Lab 5 Report: Monte Carlo Localization Utilizing LiDAR and Particle Filtering

Team 3

Thomas Buckley
Fiona Wang
Sriram Sethuraman
Ben Lammers

6.4200 RSS

April 11, 2025

1 Introduction

Ben Lammers

As we continue to develop, build, and test algorithms for our autonomous vehicle, having access to information about the robot's position will allow us to solve more complex problems.

In Lab 5, our goal was to design a localization algorithm—one that could determine the robot's position on a map using an initial guess, odometry data, Light Detection and Ranging (LiDAR) data, and the map itself. We decided to take on an approach that incorporated all four of these elements. We forecasted our approach would be along the lines of creating a series of guesses around initial estimate, moving those guesses around based on the robot's motion, and evaluating those guesses based on how our LiDAR data match our map. After this evaluation, we would have an estimate of the robot's position. We forecast success as giving a smooth and accurate estimate of the robot's position.

As our platform for development, test, and evaluation, we continue to use the same racecar for this lab as in past labs, which is a 4-wheel (front-wheel steering racecar) powered by an NVIDIA Jetson running ROS2.

2 Technical Approach

Ben Lammers, Sriram Sethuraman, Fiona Wang

The technical task was to publish the real-time position of the robot on a provided map, given an initial position guess, odometry data, and LiDAR data. Our approach relied on the Monte Carlo localization method. First, we take our initial guess of the position and orientation and sample an array of 200 particles (each with a different orientation and position) from a normal distribution centered at that guess. Next, we update those particles based on the motion of the robot using a motion model. Then, we calculate the probability of each particle representing the correct position and orientation of the robot using a sensor model that takes in LiDAR data. A particle filter then determines which particles are most likely to represent the true position, publishes a position estimate (from the average of those particles), and resamples the particles to concentrate around the most likely positions. The resampled particles are fed back into the motion model, and the process repeats.

2.1 Motion Model

The goal of the motion model is to use odometry data to update our current particles so that they reflect the movement of the robot. The motion model takes in an array of particles, as well as odometry data, including estimated changes in position (Δx, Δy, $\Delta\theta$). This array is either from the previous iteration's particle filter (described below) or sampled from a normal distribution centered at our initial guess position.

After receiving the array of particles, the motion model uses odometry data to generate updated particle positions. To account for uncertainty in the odometry readings, we add noise to the transformation equations. The noise not only captures error accumulating in the odometry data but also helps prevent premature convergence of the particles onto a single position and orientation. With a greater range of position hypotheses, the sensor model can evaluate a broader range of likely positions around the estimated state, improving overall accuracy. The final equations, with noise incorporated, are shown below.

$$x_{i+1} = x_i + (\Delta x + \varepsilon_x)\cos\theta_i - (\Delta y + \varepsilon_y)\sin\theta_i \tag{1}$$

$$y_{i+1} = y_i + (\Delta x + \varepsilon_x)\sin\theta_i + (\Delta y + \varepsilon_y)\cos\theta_i \tag{2}$$

$$\theta_{i+1} = \theta_i + \Delta\theta + \varepsilon_\theta \tag{3}$$

As we can see above, the robot's motion data are transformed by the previous estimate of θ, denoted by θ_i. We also add noise (ε) to our updates. For each particle noise is sampled from a normal distribution for x, y, and θ:

$$\varepsilon \sim \mathcal{N}(0, \sigma^2) \tag{4}$$

The standard deviation used for our normal distribution, σ, was found experimentally by testing multiple values in simulation, which is discussed further Section 3.1.

When implemented in Python, these equations are vectorized. Thus, all the math runs efficiently under the hood using NumPy's C-optimized operations. The output of the motion model is a new set of particles, each representing a hypothesis for the robot's current position and orientation after accounting for the estimated motion and added noise. These updated particles are then passed to the sensor model.

2.2 Sensor Model

The goal of the sensor model is to determine the probability that each particle is the pose of the robot. To do this, we are given the map, the robot's most recent LiDAR scan, and the current particles poses as inputs. The output is an array containing a probability corresponding to each particle. This component is essential for our particle filter to be able to accurately localize itself without relying on pure dead reckoning, which is prone to accumulated error.

The sensor model consists of two main components that work together: the precomputed probability table and the ray casting.

Ray casting is used to get ground-truth LiDAR measurements at each particle. It essentially computes what the race car's LiDAR would "see" if it were at that particle's pose. Specifically, for each particle, virtual LiDAR beams are simulated to be cast from its position. The simulated LiDAR beams then produce expected sensor readings. We can then use the probability distribution to compare the expected reading for each particle with the sensor data from the race car using the probability distribution.

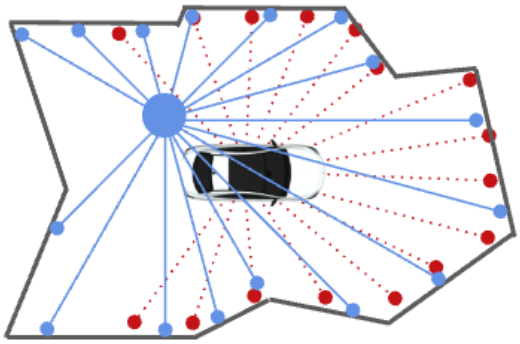

Figure 1: Raycasting visualization. The blue circle is the particle and the blue lines represent ground truth LiDAR measurements computed through ray casting. The red lines coming out of the car represent real-world observed LiDAR data.

The likelihood of the robot being located at a particle's pose is the same as the probability that the ground-truth LiDAR readings x_k match the observed LiDAR readings from the race car z_k. To compute this, we first find the probability of each individual LiDAR beam's length being the ground truth from the particle given the real observed length. These individual probabilities are then multiplied together to give the final probability for each particle.
Putting this logic into equations the goal is:

$$p(z_k|x_k) = \prod_i p(z_k^{(i)}|x_k^{(i)}) \tag{5}$$

at each particle.

To compute the likelihood of observing a LiDAR measurement $z_k^{(i)}$ given the ground truth $x_k^{(i)}$ and map m, we model it as a combination of four probability distributions (as seen in lab instructions):

1. **Accurate measurement**: Modeled as a Gaussian around the expected range d:

$$p_{\text{hit}}(z_k^{(i)}|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^{(i)}-d)^2}{2\sigma^2}\right), & 0 \le z_k^{(i)} \le z_{max} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

2. **Short measurement** due to unexpected obstacles:

$$p_{\text{short}}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{2}{d}\left(1 - \frac{z_k^{(i)}}{d}\right), & 0 \le z_k^{(i)} \le d \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

3. **Max-range measurement** representing missed detections:

$$p_{\max}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{\epsilon}, & z_{max} - \epsilon \leq z_k^{(i)} \leq z_{max} \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

4. **Random measurements** modeling sensor noise or unexpected scenarios:

$$p_{\text{rand}}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}}, & 0 \leq z_k^{(i)} \leq z_{max} \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

The final measurement likelihood combines these distributions with respective weights:

$$p(z_k^{(i)}|x_k, m) = \alpha_{\text{hit}} p_{\text{hit}} + \alpha_{\text{short}} p_{\text{short}} + \alpha_{\max} p_{\max} + \alpha_{\text{rand}} p_{\text{rand}} \tag{10}$$

with the constraint:

$$\alpha_{\text{hit}} + \alpha_{\text{short}} + \alpha_{\max} + \alpha_{\text{rand}} = 1 \tag{11}$$

In our implementation the following values of alpha and sigma were used:

$$\alpha_{hit} = 0.74, \quad \alpha_{short} = 0.07, \quad \alpha_{max} = 0.07, \quad \alpha_{rand} = 0.12, \quad \sigma = 0.5$$

A discretized grid consisting of 200 values of $z_k^{(i)}$ and 200 values of $x_k^{(i)}$ was created, and using these atan2s, all 40,000 values of $p(z_k^{(i)}|x_k^{(i)})$ were calculated ahead of time and stored in a lookup table for use in calculation, as seen below in Fig. 2.

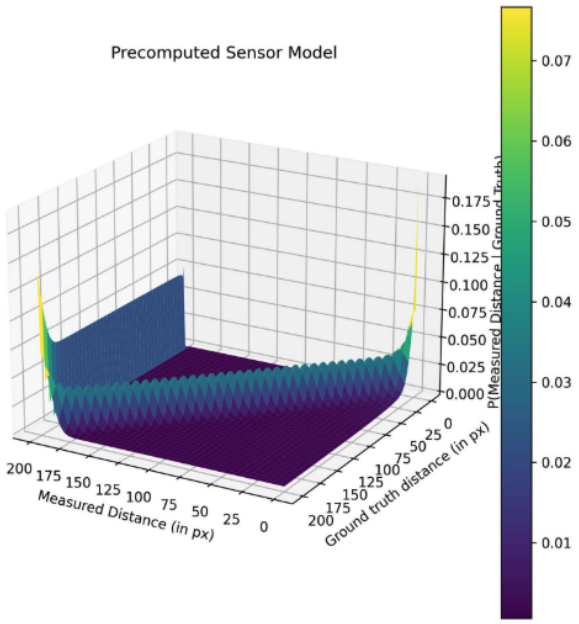

Figure 2: Precomputed probability table. Shows the 40,000 precomputed valyes of $p(z_k^{(i)}|x_k^{(i)})$, which are stored in a lookup table. The probability is the highest when the measured distance is equal to the ground truth distance, as shown by the ridge in the graph.

In summary, we are trying to find the likelihood of each particle being at the robot's position. To do that we generate what each particle would see in LiDAR, and compare that with the real LiDAR data. At each particle, we multiply all the individual probabilities of each LiDAR beam matching the measured value (obtained using look-up table). We then normalize and return this array of probabilities to the particle filter.

2.3 Particle Filter

Our final task for our Monte Carlo localization method was to assemble our particle filter using our motion and sensor model. To do this, we implemented multiple callback functions to initialize our particles, compute probabilities, re-sample particles, and publish our pose. First, we select an initial pose using the 2D Post Estimate tool in RViz. Using the selected pose, we initialize our particles using a multivariate normal distribution surrounding the chosen point. Although the particles will be resampled in other callback functions based on the laser scans and odometry data, we chose to initialize with a uniform distri-

bution to prevent the kidnapped robot problem from occurring.

After initializing, our odom_callback function determines the change in position using the twist component of the OdometryMessage. Inputting the change in position, we update the particles using the motion model explained above. Next, our laser_callback function takes a laser scan and first, downsamples the number of scans to 99. We chose to downsample from over 1000 to under 100 scans because we observed that it helped reduce how "peaked" our probabilities were and it required less ray casting. The callback function then computes the probabilities using our sensor model, raises the output by a power of $\frac{1}{3}$, and normalizes the probabilities. By raising the probabilities to a power of $\frac{1}{3}$, we were able to "squash" and spread out the probabilities, which prevented high probabilities from causing a quick and erroneous convergence.

Finally, we publish our new pose in our publish_pose function by finding the 'average' position. To do this, we find the weighted average of the x and y values based on our probabilities and find the circular mean for our θ value. To do this we used the formula:

$$\theta = \text{atan2}(\frac{1}{n}\sum_{j=1}^{n}\sin(\alpha_j), \frac{1}{n}\sum_{j=1}^{n}\cos(\alpha_j)) \tag{12}$$

Additionally, we also published our particles to RViz to help with visualization and debugging.

3 Experimental Evaluation

Fiona Wang, Thomas Buckley

In order to evaluate our Monte Carlo Localization, we wanted to evaluate each of our models and the particle filter in both simulation and the real world. Before testing the particle filter, we first utilized the provided unit tests to ensure that our motion and sensor models were performing without any major issues.

3.1 Simulation Evaluation

To evaluate our particle filter, we first tested in the simulation. Our most initial tests focused on using the 2D Pose Estimate tool to check that our localization was accurately estimating the robots position. After debugging initial issues, we used our wall follower to qualitatively and quantitatively evaluate our localization in simulation. To do this, we focused on testing two paths: Stata Path A, a path with multiple turns and indents, and Stata Path B, a straight wall. By recording Rosbags, we were able to replay the robot's path and visually inspect if our particle cloud and odometry arrows were accurately updating and following the racecar's path. We graphed the results of our tests, overlaying

the estimated path with the actual path (the ground truth position) and determining the error between them. These graphs are displayed in Fig. 3 below, and it can be seen that the estimated path matches the actual path relatively well. The estimated position only drifted away from the ground truth (as seen in path b in Fig. 3) when the robot went through a wall.

Following these initial plots, we aimed to quantify the impact of introducing noise into our motion model. We incorporated Gaussian noise into the robot's motion to reflect uncertainty during movement. We observed that larger noise values caused the particles to spread out more broadly, allowing the particle filter to recover effectively from divergent paths. Conversely, very low noise levels led particles to converge tightly, which increased the risk of the robot's true position falling outside this narrow particle distribution.

To systematically evaluate the effect of noise, we tested various values of the noise standard deviation in simulations and measured the average distance error between the particle filter's estimated position and the ground-truth position. As summarized in Table 1, the mean position error slightly increased with higher noise levels on Stata Path A. This result aligns with our expectations, as introducing more noise naturally contributes additional uncertainty to the estimate.

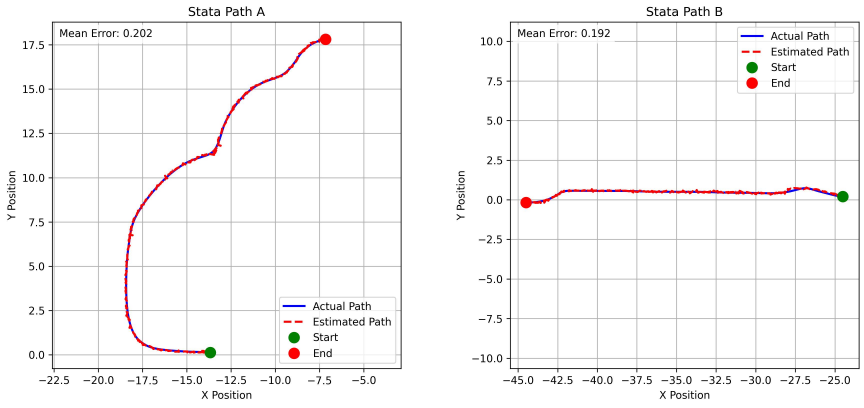

Figure 3: **In Simulation Trials.** Our localization was evaluated in simulation by running the wall follower on two paths (path a on the left and b on the right) and comparing the estimated path with the actual path (ground truth position).

3.2 Real World Evaluation

When moving from simulation to the real world, our particle filter worked but had a major problem. The 200 particles in our filter would very quickly converge

Table 1: Different Values of Noise std in Motion Model on Stata Path A

x, y noise	theta noise	Error (Mean ± Std Error)
0.05	0.1	0.1273 ± 0.0539
0.10	0.1	0.1980 ± 0.0559
0.15	0.1	0.2124 ± 0.0670
0.20	0.1	0.2124 ± 0.0670

to the same pose. We discovered that this rapid convergence occurred because our initial probability distribution contained only a few poses with disproportionately high probabilities. Consequently, the resampling step repeatedly selected these high-probability particles, causing premature convergence and a loss of diversity within the particle set. This made the particle filter ineffective in real-world testing, as it lacked the necessary robustness to recover from early inaccuracies and failed to adequately represent uncertainty, severely limiting its localization accuracy and reliability.

We made three major changes to fix this. First, we increased the standard deviation of the Gaussian noise in the motion model from 0.05 to 0.10, as this showed good experimental results. We kept the standard deviation of theta constant throughout these experiments, as our goal was to increase the spatial distance between particles for increased robustness. Next, we added a time delay for when we resampled particles that we could adjust. We noticed that this resulted in "pulsating" clusters of particles, which would gradually grow, then converge once we resampled. However, this performed much better than reasmpling on every update of our sensor model. Finally, we "squashed" the probability distribution by putting it to the power of $\frac{1}{3}$ as mentioned in the technical approach section. This further increased the diversity of sampled points and maintining good accuracy.

To assess real world evaluation, we conducted two experiments where we recorded the path of the robot while recording a Rosbag. We then compared the recorded video to the estimated pose from our particle filter qualitatively. Our particle filter was highly accurate over long durations (multiple minutes) and for complex paths in the Stata basement.

4 Conclusion

Sriram Sethuraman

Our implementation of localization using particle filtering has successfully enabled our race car to accurately determine its pose within the basement of Stata. The combination of our odometry-based motion model and the LiDAR-based

sensor model allowed our particle filter to effectively update an accurate estimate of the car's pose in real time.

Qualitative analyses conducted in both simulation and the real world demonstrated our algorithm's ability to track the pose of our moving car in real time, as well as compensate for noisy environments. These results also showed close alignment with estimated positions and actual trajectories. Quantitative results further backed up our findings, showing low cross-track error between ground truth data and estimated poses in simulation. However, obtaining ground truth data in real-world testing proved to be a significant challenge.

Future improvements will focus on more robust validation in real-world scenarios, measuring robot positions and planning trajectories to more accurately estimate real-world error. Additionally, particle resampling was suggested as a method to make the algorithm more robust to fast convergence. We learned valuable technical and teamwork related lessons that we will also carry with us into the next lab.

5 Lessons Learned

Fiona Wang
During this lab, I was exposed to many new concepts—primarily the Monte Carlo Localization algorithm. This was a completely new idea to me so there was definitely a steeper learning curve especially when it came to making sure that I understood all of the math and models that we needed to implement. Some learning strategies that helped me included watching visualizations that I found online and talking to my teammates about how we needed to approach our task. Additionally, I also learned about the importance of unit tests when implementing our particle filter. Knowing that our motion and sensor models were working helped simplify the debugging process and eliminate potential problem areas. However, there were still times when the unit tests were not completely comprehensive, and we needed to go back to edit and tune our motion and sensor models. Overall, I am proud of how our localization turned out and I am excited to assemble all of the different components during the next lab!

Ben Lammers
I learned real-world applications of my prior Bayesian probability coursework. I learned how math can model uncertainty in real-world data, such as odometry and sensor readings, but still produce precise results. Implementing these algorithms on hardware helped solidify my understanding of these concepts, making them feel more practical and tangible. From a communication perspective, I learned the importance of proper documentation and how to collaborate effectively. Multiple times, my teammates and I found ourselves rewriting each other's code simply because we did not understand how it worked or that it was already tested and complete. In the future, I plan to ensure that all code is

well-documented with clear explanations of the logic behind each function and method. I want to prioritize communication among team members, ensuring that we discuss our approaches and progress regularly to avoid doing the same work.

Thomas Buckley
During this lab, I gained insight into the value of having a standardized testing methodology and clearly defined evaluation pipelines. As our system continues to grow in complexity, consistently and accurately assessing performance becomes increasingly challenging. For example, we needed to run many systems at the same time (localization, simulator, wall_follower, the evaluation pipeline), as well as have a standardized starting and ending position for the car. Additionally, as system complexity grows, we needed to be more creative in our evaluation metrics (especially in the real-world where we can't easily obtain a ground truth for robot position). Building out a scalable way to do this evaluation was difficult, but helped us rapidly evaluate robot parameters which will save time later on.

Sriram Sethuraman
A robust understanding of the math behind the algorithms we are implementing makes the programming and debugging much more efficient. I also learned that communication of progress and code comments are very important so we know what work has been done and we don't do any redundant work.