Lab 5 Report: Localization

Team 19

Michael Wong
Alan Yu
Joshua Sohn
Owen Matteson
Ragulan Sivakumar
Shenzhe Yao

6.4200: RSS

April 13, 2023

1 Introduction - MW

In this paper, we present the results of our lab experiment aimed at solving the challenging problem of robotic localization. Specifically, we implemented the Monte Carlo Localization algorithm, also known as MCL or particle filter, to determine the orientation and position of a racecar in a known environment using solely lidar detection and odometry data.

Localization is a critical competency required by autonomous robots as it is essential for making decisions about future actions based on their location. Despite its apparent simplicity, localization remains an active research area in robotics, and its solution is far from trivial. In this paper, we discuss our implementation of the particle filter algorithm and present our experimental results, demonstrating its effectiveness in localizing the racecar throughout the Building 32 basement.

2 Technical Approach

2.1 Problem Formulation and Approach - AY

In this problem, we assume we have access to a map M and our objective is to approximate the true position (and orientation) x_k of the racecar at each time step k within M. At time each k, we are given the following two quantities.

- Δx: the odometry data consisting of the velocity v and angular speed ω
- $\{z_k^{(i)}\}_{i=1}^{M}$: M laser range scans from the LiDAR.

A simple idea to represent the uncertainty in the estimated pose is to describe x_k as a *probability distribution* conditioned on odometry and laser scans,

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

In order to update uncertainty given new data, we need additional assumptions. Hence, we define the **motion model** to be the distribution $\mathbb{P}(x_k|u_k, x_{k-1})$ defined by the user. Similarly, we define the **sensor model** to be the distribution $\mathbb{P}(z_k|x_k)$. Given these quantities, we can use a two-step Bayesian filter to update our beliefs.

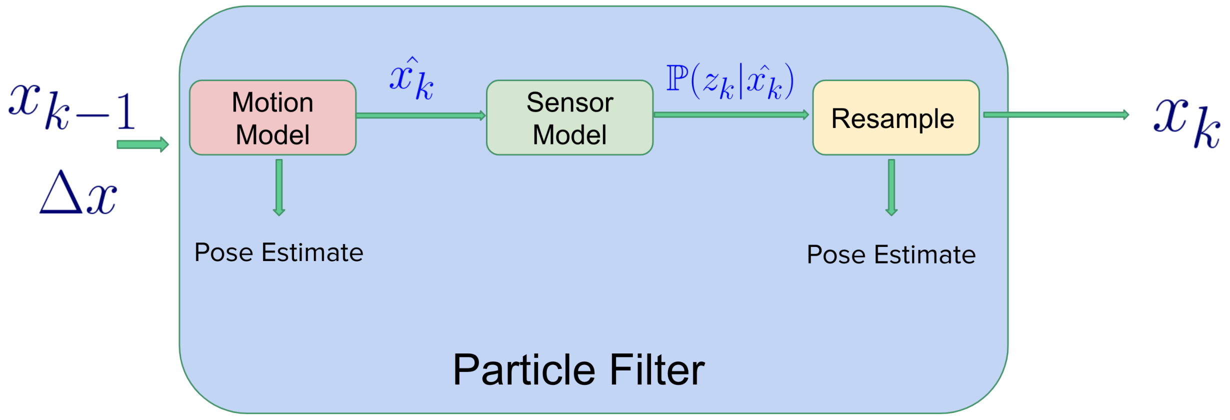


Figure 1: A block diagram showing the plan for the particle filter. We send previous particles and odometry data into the motion model to predict the next set of particles. The sensor model gives us the posterior distribution, which we use to resample a set of likely particles. Using the particles, we publish a pose estimate at each step.

1. Predict: $\int \mathbb{P}(x_k|u_k, x_{k-1})\mathbb{P}(x_{k-1}|u_{1:k-1}, z_{1:k-1})\ dx_{k-1}$.
2. Update: $\mathbb{P}(u_{1:k}, z_{1:k}) \propto \mathbb{P}(z_k|x_k)\mathbb{P}(x_k|u_{1:k}, z_{1:k-1})$.

While this approach is valid, it is intractable for our current problem. Instead, we use **Monte Carlo Localization (MCL)** to approximate $\mathbb{P}(x_k|u_{1:k}, z_{1:k})$ by weights $w_k^{(j)}$ on a set of particles $x_k^{(j)}$, where $j \in [N]$. Here, we have three steps which draw inspiration from the Bayes Filter.

1. Predict: For each $j \in [N]$, sample from the motion model using Δx to predict $\hat{x_k}^j$ from x_{k-1}^j.
2. Update: Using the sensor model and the vector of laser scans z_k, update $w_k^{(j)} \propto \mathbb{P}(z_k|\hat{x_k}^{(j)})$.
3. Resample: Sample x_k from the distribution on $\hat{x_k}$ defined by w_k.

At each step, we output a pose estimate using our particles which will be elaborated on in Section 2.4. The overall pipeline is illustrated in Figure 1.

2.2 Motion Model - AY

In the motion model module, we need to predict the subsequent positions of the particles given odometry data $\Delta x = (\delta x, \delta y, \delta\theta)$. As these are measured in the frame of the racecar, we need to take an additional step to calculate new positions. For each particle $x_{k-1}^{(j)} = (x_{k-1}^{(j)}, y_{k-1}^{(j)}, \theta_{k-1}^{(j)})$, we compute

$$[\hat{x_k}^{(j)}, \hat{y_k}^{(j)}]^T = \begin{bmatrix} \cos\theta_{k-1}^j & -\sin\theta_{k-1}^j \\ \sin\theta_{k-1}^j & \cos\theta_{k-1}^j \end{bmatrix} [\delta x, \delta y]^T + [x_{k-1}^{(j)}, y_{k-1}^{(j)}]^T$$

and return

$$\hat{x_k}^{(j)} = [\hat{x_k}^{(j)}, \hat{y_k}^{(j)}, \hat{\theta_k}^{(j)}]^T,$$

where $\hat{\theta_k}^{(j)} = \theta_k^{(j)} + \delta\theta$. These predictions can now be used in the sensor model where they can be resampled to produce the particles x_k.

In this case, our motion model was deterministic: Δx and x_{k-1} perfectly predict $\hat{x_k}$. However, this is not representative of noise in the real world. In order to allow for more exploration and be more robust to erroneous LiDAR and odometry data, we can add noise $Z_x \sim N(0, \sigma_x), Z_y \sim N(0, \sigma_y), Z_\theta \sim N(0, \sigma_\theta)$ to our odometry data, giving $\Delta x = (\delta x + Z_x, \delta y + Z_y, \delta\theta + Z_\theta)$ and

$$[\hat{x_k}^{(j)}, \hat{y_k}^{(j)}]^T = \begin{bmatrix} \cos\theta^j_{k-1} & -\sin\theta^j_{k-1} \\ \sin\theta^j_{k-1} & \cos\theta^j_{k-1} \end{bmatrix} [\delta x + Z_x, \delta y + Z_y]^T + [x^{(j)}_{k-1}, y^{(j)}_{k-1}]^T$$

and

$$\hat{x_k}^{(j)} = [\hat{x_k}^{(j)}, \hat{y_k}^{(j)}, \hat{\theta_k}^{(j)} + Z_\theta]^T.$$

In our implementation, we used the parameters $\sigma_x = 0.05$, $\sigma_y = 0.03$ and $\sigma_\theta = \pi/20$.

2.3 Sensor Model - RS

Our sensor model assigns probabilities to our list of particles as potential true locations of the racecar. To assign this probability, we shall compare expectation against reality. Our LIDAR scans give a list of distances to objects for 1081 distinct angles about the car, and we are also given an image of the map. From the map, we can calculate what a particle would expect to see from its LIDAR scans as well, so using these two data sets, we can assign probabilities to different particles.

2.3.1 How We Transform Real Distances Into Pixel Distances - RS

Before getting into how the probabilities are calculated, we must first discuss how a real distance d becomes a pixel distance p, for our map image only holds pixels. Let the LIDAR-to-map scale factor be denoted r. As given in the handout, our formula is

$$p = \frac{d}{\text{resolution} \cdot r}.$$

r lies in the denominator, for dividing a LIDAR distance by the LIDAR-to-map scale factor should give the map distance. There is also resolution in the denominator, to help us convert from 3-D to 2-D in raycasting. With that out of the way, we can now work towards assigning probabilities.

2.3.2 Calculating Probabilities for a Single Angle's LIDAR Scan Against the Expected

For a given angle, let z_k be the distance we get from our LIDAR scan at a particular angle, let z_{max} be the max a LIDAR scan can output as a distance, and let d be the distance we expect to sense at that same particular angle. Our model, as given in the lab handout, for how d and z_k can relate is the following:

1. **HIT**: Our LIDAR scan and expected value only differ because of noise. We model this as a Gaussian distribution for noise about the expected via the following equation for Gaussian noise.

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

   We have 0 otherwise, for something must have gone wrong with our LIDAR scan to exceed the maximum possible distance or to return a negative distance.

2. **SHORT**: An obstacle causes the LIDAR to fall short of the expected. As we get farther away from our racecar, less LIDAR scans will hit said obstacle. Thus, we can model it as the following:

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

   The fraction $\frac{2}{d}$ is to normalize the probabilities, which gets them to sum to 1. We model the probability as starting at 1 and descending to 0 and distance d linearly, and we end at d for we cannot be short beyond d.

3. **MAX**: The LIDAR scan accidentally returns the max by default.

$$p_{max}(z_k) = \begin{cases} 10 \text{ if } z_{max} - 0.1 \leq z_k \leq z_{max}, \\ 0 \text{ otherwise} \end{cases}$$

   This makes it so that if z_{max} is returned, we don't harshly punish the particle for a LIDAR mishap. Note that 0.1 and 10 were chosen arbitrarily as given in the handout.

4. **RAND**: The LIDAR scan returns a random value up to its max.

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

   This model makes a uniform distribution, which is exactly what we want for a random number.

Now, we must combine the above into a probability for that one piece of the entire LIDAR scan. We can compute a weighted average of the four quantities enumerated above. As given in the handout, we used the weights of 0.74 for hit, 0.07 for short, 0.07 for max, and 0.12 for rand. To finish, we can once again normalize across d so as to make the probabilities again sum to 1. Altogether, this creates our probability for a given angle of our LIDAR scan distance against the expected we compute from the map.

2.3.3 Calculating Probabilities for the Whole LIDAR Scan Against the Expected

Each LIDAR scan can be broken into its individual distances at each angle. As we've shown above, we can compute the probabilities of each of those individual distances against our expected value. To then bring it all together and calculate the probability for a whole scan, we can multiply the individual probabilities together.

2.4 Combining the Motion and Sensor Model into Our Particle Filter

2.4.1 Initial poses - SY

To initialize our particle filter, we start by receiving the initial pose of the racecar through a ROS pose callback. Once we have the initial pose, we generate a set of particles by sampling a Gaussian distribution centered around the initial pose, with separate standard deviations for the (x, y) positions and orientation. By sampling, we effectively create a cloud of particles around the initial pose, each representing a potential pose for the racecar, capturing the possible variations in the racecar's true position and orientation.

After generating the particles, we store them in a list that will be used for further processing in the particle filter. This list will be updated throughout the localization process as the racecar moves and receives new sensor data, allowing the particle filter to continually refine its estimate of the racecar's pose.

2.4.2 Precomputing Probabilities for Efficient Live Computations - RS

Looking back to our sensor model, the computations involved are intensive, and given how we have to do this ≈ 20 times a second for 200 particles each time, we can't afford to waste time calculating each probability, for we won't have enough time to do so. Thus, instead we should store a table of preset probabilities for given d and z_k values.
Let the table go from 0 to 200, inclusive, for both d and z_k because z_{max} is 200. Store probabilities as designated by the sensor model in this table with the following modification: we should normalize across all d values, 0 to 200, for the hit probabilities to make them sum to 1 before doing the weighted average. This is important, for our model is normalized for a continuous Gaussian distribution, but here, we are discretizing, so it's no longer necessarily normalized. Our hit probabilities could sum to greater or less than 1 and correspondingly count for more or less than the other three categories, respectively, which we would want to avoid. Thus, we normalize the hit probabilities before computing the weighted average.

Then, we converting from meters to pixels, we could also convert each number to an integer from 0 to 200. Thus, we'll be able to index into this table properly to find the probabilities we want.
Note that this will be less accurate than if we did the calculation for non-rounded numbers. However, we could not feasibly do that computation, so we had to make this tradeoff.

2.4.3 Resampling - RS

At each iteration of our sensor model, we want to resample our particles list via our probabilities so as to eliminate unlikely poses and reinforce those that are more likely. For this, we can take our calculated probabilities for each particle, normalize them, and then randomly assign each index in a new particles list of size 200 as a particle from the old list in proportion to the particles' probabilities. This completes resampling and makes it such that our particles list continues to reflect the most likely poses of our racecar.

2.4.4 Preventing Data Conflicts via Thread-Locking - OM

Our initial implementation of the particle filter processed both LiDAR scan data using the sensor model and odometry data using the motion model in parallel whenever the data was received. However, after attempting to run the filter in simulation, we noticed that there was occasionally lagging in the poses of the hypothesis particles behind the ground truth. We speculated that this was due to some odometry data being skipped over or failing to be processed. We tracked the issue back to the fact that the sensor model and motion model were both writing to the same state variable, `particles`, and were frequently conflicting.

ROSpy subscribers run on separate threads, enabling the callbacks to be executed at the same time. This is often a desirable behavior, as parallel processing can decrease computation time; however, in this case, this is the source of the above issue.

The sensor model operates with a non-negligibly greater computation time than the motion model due to the ray tracing it performs on each hypothesis particle. Consequently, if a LiDAR scan was subscribed to and an odometry message was received afterwards, the motion model could potentially run to completion before the sensor model did. This would result in the motion model updating `particles` first and the sensor model overwriting this update afterwards despite the data being received in the opposite order. Therefore, some odometry data would never be expressed in the `particles` array, resulting in the lagging behavior mentioned previously.

The solution to resolving these conflicts is to remove the parallel processing nature of the particle filter. We would like for each message to be processed in full and its output to be stored in the particle filter object before any other message could be processed. Since the two callbacks are executed on separate threads by default, our team took the approach of using thread locking to achieve this goal. By storing a single lock object from the python threading library and utilizing its acquire and release member functions in each callback, we demand that one callback must finish before another is allowed to begin. A generic template callback that uses thread locking is shown in algorithm 1, and a block diagram illustrating the two states of the particle filter is shown in figures 2 and 3.

Algorithm 1 Pseudocode demonstrating the use of thread locking on a generic callback function.

```
1: $L \leftarrow$ threading.lock();
2: procedure GenericCallback(data)
3:     $L$.acquire();
4:     process data;
5:     $L$.release();
```

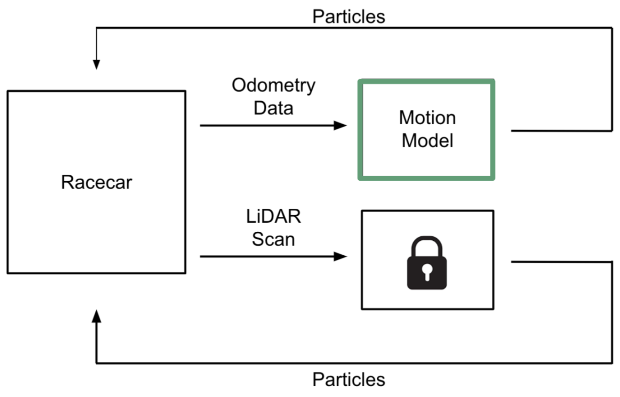


Figure 2: One state of the particle filter in which the motion model is processing odometry data and the sensor model is locked to prevent conflicts.

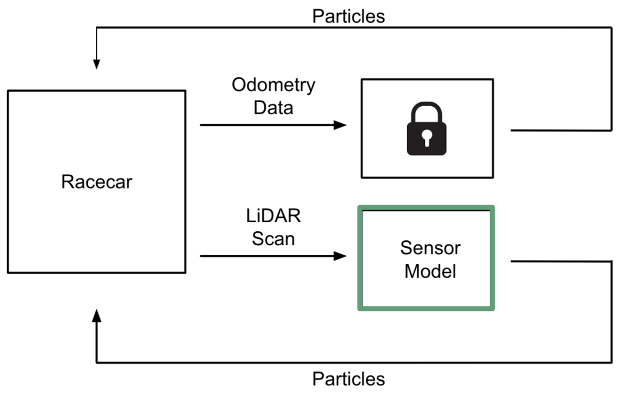


Figure 3: The second state of the particle filter in which the sensor model is processing LiDAR data and the motion model is locked to prevent conflicts.

2.5 Modifications for Deployment in a Real Environment- OM

While the simulation was a generally accurate approximation of a real environment, there were a few inconsistencies when deploying the particle filter to the racecar hardware that needed to be accounted for.

The first major complication was the difference between the LiDAR scan data that is generated in simulation and that received from the hardware. In simulation, we used 100 increments, and therefore 100 individual probabilities, per LiDAR scan. However, when deploying the filter onto the racecar, we noticed that the car's LiDAR sensor records 1000 increments per scan message. We assumed this would only be a simple parameter change to match the number of increments of the ray-traced hypothesis particle scans with that of the hardware. However, as we were multiplying 1000 probabilities together, each of which is relatively small in itself, the result was so small (10^{-300}) that it was interpreted as zero by the computer. In order to solve this, we downsampled each LiDAR scan message by only considering every 10th increment, seen in figure 4, to reduce the number of multiplications done to find the scan probability.

In order to further lessen the chance of numerical instability issues, we opted to use the exponential of the

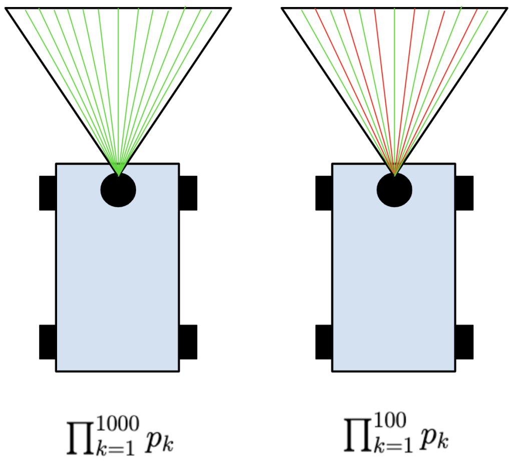


Figure 4: A diagram illustrating the downsampling done on the LiDAR scan messages to avoid floating point error when multiplying the probability of each increment in the scan. Green is used to indicate increments that were included in the calculation while red indicates ones that were skipped over. For the purpose of simplicity, this diagram depicts every other increment being included, while on the car, every 10th is included.

sum of logarithms in place of a standard product, seen in equation 1.

$$\prod_{k=1}^{100} p_k = \exp \sum_{k=1}^{100} \log p_k \tag{1}$$

The final issue we encountered was with the real environment not being a perfect match of the provided static map of the MIT Building 32 basement. For example, the shifting of a vending machine or the placement of a box against a previously empty wall could cause differences between the real environment and the "known" map. These differences, despite appearing to be minor, can cause large discrepancies between a simulated scan at a given pose and the actual scan at the same pose. While this is a general drawback of the Monte Carlo Localization algorithm, we attempted to remedy this by increasing the number of hypothesis particles from 200 in hopes of bettering the accuracy of the localization, as well as tuning the odometry noise parameters. After some qualitative testing on a noisy section of the map, we decided that increasing to 1000 particles and decreasing the x-noise parameter to 0.005 from 0.03.

3 Experimental Evaluation

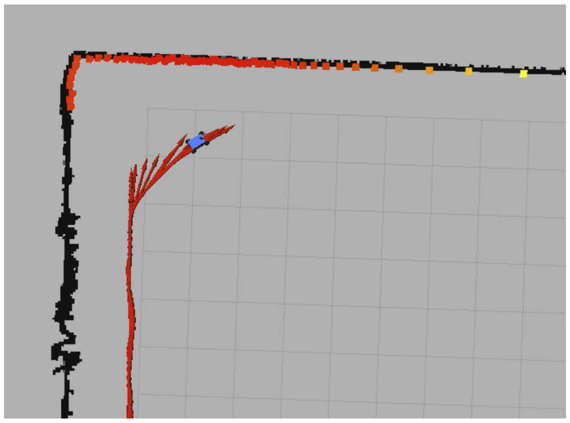
3.1 Simulation Tests - SY

We conducted a series of simulation tests to evaluate the performance of our Monte Carlo Localization implementation. These tests were designed to analysis the effects of adding noise to the particle list on the algorithm's performance, and to assess the accuracy and robustness of our tuned localization algorithm under various conditions.

3.1.1 Test 1: Analysis Localization Performance with Adding Noise Approach

In this test, our goal was to verify whether adding noise to the particle list would increase the robustness of the localization algorithm. The motivation behind this test was to simulate real-world scenarios, where the car may encounter varying levels of noise due to sensor inaccuracies, environmental factors, or other sources of uncertainty. We wanted to ensure that our MCL implementation could adapt to these challenges and maintain accurate localization.

We conducted the test with and without adding noise to the particles, generating localization and ground truth trajectories under situation with noise and without noise (as Figure 12). During the process of the car turning, we noticed that the accuracy of localization becomes larger in the case of adding noise qualitatively.


(a) Localization trajectory without added noise

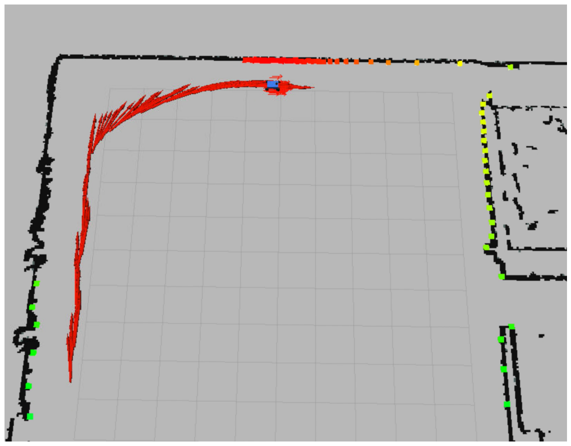
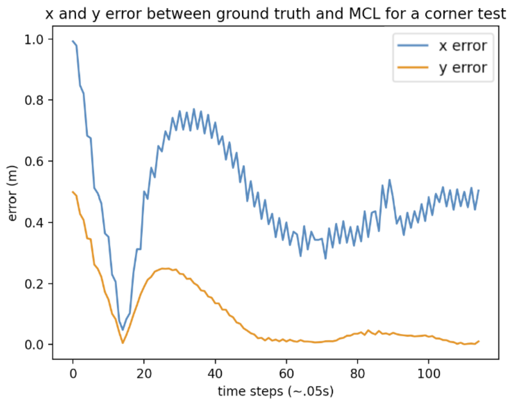
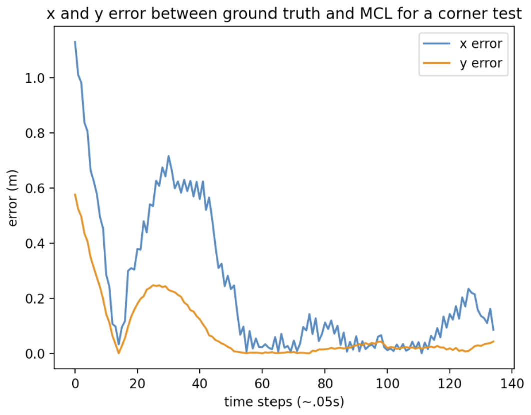

(b) Localization trajectory with added noise

Figure 5: Visualization of Localization trajectories with and without added noise, it can be seen that we have added some noise around the car position in (b) diagram.

To further analyze the impact of noise on the algorithm's performance, we designed the following tests: tuning the noise parameters in the x-direction from 0.01m to 0.05m standard deviation while keeping the y-direction noise at 0.03m standard deviation. The results of error distance between localization and ground truth were visualized in two graphs: *Figure 6 (a)*, illustrating the x and y errors for a simulation corner test with an x noise standard deviation of 0.01m and y noise standard deviation of 0.03m, and *Figure 6 (b)*, illustrating the x and y errors for a simulation corner test with an x noise standard deviation of 0.05m and y noise standard deviation of 0.03m.



X and Y errors with x noise standard deviation of 0.01m and y noise standard deviation of 0.03m



X and Y errors with x noise standard deviation of 0.05m and y noise standard deviation of 0.03m

Figure 6: Error distance on X and Y directions under different noise parameters

Table 1: Different Noise Standard Deviation Test Results

Trial	$\sigma_x = 0.01$, $\sigma_y = 0.03$	$\sigma_x = 0.05$, $\sigma_y = 0.03$
Average X Error (m)	0.45	0.23

The average distance errors in x-direction are showed in Table 1. The comparison between the two graphs and average errors revealed that the x-error between localization and ground truth decreased when the x-direction noise parameter was increased from 0.01m to 0.05m, indicating that increasing the noise parameter values within a certain range can help to reduce noise. This result confirms that adding noise to particle list approach contributes to improved robustness and reduced error in our MCL implementation, enabling it to operate effectively in car turning case. Also, it is reasonable that noise parameter in one direction will not affect the accuracy in other direction.

3.1.2 Test 2: Test on Autograder's Preset Trajectory

The second test was performed using the auto-grader that provided a preset trajectory. The auto-grader ran our localization program and compared the results obtained from our solution, the instructor's provided solution, and the ground truth. The comparison was visualized in a plot (*Figure 7*), where our localization trajectory closely matched both the instructor's solution and the ground truth. This demonstrates that our MCL implementation can accurately estimate the car's pose and perform well under complex trajectory conditions.

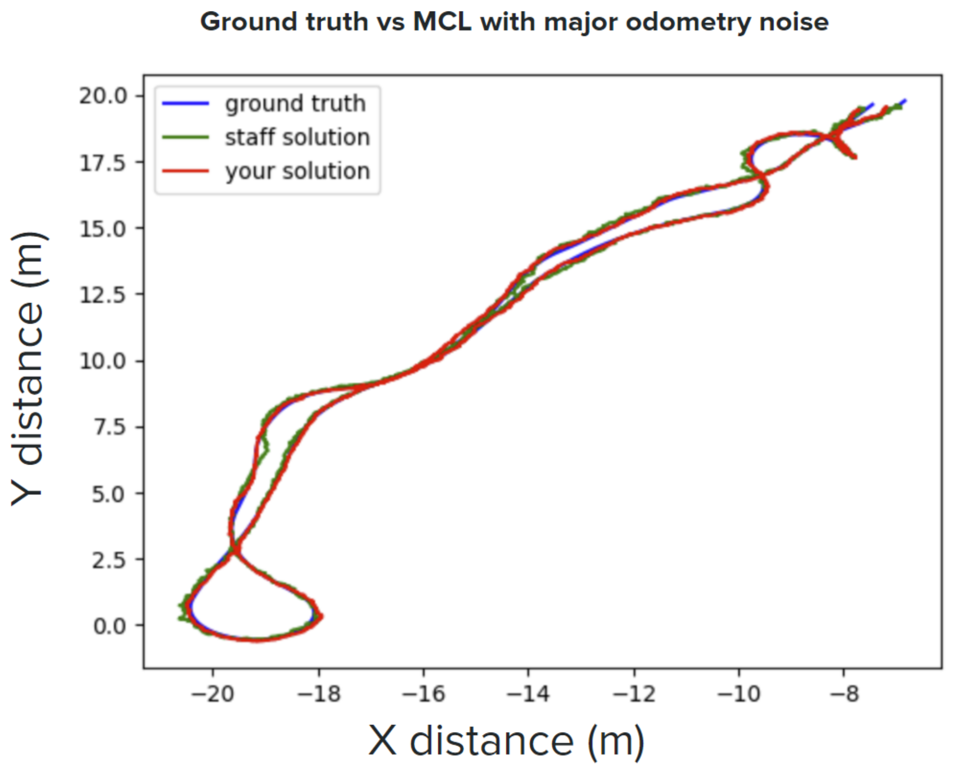


Figure 7: Comparison of our localization solution, the reference solution, and the ground truth trajectory in auto-grader's preset trajectory

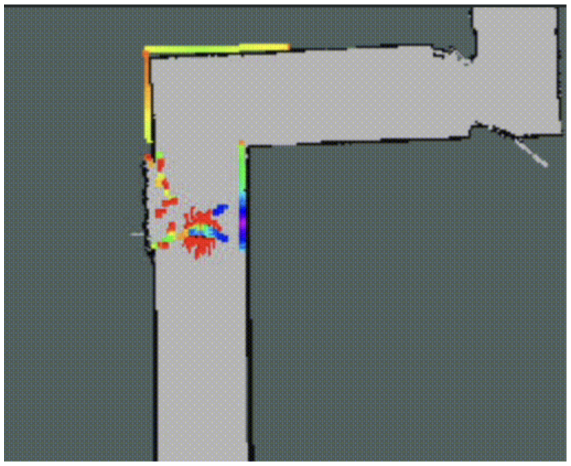
The success of these simulation tests provided us with the confidence that our algorithm could handle more complex and challenging scenarios as in real world.

3.2 Real-World Tests - AY

We also conduct a suite of quantitative and qualitative test cases to evaluate the performance on our deployed implementation. Because we do not have access to the ground truth transformation between map and base link as in the simulation tests, we resort to different metrics to evaluate performance.

3.2.1 Qualitative Results

For each step through the pipeline, we are publishing an estimate of the average pose of the base link. We are also receiving the ground-truth laser scans from the LiDAR. If our pose estimate is correct, the laser scans should exactly line up with the map features (e.g. the walls, indents, corridors, etc.) as we travel through the basement. We noticed that this was indeed the case, as shown in Figure 8.


(a) RViZ LiDAR Scan with Particle Cloud

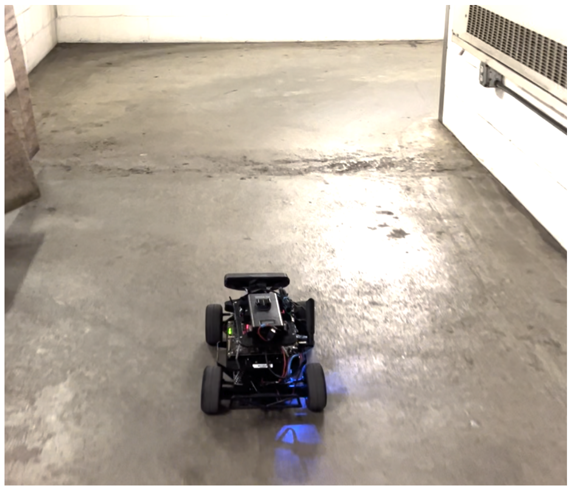

(b) Pose of the Car in the Real World

Figure 8: Visualization of the qualitative performance of our MCL implementation. In (b), the car is moving forward about to turn around the wall. The pose of the car is estimated correctly as shown in (a), since the laser scans align with the wall features. The noise to the left of the car is due to a moving curtain and suggests that the pose estimate is robust to random noise in some cases.

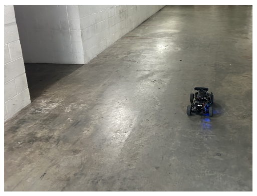
3.2.2 Quantitative Results

We investigate two classes of tests to quantitatively evaluate the MCL, demonstrating that it is robust to noisy odometry and offset initial poses. Specifically, we test:

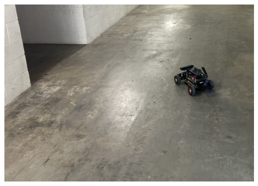
1. Convergence Rate: When placed near a wall feature (e.g. indentation), how long does it take for the particles to converge to a precise estimate of the pose? Can the car still estimate this effectively when it is offset from its initial pose estimate by position and/or orientation?

2. Cross Track Error: Given two points on the map, p_1 and p_2, how accurate is our estimate of p_2 when we start at p_1 and move toward it?

Convergence Rate We place the car at various positions as shown in Figure 9. The car is stationary and we begin timing the instant the localization begins and stop whenever the particles attain some variance threshold ($\sigma \leq 0.2$) and the best scan probability exceeds some value p_0. Table 2 summarizes our results, and Figure 10 shows LiDAR scans after convergence for the non-ground truth positions. Our results indicate that the localization is robust to slight deviations in the initial pose.


(a) Ground Truth Position

(b) Far Translation Offset


(c) Translation + Rotation Offset

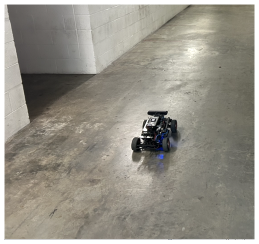
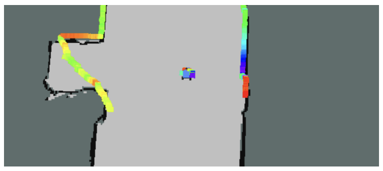
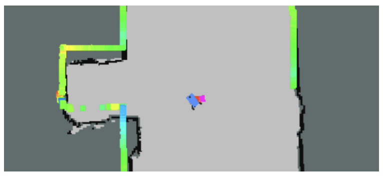

(d) Close Translation Offset

Figure 9: Positions of the car used for convergence rate testing. We start with (a) the ground truth position next to the wall indentation landmark. We vary offsetting by translation and rotation and estimate the time for particles to converge.

Table 2: Convergence Test Results

Trial	**Position** (x, y)	**Orientation (z,w)**	σ **(m)**	**Convergence Time (s)**
Ground Truth	(-33.11, 16.63)	(0.72, 0.70)	0.087	0.291
Far Translation	(-32.14, 17.09)	(0.71, 0.70)	0.100	0.441
Translation + Rotation	(-33.15, 16.83)	(0.91, 0.41)	0.043	0.318
Close Translation	(-34.00, 16.35)	(0.71, 0.70)	0.015	0.307


(a) Far Translation


(b) Translation + Rotation

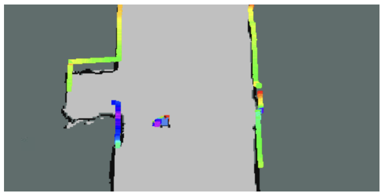

(c) Close Translation

Figure 10: Final LiDAR scans shown in simulation after the particles have fell below the variance threshold. The scans line up with the actual wall features, suggesting that the estimated pose is accurate. In (a), despite opening the door in the indentation, the car can still localize itself.

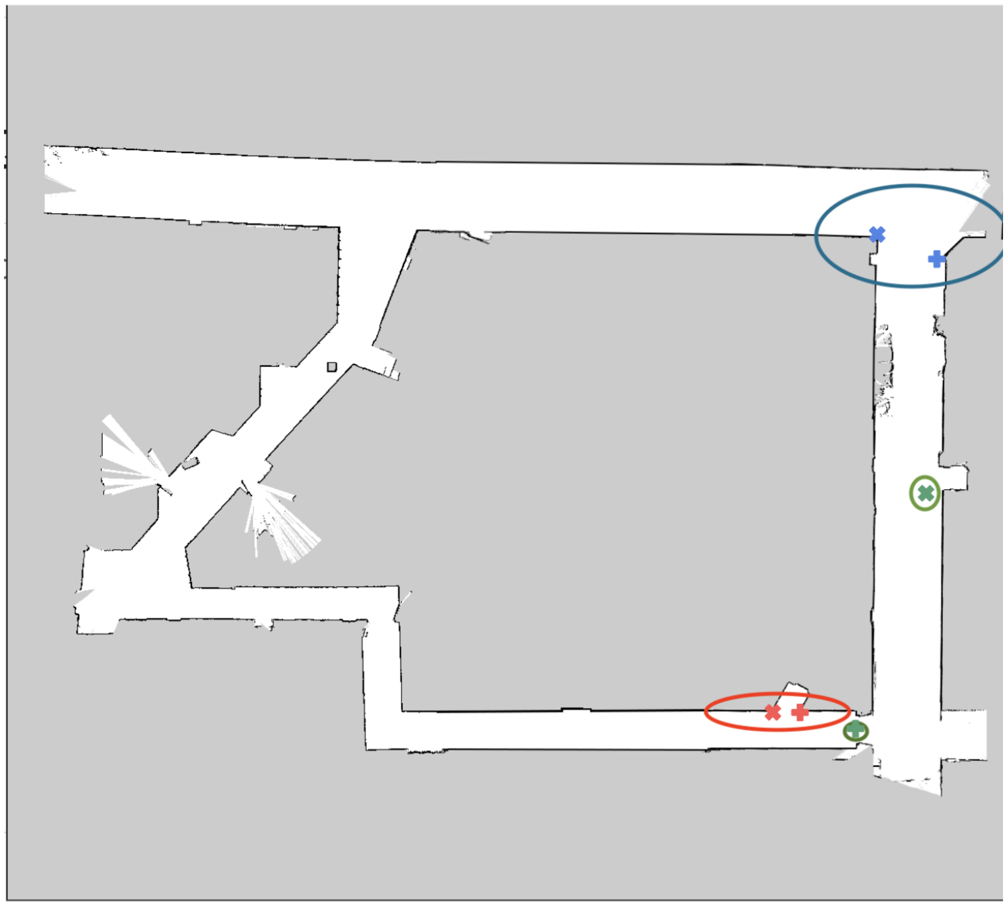


Figure 11: Map showing our cross-track testing points. The plus (+) signs indicate the starting points and the cross ($\times$) indicates the ending points.

Cross Track Error We also evaluate the localization when motion is added using a cross track error metric. We consider the three cases shown in Figure 11. Results are shown in Table 3. The ground-truth initial and final points are denoted as $p_i, i \in \{1, 2\}$. The estimated points by the localization are denoted as $\hat{p_i}$. We notice that the error $||p_2 - \hat{p_2}||_2$ is always close to or even better than the starting error $||p_1 - \hat{p_1}||_2$, suggesting that the MCL is capable of localizing itself when non-stationary. We speculate that the initial error may be due to inconsistencies between the provided map and the actual environment.

3.3 Limitations - JS

While the Monte Carlo Localization algorithm is a very powerful tool, it has a couple key limitations. First, MCL operates under the static world assumption, or the Markov assumption, which means that the algorithm only works as intended if there are no dynamic elements. This is often not the case in the real world, as most environments we want a robot to be able to navigate contain people moving around. Similarly, another limitation is that the robot must have an accurate map of the world beforehand in order to estimate its position and orientation, as we noticed in the real world tests. Finally, there is an inherent trade-off between the accuracy and the computational cost of the model, which is controlled by the number of particles. Increasing the number of particles allows the approximation to be more accurate, but requires more computational resources at each time step. On the other hand, decreasing the number of particles is optimal for computation but negatively affects the approximation.

Table 3: Cross Track Points and Errors

Trial	p_1	p_2	$\hat{p_1}$	$\hat{p_2}$	$\|\|p_1 - \hat{p_1}\|\|_2$	$\|\|p_2 - \hat{p_2}\|\|_2$
1	(-25.86, 32.84)	(-23.87, 32.82)	(-25.95, 33.06)	(-24.07, 33.03)	0.237	0.290
2	(-24.82, 34.05)	(-34.73, 18.40)	(-25.03, 33.96)	(-34.71, 18.25)	0.228	0.151
3	(-35.00, 2.76)	(-31.44, 1.92)	(-35.37, 2.62)	(-31.12, 1.75)	0.396	0.362

4 Conclusion - JS

In this lab experiment, we successfully solved the problem of robotic localization using the Monte Carlo Localization algorithm. First, we implemented the motion model, which predicts the subsequent positions of particles given odometry data, as well as the sensor model, which takes the output of the motion model and assigns probabilities to each particle. Then, we combined the motion and sensor models, making sure to precompute and memoize probability values to optimize performance and use thread-locking to prevent data conflicts. After we finished implementing MCL, we ran tests both in simulation and in the real world to evaluate performance which demonstrate the robustness of our implementation to initial pose offsets and that it is reliable when the car is in motion.

5 Lessons Learned

[**AY**] From a technical perspective, a common theme I learned was that accuracy and computational efficiency are often at odds with each other, and we need to make calculated decisions to balance them effectively. For example, with infinite compuational time, we could use Bayesian Filtering to fully model the uncertainty in the car's position. However, because the approach is not tractable, we need to resort to MCL and use a set of particles to approximate the distribution of the car's position. We encountered this again when we chose to down-sample the laser scans in order to reduce computation time, yet retain representativeness of the scan as a whole. I also enjoyed learning about convergence rate and cross track tests; I think they were interesting ways to evaluate the performance of the particle filter. Additionally, this was my first time working with thread locking, and it was great to see a real application of why it is useful and how convenient it is to implement with the locking python package.

From a CI perspective, I learned that is is important to share solutions to hardware issues with other teams and document it on Piazza. Many teams were encountering similar hardware issues, and because the course staff was often not able to help everyone, it saves everyone's time to document solutions you encounter to these problems. At one point, I (and many other teams) was running into an error in the setup.py file. After digging through the build file, I finally discovered that the sim_ws file on the car was missing, and we needed to copy it over from the docker container and make sure environment variables were sourced properly in order for the setup.py to work. Documenting this on Piazza was helpful to a lot of teams and time for everyone involved. Although frustrating, one should expect these problems to arise and share how to resolve them.

[**OM**] I think that the main technical lesson I learned in this lab was regarding balancing the needs of various environments for the particle filter. The particle filter relied heavily on the LiDAR data for noisy, feature-filled environments, so higher noise values increased accuracy as the sensor model was able to filter down improbable particles easily. However, in the lack of these features, like in a long, straight hallway, there was a greater reliance on odometry data. In this case, it was more useful to use lower noise values so that the odometry was more similar to the ground truth, as the sensor model suffered in simple environments. As we wanted the car to perform well in both cases, we had to find noise values that best balanced the needs of each environment, which took a great deal of testing.

In terms of a communication lesson, I learned that it is always useful to ask other groups what they are struggling with. I found it very common that almost every group ran into the same bugs and complications with hardware. Rather than every group having to solve these issues independently, it saves a lot of time by sharing solutions so more time can be spent on focusing on the actual lab and improving the performance of the code.

[MW] Reflecting on the recent lab, I can honestly say that it was the most challenging one so far. However, it was also one of the most rewarding experiences in terms of gaining a deeper understanding of localization. Despite the concepts being easy to understand, implementing the MCL algorithm proved to be quite challenging. It required a great deal of optimization and tuning to get everything to work seamlessly. As a team, we had to focus on effective communication and collaboration to implement the MCL algorithm successfully. We quickly realized that we needed to work together and support one another to achieve our goal. We communicated frequently, asked for help when needed, and provided assistance to those who were struggling. One of the initial difficulties we have was working on the project before and after spring break. This disrupted our momentum and made it challenging to remain focused. Looking back, I think we could have benefited more from reaching out to others, including our TA or other groups, who were also struggling with the same problem. We should also take advantage of office hours and other resources available to us. This way, we can work together more efficiently, overcome any obstacles more easily, and achieve our goals more effectively.

[JS] Technical: I learned about the importance of noise in implementing the MCL algorithm. Without noise, the MCL algorithm is prone to global localization failures, where the algorithm is unable to recover if the single most likely pose turns out to be incorrect. By introducing noise, in the form of random particles, there is an extra level of robustness against both global localization failures and kidnapping. In addition, I learned that locks can be used to prevent conflicts when updating the same thing from multiple threads. Before this lab, I didn't even know what a thread was, nevertheless how to implement thread-locking.

Communication: While there is no panacea to solve all our Real2Sim problems, I learned that having a structured approach makes debugging much more efficient. Many of the hardware problems we ran into during this lab were different from previous labs, and therefore required new technical solutions. However, while the problems themselves were unique, we were able to resolve them quicker than before. I believe this can be attributed to our more organized debugging methods. One particular strategy that proved to be helpful was explaining our reasoning/code to another teammate that wasn't as familiar with a specific module. The teammate may have a new insight, but the act of talking through the logic was helpful in itself.

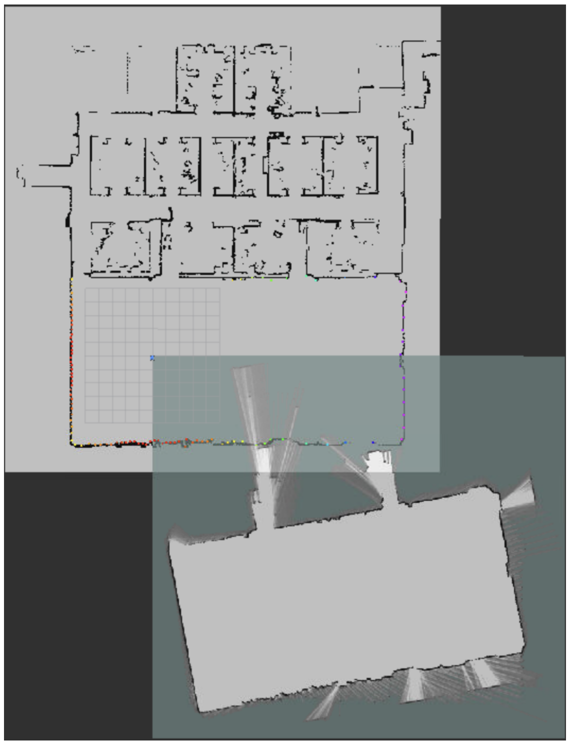
[RS] From a technical perspective, I learned about how noise can be used positively rather than always being a bane we must deal with, for without the use of noise, our poses gradually became more and more erroneous, whereas once we used noise on our next pose calculations, we no longer had that problem. I also learned how we can use a probabilistic model to represent a system where we have uncertainty and use those to make accurate estimations of said system. This was key with our method of resampling our particles list to reflect more probable poses and thereby converge towards our actual pose. From a communications perspective, I learned the value of having multiple people working together when debugging. When we tried to debug our code, we never got stuck in a rabbit-hole and always had new ideas because several brains were puzzling together over the problem. Likewise, talking with other teams also helped me generate new ideas. For example, when trying to come up with quantitative ways of measuring our localization methods, my ideas weren't translating to real-world tests. After talking with someone from another team, I was able to come up with new ideas that we were eventually able to test. These examples show the power of multiple brains to come up with new ideas and better solutions.

[SY] On the technical side, I learned the method of the Monte Carlo Localization with particle filter. Through testing in both simulation and real-world environments, we gained a deeper understanding of the roles different parameters played in the overall performance of the filter. Also, this experience emphasized the balance between accuracy and computational efficiency in algorithm implementation, which is essential for us to develop an effective algorithm in the real world. The process of adjusting parameters to suit different environments and conditions has expanded my understanding of the applicability of such algorithms in real-world scenarios.

Regarding the CI aspect, I realized the importance of synchronizing information between teams to promote a collaborative and efficient work environment. By sharing experiences, learning from other teams and experience on piazza, we were able to avoid wasting time on low-efficiency tasks, such as debugging autograder tests or addressing hardware issues. Furthermore, effective collaboration and communication within our team also played a critical role in project advancement. When faced with difficulties in assigned tasks, the assistance and support from teammates proved to be much valuable in overcoming challenges.

6 Appendix

6.1 Cartographer - JS


(a) Results of Cartographer in simulation with the map of Building 31 and laser scan data

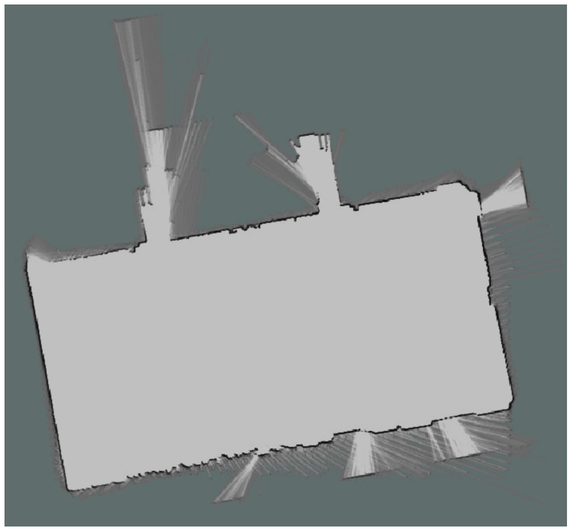

(b) Results of Cartographer in simulation

Figure 12: Map of Building 31 using Cartographer.

We were able to build a map of Building 31 using Google Cartographer in a simulation environment. Unfortunately, there was a considerable amount of noise around some of the edges due to the laser scan going through the wall.