noise and predicted locations. Additionally, this video was recorded at a fast speed with tight turns, which suggests that our particle filter stands up to extreme cases in driving scenarios.

We did not numerically calculate the error on the real-world run because the odometry data on the car is prone to error as compared to when we were just running in simulation. This made tracking a ground truth location infeasible, which means our error calculation would have been incorrect. Thus, as mentioned, we qualitatively evaluated the performance of the particle filter on the robot.

4 Conclusion

(Nico)
We were able to successfully implement Monte Carlo localization on our robot. Our robot can localize itself given map data, a LiDAR scan, and odometry data. Our team divided the work for the Monte Carlo localization into subsections (motion model, sensor model and particle filter) and approached each component in parallel. For this lab, we gave the robot its starting position. In the future, we would like to implement SLAM, simultaneous localization and mapping. Using a SLAM algorithm we could start the robot from an unknown location and the robot would still be able to localize itself. Additionally, we are interested in further testing our robot at different speeds. We are interested in exploring if there is a correlation between speed and noise generation in the motion model. We hypothesize that as speed increases the standard deviation for generated noise will also increase. However, we are unsure if this relationship is linear or exponential.

5 Lessons Learned

(Eli)
These comments have been aggregated from our reflections. In a group, we all shared our comments and wrote this reflection together. Our group found this lab to be the most challenging. Our group started off well by dividing work between all members and meeting all deadlines. However, towards the completion date, we stopped meeting deadlines. This caused us to spend more time in the lab in the last few days. For future labs, we will divide the work early and ensure deadlines are kept.

Since this was our first written assignment, our team divided the written lab report into five sections. We used a similar process as we do for dividing code. This worked well for the team. Each of us worked on a specific section. We then cross-checked one another to ensure a constant style and to catch errors.

For the briefing, each member built the slides for the code they wrote. We learned from previous lab briefings that this works best for our team.