Monte Carlo Localization on Racecar

Andy Li
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
andyli27@mit.edu

Antonio Avila
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
antonio3@mit.edu

Harmanpreet Kaur
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
hkaur@mit.edu

Mia Tian
MIT Department of Aeronautics and Astronautics
Cambridge, USA
miatian@mit.edu

Natalie Huang
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
huangn@mit.edu

1 Introduction

Natalie Huang

The purpose of this lab is to use Monte Carlo Localization to determine the position and orientation of our racecar in a known environment– the Stata basement. This functionality will be important in the final challenge for the racecar to navigate around a track in the basement, because correct autonomous navigation requires us to not only be able to detect and follow walls and lines as we have implemented previously, but additionally recognize where the racecar is along the track. In this lab, the specific deliverables we met were creating a working version of the localization algorithm in simulation, and testing this in real life in the Stata basement.

Our team implemented the Monte Carlo Localization algorithm, which consists of three main modules. The algorithm generates a large number of particles, which represent possible poses and orientations. At the beginning, the particles randomly surround the initial estimate. The motion model applies odometry data to these particles to update their positions according to the racecar's movement. The sensor model uses real-life LiDAR data to calculate the probability of each particle being the true position of the racecar, and narrows down the possibilities accordingly. Finally, the particle filter repeatedly runs both models using updated odometry and LiDAR and publishes the average pose of the particles to provide a real-time estimate of car's pose.

2 Technical Approach

This section will cover the design and implementation of the motion model, sensor model, and particle filter.

2.1 Motion Model

The motion model uses the car's odometry to update the positions of the particles. Updating the positions is done particle-wise by calculating the odometry transformation matrix and applying it to the last position of the particle. This matrix product gives the estimated position of the particle on the next time step. Suppose the last time step is $k-1$ and the next is k. Given a particle x_{k-1}, y_{k-1}, θ_{k-1}, and odometry data Δx, Δy, $\Delta\theta$.

$$T_{k-1}T_{\Delta} = T_k \tag{1}$$

The T matrices are the corresponding two dimensional transform matrices of the form:

$$\begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

By this equation we are able to update the positions of the particles. However, noise must be added as well to account for small uncertainties in the model.

2.1.1 Noise: To account for noise in our motion we added small random values to the newly calculated particle positions particle-wise. For each particle we picked a random value in the range $[-0.3, 0.3]$ for x and y, (ϵ_x, and ϵ_y respectively) and a random value in the range $[-\pi/15, \pi/15]$ for θ, (ϵ_θ). We tested both Gaussian and uniform distributions to pull the random values from. Unsurprisingly, pulling random values from a Gaussian provided quicker convergence in the overall localization than pulling random values from a uniform distribution. This makes sense because Gaussian noise implies a higher degree of confidence in the odometry measurements. Refer to section 3.2 for further discussion on testing.

2.2 Sensor Model

The sensor model is important in deciding which particles should be propagated and which particles should be discarded. It gives us a way to quantify how likely the robot is at a given particle's position given our LiDAR data and a reference scan in a simulated environment. The probability of our robot being at a particle at time k given scan data z_k, a map m and position

x_k, assuming LiDAR beams are independent of one another is:

$$p(z_k|x_k, m) = \prod_{i=1}^{n} p(z_k^{(i)}|x_k, m) \tag{3}$$

The probability of each particle $p(z_k^{(i)}|x_k, m)$ can be computed as the sum of $\alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m)$, $\alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m)$, $\alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m)$, $\alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m)$.

$$p_{hit} = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^{(i)}-d)^2}{2\sigma^2}\right) & 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$p_{short} = \begin{cases} \frac{2}{d}\left(1 - \frac{z_k^{(i)}}{d}\right) & 0 \leq z_k^{(i)} \leq d, d \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

$$p_{max} = \begin{cases} \frac{1}{\epsilon} & z_{max} - \epsilon \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$p_{rand} = \begin{cases} \frac{1}{z_{max}} & 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

In these equations d is the expected sensor data computed via a simulated LiDAR scan. ϵ is a small number and $z_k^{(i)}$ is the ith LiDAR beam from the robot at time k. We used $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, and $\alpha_{rand} = 0.12$. We used a $\sigma = 8$.

The sensor model calculates $p(z_k|x_k, m)$ for each particle and returns an array of these probabilities to be used in the particle filter.

2.2.1 Precomputed Table: Calculating these probabilities on the fly for each particle would be too slow. In order to compute the values $p(z_k|x_k, m)$ for each particle quickly, we discretized our function into a 201 by 201 table where columns corresponded to d and rows corresponded to $z_k^{(i)}$ (the two inputs to our function). This meant that our ϵ would be 1 simplifying $p_{max} = 1$ if $z_k^{(i)} = z_{max}$ else 0. It also meant that our z_{max} would be 200, and that we would need to not only normalize the columns of the matrix but also the output of p_{hit} across the whole table. Having a pre-computed table also meant that we would need to convert from meters to "pixels" in the table each time we wanted a value from it. This was achieved by dividing the LiDAR data by the map resolution times the lidar-to-map scale.

2.3 Particle Filter

Andy Li

2.3.1 Bayes Filter Derivation: Our method relies on the characterization of the robot pose as a probability distribution, known as the *belief*, based on data from exteroceptive and proprioceptive sensors. In probabilistic notation, we can write it as:

$$\mathbb{P}(x_k|u_{1:k}, z_{1:k}) \tag{8}$$

where x_k is the robot pose at time k, $u_{1:k}$ is the control input from time 1 to k, and $z_{1:k}$ is the sensor data from time 1 to k. The control input and sensor data are given.

2.3.2 The Particle Filter:

2.3.3 Implementation: In our robotic system, the control input is the odometry data from the encoders and Inertial Measurement Unit of the racecar. The sensor data is data from the racecar's 2D Lidar scans. Therefore, our particle filter is written as a Robot Operating System node that subscribes to the odometry and Lidar data topics. Analogous to the Predict and Update steps of the filter in theory, we created two callbacks for the odometry and Lidar topics, respectively. Assuming that our particles have been populated (more details in 2.3.4), the Predict and Update steps simply apply our previously written frameworks.

In the Predict callback, we use the motion model outlined in 2.1. We convert each particle pose into a transform matrix and then multiply it by the transform matrix given by the odometry data. Noise is added in this step to make each particle update slightly differently, to model the inherent error in our odometry estimation. Over time, without any correction from the Update step, the variance of the particles will increase, which matches our desired behavior: without external sensor data, our uncertainty of the belief will only increase. Whenever external sensor data is given, the particles will cover a larger area, increasing the likelihood that one is close to the ground truth pose, even if the actual location of the robot differs from what unfiltered odometry data would predict.

In the Update callback, we update the associated probabilities of each particle using the Lidar data. Given the scan from the robot, we use the sensor model outlined in 2.2 to compute the probability of the each particle being the ground truth by comparing the actual scan with the simulated scan from the pose of that particular particle. Once all the probabilities are computed, we resample the collection of particles. Whenever each particle in the new collection is selected, the probability of an old particle being picked is equivalent to its associated probability of being the ground truth pose. Over time, this theoretically ensures convergence around the ground truth pose, as particles that are closer to the ground truth have a higher probability of their simulated scans matching the real Lidar data, and thus a higher probability of being the ground truth pose. They will get repopulated more compared to particles with higher error relative to the ground truth pose, and even with the added noise of the Predict step, the particles in the filter decrease in variance around the ground truth pose.

2.3.4 Parameters and Tuning: In our implementation of the particle filter, the main contributor to computational complexity is the number of active particles being tracked. We

chose to use 100 particles in our implementation to keep our runtime fast on the hardware of the racecar. Though adding more particles could possibly increase ability to converge regardless of the initial distribution, we instead opted for efficiency. Since the particle filter needs to accurately track the robot's movement at high speeds, the priority is to ensure its ability to run at least as fast as the Lidar data, at 20 Hz. To guarantee convergence around the initial pose despite the smaller number of particles, we set the initial 100 particles to be a random distribution around an initial pose estimate that is set by the user before the filter is initialized. This avoids the computationally expensive kidnapped robot problem entirely, which is optimal in our specific use-case of the particle filter.

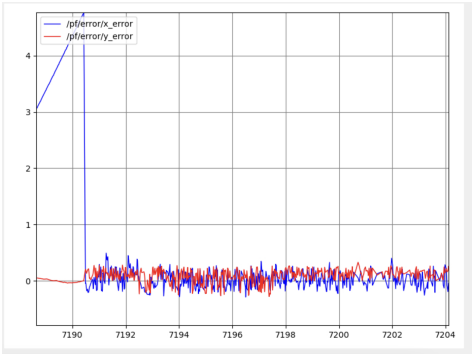

Fig. 1. Cross-track error when taking pose with max probability

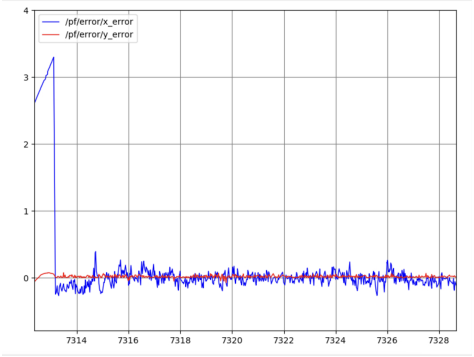

Fig. 2. Cross-track error when calculating pose with weighted average

Natalie Huang and Andy Li

In order to effectively utilize the output of the particle filter, we also had to determine what it should return as the best estimate for the robot pose given all the particles. To do this, we tested two different methods of determining the "average" pose: taking the pose with maximum probability, and taking a weighted average of all particles, where the "weight" assigned to each possible particle was its associated probability. To evaluate the two approaches, we ran the filter using both methodologies in the RViz simulation and graphed the respective cross-track error in Figures 1 and 2. We found that the variation in cross-track error was smaller using the weighted average approach, but that both approaches resulted in error converging around zero.

3 Experimental Evaluation

Natalie Huang

3.1 Unit Tests

We initially tested our implementations of the sensor and motion model through unit tests provided by the course staff to confirm that our approach was correct. However, these unit tests do not test these models in the presence of noise in the odometry data, so we conducted extra tests described below.

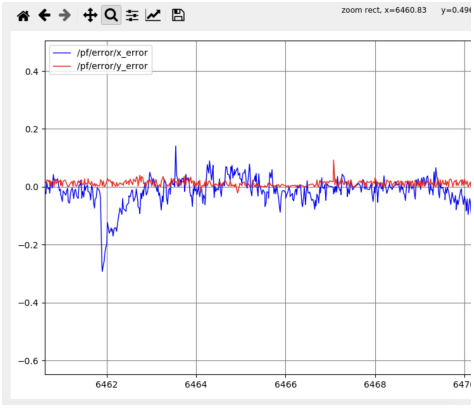

Fig. 3. Cross-track error with added noise distribution $\mathcal{N}(0, 0.04^2)$

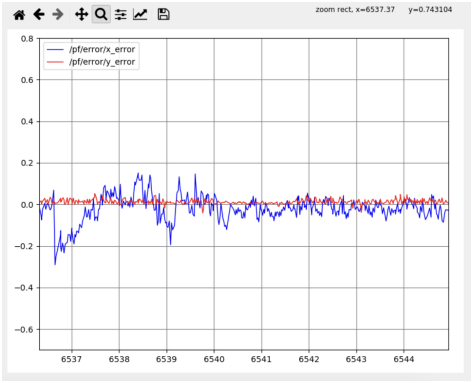

Fig. 4. Cross-track error with added noise distribution $\mathcal{N}(0, 0.06^2)$

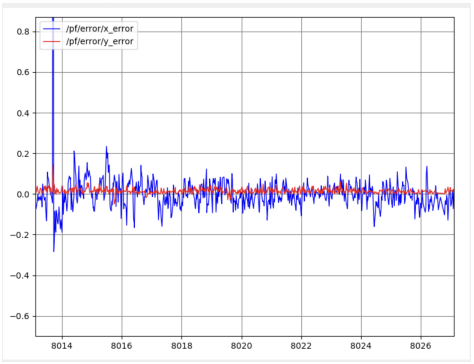

Fig. 5. Cross-track error with added noise distribution $\mathcal{N}(0, 0.1^2)$

3.2 Testing in Simulation

We tested our implementation in simulation using the wall follower. To quantitatively assess the performance of our algorithm, we graphed the position and orientation error by comparing our estimated pose with the pose provided by the ground-truth odometry data. We used these graphs to test the robustness of our implementation in the presence of different noise distributions applied to the motion model.

3.2.1 Gaussian Noise: We tested different Gaussian noise distributions centered around 0 with varying standard deviations, and graphed the cross-track error below. By comparing the performance of the car to the distribution of added noise, we can evaluate whether our implementation is robust to real-life odometry noise. The cross-track error corresponding to each distribution is shown in Figures 3, 4, and 5.

We tested adding noise distributed normally with standard deviations up to 0.1. We found that the convergence rate, which we defined as time until steady-state error, remained steady across the distributions and did not exceed 0.3 seconds. We noted that the variance of the cross-track error grew with the variance of the added noise, but the mean error remained at 0 and the maximum error was 0.2 after convergence at most. We concluded that our implementation was on average accurate to the ground-truth pose and the mean error remains close to 0 despite increases in variance of noise. A video of the particle filter working in simulation can be found here.

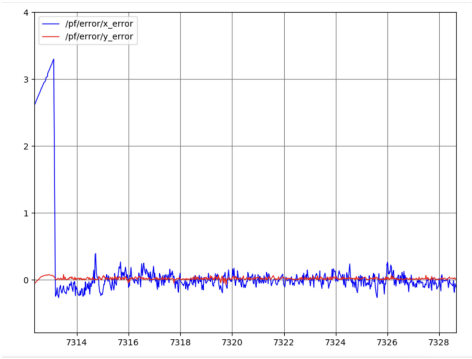

Fig. 6. Cross-track error with added noise distribution *U(-0.5, 0.5)*

3.2.2 Uniform Noise: We additionally tested uniform noise distributions up to a range of $[-0.5, 0.5]$. Once again, we did not see a huge difference in convergence rates or average error, meaning that our implementation is robust to noisy odometry.

3.3 Testing in Real Life

We tested our implementation on our actual racecar. Some changes were required to implement this. First, the coordinate axes of the car in real life were flipped, and we had to account for this by processing the odometry data slightly differently. Furthermore, to visualize the actual LiDAR points detected by the racecar, we had to publish additional transforms between the map frame and racecar frame.

Unlike in simulation, we did not have a reliable method of determining the true location of the robot. Therefore, evaluation on this portion was mainly qualitative. We placed the robot at distinguishable locations, like corners or wall indents, and approximated the initial pose in simulation. As the robot moved around the basement, we ensured that its position in the simulation matched the position in real life. Another qualitative metric we used was analyzing the visualization of the LiDAR data, as displayed in Fig. 7.

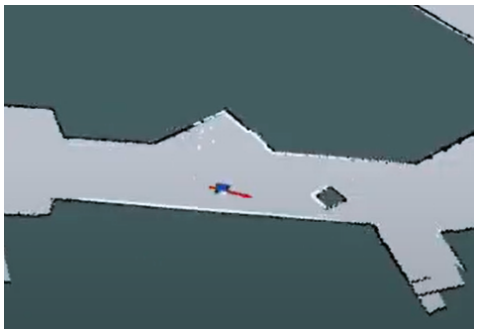

Fig. 7. Screenshot from RViz visualization of real life test. The white dots represent the LiDAR points in the frame of our estimated pose. Since these dots align very closely with the walls of the map, the estimated pose is highly likely to be correct.

A video of the real life test and corresponding RViz visualization can be found here.

4 Conclusion

Harmanpreet Kaur

In this lab, we successfully implemented all parts of the Monte Carlo Localization algorithm. We used our motion model to update the positions of the particles, our sensor model to apply likelihoods to each particle, and our particle filter to use both these models to output the most likely pose for our racecar. We have proven that we can use this implementation to determine the pose of our car in the Stata basement using a known map of it, which will aid us for the final challenge.

Next, we will move on to the Lab 6: Path Planning. With full confidence in our particle filter, we believe we can use it alongside pure pursuit to follow a predefined trajectory. With other parts of the lab, we will use it to enable real-time path planning and execution. hello

5 Lessons Learned

5.1 Andy Li

TODO:CI lessons
TODO:tech lessons

5.2 Antonio Avila

TODO:CI lessons
TODO:tech lessons

5.3 Harmanpreet Kaur

Over the course of this lab, I've learned many valuable Communication-Intensive lessons. A majority of it has been how to write a report like this, including how to site important figures and caption data well.

I've enjoyed working on this lab as a whole, and being able to take lessons taught in lecture and see them transformed into real-life code. In this lab in particular, I've learned a lot about the Monte Carlo algorithm, how the predicting pose is calculated, as well as it's advantages and limitations.

5.4 Mia Tian

Because I was out of town for a lot of the past two weeks (spring break, graduate school visit, eclipse), a communication-intensive challenge for me was to figure out how to contribute at various stages of the technical and communicative process. I learned that although I might not be able to help write the initial code, I could help to test it. Likewise, although I was not able to write a lot of the report, I could help by proofreading.

Regarding the technical aspect, I enjoyed learning about Monte Carlo Localization. I also learned some debugging tools. For example, for the motion model, it was useful to run the model without noise. This helped us find and isolate the problem.

5.5 Natalie Huang

Overall, I enjoyed working on this lab and was able to learn new skills, both technically and communication-wise. Since most of the coding was done over spring break, I took on more of a debugging and evaluation role, and had to determine and collect qualitative and quantitative data that demonstrated the effectiveness of our implementation. We also did a better job of modularizing the work, especially with the help of the unit tests, so it was much easier transitioning from simulation to the real racecar this time compared to previous labs. Teamwork and communication wise, I felt like there were less tasks this time and rather than splitting up, we had to debug and work together a lot more. Additionally, I gained experience in writing a lab report.