# Lab 5 Report: Monte Carlo Localization

Team 8

Monica Chan
Cristine Chen
Alex Franks
Kyle Fu
Asa Paparo

6.4200 Robotics: Science and Systems

April 12, 2025

## 1 Introduction

*Author: Asa Paparo*
*Editor: Cristine Chen*

This lab focused on localizing the racecar within a predefined map using the Monte Carlo Localization (MCL) algorithm, a probabilistic method for localization. Localization is a fundamental problem in autonomous systems, as it allows robots to determine their precise position and orientation, which is essential for tasks like path planning and obstacle avoidance. MCL is widely used in autonomous vehicles, drones, and other mobile robots for navigating unknown or dynamic environments. We applied MCL to the racecar to estimate its pose (position and orientation) as it moved around the Stata basement. To accomplish our task, we implemented the MCL algorithm in three main components, illustrated in Fig. 1:

1. Motion Model: The motion model predicts the robot's future pose based on its previous pose and the action odometry data from the wheel encoders. This model accounts for uncertainty in movement, such as slippage or wheel inaccuracies, by introducing noise into the prediction.

2. Sensor Model: The sensor model evaluates how likely a given predicted pose is, based on the data obtained from the robot's LiDAR sensors. It compares the sensor readings to the expected readings from the map, assigning a likelihood or weight to each predicted pose. The closer the predicted pose is to the actual sensor data, the higher the weight.

3. Particle Filter: The particle filter is used to manage a set of possible poses (represented by particles). It iteratively prunes the set by resampling the particles, discarding those that are unlikely given the sensor data, and focusing on those that have a higher likelihood. Over time, this process refines the robot's estimate of its true position and orientation.

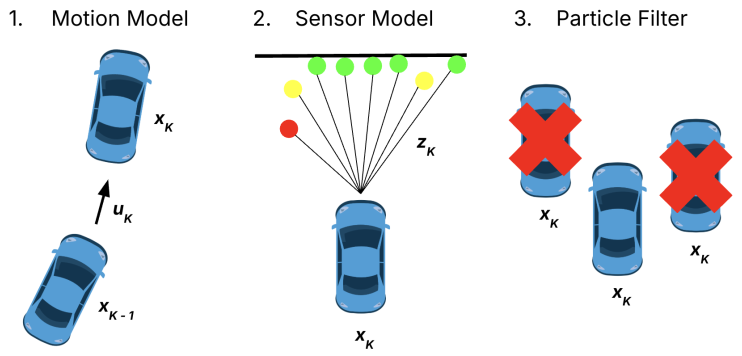

**Fig. 1:** Components of the MCL algorithm. The motion model uses odometry to predict the next robot pose, the sensor model determines the likelihood of a pose being correct, and the particle filter resamples to remove unlikely poses.

The components of the MCL algorithm were first implemented on a simulated racecar within a model of the Stata basement. The controlled environment allowed us to test and fine-tune the algorithm, using ground truth data to validate localization results. The simulation allowed us to evaluate the algorithm's performance under ideal conditions, where we could compare the predicted positions with the known true positions of the robot.

Following successful simulation-based localization, the algorithm was applied to the physical racecar. The real-world deployment introduced additional challenges, such as sensor noise, dynamic obstacles, and motion inaccuracies caused by environmental factors. These factors added complexity to the task, requiring us to adapt the algorithm to account for additional sources of uncertainty that were not present in the simulation.

This lab not only deepened our understanding of the core principles of localization but also provided hands-on experience with implementing a sophisticated algorithm like MCL. It bridged the gap between theoretical concepts and real-world application, highlighting the challenges of deploying algorithms in dynamic, real-world environments.

# 2 Technical Approach

## 2.1 Motion Model

*Author: Cristine Chen*
*Editor: Alex Franks*

In the motion model, our goal is to predict the future pose of the racecar, given its initial pose and odometry data from its movement. We designed two versions of the motion model: a deterministic model for the simulation and a non-deterministic model for the actual racecar.

- Deterministic motion model: Ground truth odometry data was used in simulation.
- Non-deterministic motion model: For the actual racecar, noise was incorporated into the model to account for noisy wheel odometry data.

To predict the future pose of the robot, we began with an $N \times 3$ array of particles, where $N$ represents the number of particles. Each particle is a vector $(x, y, \theta)$ that represents the racecar's location relative to a fixed origin on a 2D plane. In addition to the particle data, we also accepted an odometry vector as a $3 \times 1$ matrix representing the odometry data $\Delta x$, which provides information about the robot's movement. Next, we transformed both the particles and $\Delta x$ from the vector representation into $3 \times 3$ homogeneous transformation matrices. This transformation allowed us to account for both the translation (position) and rotation (orientation) of the racecar in 2D space with respect to the world frame $W$. We can model $x_{k-1}$, the previous pose of the racecar at $t = k-1$ and $x_k$, the predicted future pose of the racecar at $t = k$, as homogeneous transformation matrices defined in Equation 1 as

$$T = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \tag{1}$$

where $R$ represents the $2 \times 2$ rotation matrix and $t = \begin{bmatrix} x & y \end{bmatrix}^T$ is the translation vector. The rotation matrix $R$ is defined in Equation 2 as

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{2}$$

Using these transformation matrices, we applied Equation 3 to calculate the predicted future pose of each particle

$$T_k^W = T_{k-1}^W \cdot T_k^{k-1} \tag{3}$$

where $\Delta x$ can be modeled as the homogeneous transformation matrix $T_k^{k-1}$, representing the pose of the robot at $t = k$ with respect to its pose at $t = k-1$. The prediction is done by taking the dot product of the transformation matrices for each particle and $\Delta x$, updating their positions and orientations accordingly.

In the non-deterministic model, we added noise to the particles before performing the calculation for the predicted future pose. We first initialized a timer to track the elapsed time ($dt$) since the last odometry reading, which we used to calculate the change in position. Finding $dt$ allowed us to scale the noise based on the velocity and avoid issues caused by delayed odometry data. We calculated the velocity components (linear velocities $v_x$ and $v_y$, and angular velocity $v_\theta$) by dividing the displacement by $dt$. To determine the amount of noise to add, we defined base noise levels and scaled them according to the computed velocities. For the $x$ and $y$ directions, the noise standard deviation was calculated according to Equation 4 and 5 respectively. For the orientation $\theta$, we used Equation 6.

$$x_{\text{dev}} = (1.0 + 0.5 \cdot |v_x|) \cdot dt \tag{4}$$

$$y_{\text{dev}} = (1.0 + 0.5 \cdot |v_y|) \cdot dt \tag{5}$$

$$\theta_{\text{dev}} = (\pi/4 + 1.0 \cdot |v_\theta|) \cdot dt \tag{6}$$

These velocity-based deviations were then used as the standard deviations for generating random noise from a Gaussian normal distribution with a mean of 0. By scaling the noise based on $v_x$, $v_y$, and $v_\theta$, we ensured that faster movements introduced greater uncertainty, resulting in a more realistic and robust particle motion model.

To improve performance, especially given the large number of particles and the high frequency of updates required for real-time operation, we avoided using for-loops entirely. Instead, we leveraged NumPy's vectorized operations, which allowed us to efficiently update all particle poses in parallel. This approach not only enhanced computational speed but also ensured that the motion model could scale effectively for implementation on the actual racecar.

## 2.2 Sensor Model

*Author: Alex Franks*
*Editor: Kyle Fu*

In the sensor model, our goal is to predict the probability that a given predicted future pose is true given the sensor measurements from our LiDAR scan. To do so efficiently and accurately, we use a precomputed probabilistic model for the sensor measurements. We create a discretized 2D probability lookup table, where each entry at position $[z_k(i)][d]$ represents the probability of observing a LiDAR measurement $z_k(i)$ (measured distance) when the expected distance to an obstacle is $d$ according to the predicted future pose $x_k$ and map $m$. These measurements are in pixels (px), where 1 px $\approx$ 0.0504 m. The table is then populated by combining the probabilities of the four types of LiDAR measurement noise where:

1. $p_{\text{hit}}$ models the probability of detecting a known obstacle in the map.
2. $p_{\text{short}}$ models the probability of short-range measurements (unexpected obstacles).
3. $p_{\max}$ accounts for max-range readings (missed measurements).
4. $p_{\text{rand}}$ represents completely random noise.

The calculations for each of these probabilities are defined in Equation 7-10:

$$p_{\text{hit}}(z_k(i) \mid x_k, m) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k(i)-d)^2}{2\sigma^2}\right) & \text{if } 0 \leq z_k(i) \leq z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where probability $p_{\text{hit}}(z_k(i) \mid x_k, m)$ is modeled as a Gaussian (normal) distribution within the valid range $[0, z_{\max}]$.

$$p_{\text{short}}(z_k(i) \mid x_k, m) = \begin{cases} \frac{2}{d}\left(1 - \frac{z_k(i)}{d}\right) & \text{if } 0 \leq z_k(i) \leq d \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

where probability $p_{\text{short}}(z_k(i) \mid x_k, m)$ is modeled as a Triangular distribution within the valid range $[0, d]$.

$$p_{\max}(z_k(i) \mid x_k, m) = \begin{cases} \frac{1}{\epsilon} & \text{if } z_{\max} - \epsilon \leq z_k(i) \leq z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where probability $p_{\max}(z_k(i) \mid x_k, m)$ is modeled as a Uniform distribution within the valid range $[z_{\max} - \epsilon, z_{\max}]$ and where $\epsilon = 0.1$. When discretizing this probability, we simply set $p_{\max}(z_{\max} \mid x_k, m) = 1$.

$$p_{\text{rand}}(z_k(i) \mid x_k, m) = \begin{cases} \frac{1}{z_{\max}} & \text{if } 0 \leq z_k(i) \leq z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where probability $p_{\text{rand}}(z_k(i) \mid x_k, m)$ is modeled as a Uniform distribution within the valid range $[0, z_{\max}]$.

We also define and set key hyperparameters ($\alpha_{hit} = 0.73, \alpha_{short} = 0.1, \alpha_{max} = 0.06, \alpha_{rand} = 0.11, \sigma = 8$ px, $z_{\max} = 200$ px) that weight the contribution of each probability. These $\alpha$ parameters, which were empirically tuned with the added consideration that unexpected obstacles were common on the racecar (higher $\alpha_{\text{short}}$), control the shape of the probabilistic model and sum up to 1. Each of the aforementioned probabilities are scaled by their respective $\alpha$ values and combined into a total as shown in Equation 11.

$$p(z_k(i) \mid x_k, m) = \alpha_{\text{hit}} \cdot p_{\text{hit}} + \alpha_{\text{short}} \cdot p_{\text{short}} + \alpha_{\max} \cdot p_{\max} + \alpha_{\text{rand}} \cdot p_{\text{rand}} \tag{11}$$

The entire lookup table is normalized across columns so that for a given expected distance $d$, the probabilities of all possible measurements $z_k(i)$ sum to 1. This normalization ensures that the resulting probabilities are valid and can be directly used in a likelihood computation.

To apply our lookup table, we first subscribe to a ROS2 occupancy grid map topic. When the map is received, we convert it into a NumPy array suitable for fast access and load it into PyScanSimulator2D — a raycasting simulator. This tool simulates what the LiDAR would read from any given pose on the map. Before likelihood computation, we downsample the incoming LiDAR scan to reduce the number of beams to 100 per particle. Fewer beams mean faster computation per particle, which is essential when tracking hundreds of particles in real time. Then we simulate what the LiDAR would measure from each particle's pose using the map. These simulated distances $d$ and the real measurements $z_k(i)$ are both discretized (converted to integer pixels) and clipped to stay within bounds. Clipping ensures that we do not index outside the precomputed table due to noisy or invalid values. For each particle, we calculate the total likelihood of the LiDAR scan given the pose $x_k$ using Equation 12.

$$\mathbb{P}(z_k' \mid x_k) = \mathbb{P}\left(z_k'(1), \ldots, z_k'(n) \mid x_k\right) = \prod_{i=1}^{n} \mathbb{P}\left(z_k'(i) \mid x_k\right) \tag{12}$$

This equation calculates the product of the individual beam likelihoods, assuming the beams are conditionally independent. Each term $P(z_k'(i) \mid x_k)$ is efficiently retrieved from the lookup table. This multiplication process gives us the overall probability that the particle's pose is correct given the entire scan.

We observed that the resulting particle probability distribution tended to be very peaked. This distribution, along with the noised motion model, created instances in long hallways where no particles matched the true pose of the robot, but a far away particle happened to have a slightly more aligned LiDAR scan, resulting in the closer particle being thrown away. We attempted to spread out the distribution by increasing the noise of the sensor model ($\sigma$), but it greatly reduced the reliability of the pose estimate in the typical case. It was also infeasible to increase the number of particles above 100 due to computational cost.

To alleviate peaked distribution issues, we implemented a squashing function shown in Equation 13 on the particle probabilities.

$$\mathbb{P}(z_k \mid x_k) \propto \sqrt{\mathbb{P}(z_k' \mid x_k)} \tag{13}$$

A square root squashing function lowers the impact of each LiDAR measurement on the particle's probability without modifying the sensor's probabilistic model. One way to see this process is that it makes 2 LiDAR measurements have roughly the same impact as 1 original measurement. We normalized these particle probabilities to sum to 1.

## 2.3 Particle Filter

*Author: Kyle Fu*
*Editor: Asa Paparo*

In our particle filter model, our goal is to maintain a probabilistic estimate of the robot's pose $(x,\ y,\ \theta)$ over time by fusing odometry and LiDAR data through the recursive application of Bayes' Filter. The filter operates in two main phases: prediction and update.

When new odometry data arrives, we first perform a prediction step that models how the robot is likely to have moved according to Equation 14.

$$P(x_k \mid u_{1:k}, z_{1:k-1}) = \int \underbrace{P(x_k \mid x_{k-1}, u_k)}_{motion\ model} \cdot P(x_{k-1} \mid u_{1:k-1}, z_{1:k-1})\, dx_{k-1} \quad (14)$$

This integral reflects the application of the motion model $P(x_k \mid x_{k-1}, u_k)$, which describes how the robot transitions between states given control input $u_k$ (odometry). In practice, we approximate this integral by sampling: each particle is propagated forward using the odometry-derived linear and rotational velocities and the time delta $(dt)$, with Gaussian noise added to reflect real-world uncertainty in actuation. We computed the delta pose from the odometry message in Equations 15-17.

$$\Delta x = v_x \cdot dt \quad (15)$$

$$\Delta y = v_y \cdot dt \quad (16)$$

$$\Delta \theta = \omega_z \cdot dt \quad (17)$$

We apply this delta to each particle's pose, adding zero-mean Gaussian noise to simulate uncertainty in both translation and rotation. This probabilistic sampling from the motion model effectively spreads the particle cloud based on expected motion and models the diffusion of belief over time.

Once a new LiDAR scan $z_k$ is available, we perform the update step, refining our belief by measuring how well each particle explains the new observation as demonstrated in Equation 18:

$$P(x_k \mid u_{1:k}, z_{1:k}) = \alpha \cdot \underbrace{P(z_k \mid x_k)}_{\text{sensor model}} \cdot P(x_k \mid u_{1:k}, z_{1:k-1}) \quad (18)$$

Here $P(z_k \mid x_k)$ is the sensor model, which evaluates the likelihood of receiving the observed scan given a hypothesized pose. In our system, this process is implemented via a precomputed sensor model that compares the observed laser scan against a simulated scan at each particle's predicted pose in the map. Each particle is reweighted according to this likelihood. Particles that align well with the current scan (i.e., the environment as seen from that pose) receive higher weights, concentrating belief around more plausible locations.

Over time, repeated updates tend to concentrate weight on a few particles, risking particle deprivation (where diversity is lost). To counteract this loss, we perform resampling: we draw a new set of particles from the current weighted distribution using importance sampling with replacement. This sampling favors high-weight particles, duplicating good hypotheses while discarding unlikely ones. This step is essential to avoid the collapse of the filter and ensure continued adaptability to new information. Without it, most particles would contribute little, wasting computational resources.

When the robot receives an externally provided pose estimate, we initialize or reinitialize the particle cloud by sampling from a Gaussian distribution centered at the given pose. This process allows the system to recover from localization failure or initialize tracking in a known region of the map.

After resampling, we compute the robot's estimated pose as a weighted average over all particles:

- The position estimate is computed as the weighted mean of $(x, y)$.
- The orientation $\theta$ is computed as the circular mean using Equation 19 to handle the periodic nature of angles.

$$\theta = \text{atan2}\left(\sum_i w_i \sin(\theta_i), \sum_i w_i \cos(\theta_i)\right) \tag{19}$$

By iteratively applying the prediction and update steps grounded in the Bayes filter framework, the particle filter maintains a robust, adaptable estimate of pose even in the presence of sensor noise, uncertain motion, and ambiguous observations. While the true integrals of the Bayes filter are intractable, particle filtering provides a flexible and scalable approximation that converges to accurate beliefs given a sufficient number of particles. In our implementation, we used a cloud of 100 particles.

## 3 Experimental Evaluation

### 3.1 Simulation Results

*Author: Monica Chan*
*Editor: Asa Paparo*

To validate our particle filter implementation, we used a combination of Rviz visualization and simulation in the Stata basement. Our primary goal was to assess the robustness and accuracy of the localization system under realistic motion and observation conditions.

For motion inputs, we employed the wall follower algorithm developed in Lab 3 to simulate realistic navigation along the map's hallways which ensured that our motion model was exercised over meaningful trajectories. The localization system was tested in a known map with predefined long corridors, a setting that is particularly sensitive to motion model tuning due to the low information gain from LiDAR in feature-sparse environments.

We systematically varied the motion model parameters, particularly the standard deviation of Gaussian noise injected into particle propagation. This approach helped us explore the trade-off between filter robustness and accuracy. Specifically, we tested standard deviations ranging from lower values like 0.1 to higher values like 0.3 for $x$ and $y$. Fig. 2 summarizes the particle filter's performance under these configurations.

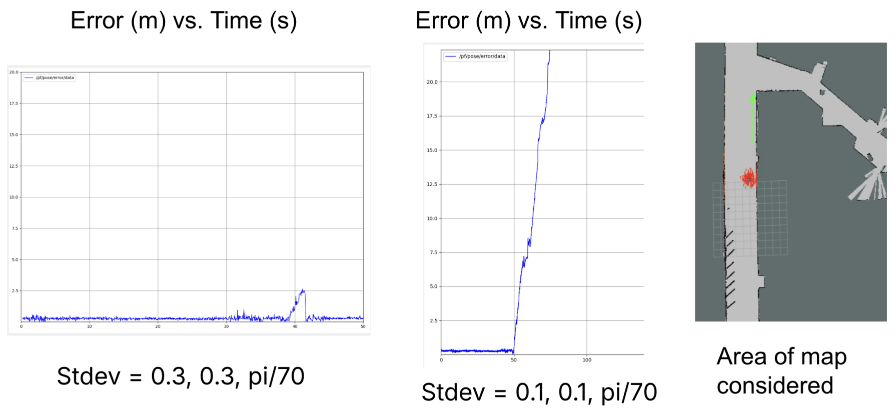

**Fig. 2:** Results of varying the amount of noise added to the motion model when traversing a long hallway. To simplify comparions in this experiment, we used Gaussian noise with the standard deviations shown above.

We found that the amount of noise added during the prediction step had a significant effect on filter stability. Adding too little noise caused the particle filter to diverge, particularly in long, straight hallways where the lack of distinctive LiDAR features meant that small odometry errors compounded over time without being corrected by observations. In these cases, the particle cloud collapsed into an inaccurate pose estimate and failed to recover.

Additionally, in real-world deployments, odometry is inherently noisy due to wheel slip and sensor drift. Since simulation provides idealized odometry, we conducted a controlled test to inject noise directly into the odometry stream. We sampled perturbations from a Gaussian distribution with standard deviations $(0.1, 0.1, \pi/6)$ in the (x, y, $\theta$) dimensions. Fig. 3 shows the resulting localization performance.

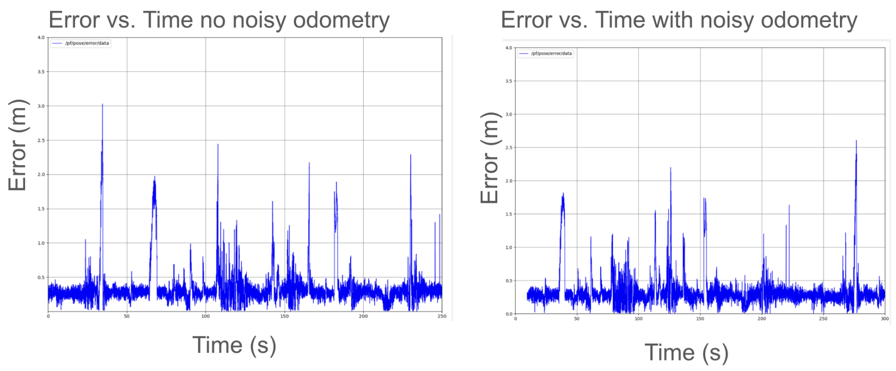

**Fig. 3:** Error over time with and without noisy odometry in simulation. Error was calculated as the distance between the calculated pose and the actual pose

Despite the added odometry noise, the particle filter was able to maintain a consistent pose estimate, demonstrating its ability to recover from noisy motion input through observation updates. This result increases our confidence in the filter's applicability to real-world conditions where sensor drift and motion uncertainty are unavoidable.

In preparation for Lab 6, we aim to refine the noise tuning further. Preliminary observations suggest that adding slightly more noise along the x-axis—the robot's primary direction of travel—could better model the variance in forward motion without compromising convergence.

## 3.2 Robot Results

*Author: Monica Chan*
*Editor: Cristine Chen*

To evaluate the effectiveness of our localization algorithm, we conducted both qualitative and quantitative tests in the Stata basement. These tests were designed to assess how accurately and consistently our particle filter tracks the robot's pose over time during real navigation scenarios.

We first conducted a full-loop test in the Stata basement, with the robot navigating the entire lap using a manual wall-following controller. A video of the run, with overlaid RViz visualizations showing the inferred particle cloud and estimated trajectory, is available here: link to video.

This demonstration shows that the particle filter remains robust over extended operation. Notably, the estimated trajectory closely tracks the robot's true motion even during sharp turns and slalom-like maneuvers in narrow corridors. Fig. 4 shows a snapshot of the RViz setup used during this test.

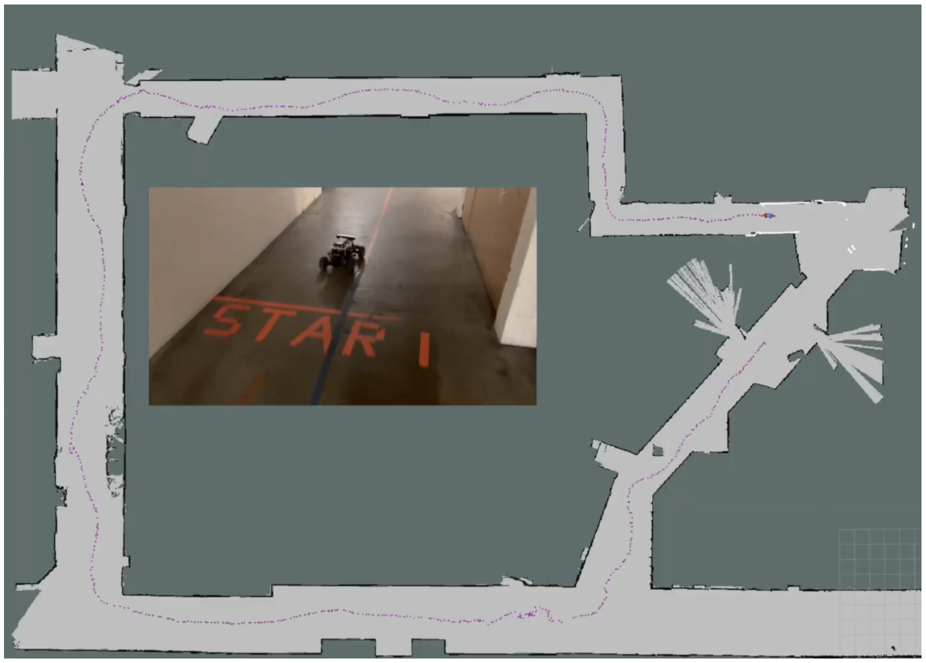

**Fig. 4:** Visualization setup for traversing the lap around Stata.

To obtain a more objective measure of accuracy, we ran a controlled experiment where the robot followed an orange tape line along a hallway. This line served as a proxy for ground-truth position. We then computed the cross-track error, defined as the Euclidean distance between the robot's estimated position (using the particle filter) and the nearest point on the reference line.

Our experimental setup is shown in Fig. 5. The orange line represents the physical tape, the green line is the ground-truth reference for the tape in the map used for error computation, and the purple trajectory is the robot's estimated path. The robot was manually driven during this test.

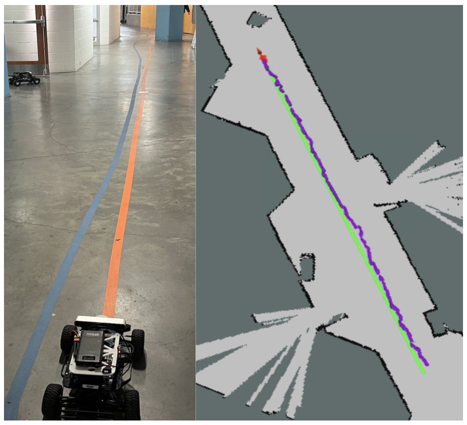

**Fig. 5:** Experiment setup for measuring cross-track error. The robot follows the orange line on the ground in the image on the left, which is visualized as the green line in the image on the right. The purple line on the right represents the robot's calculated positions over time.

The results of the cross-track error test are shown in Fig. 6.

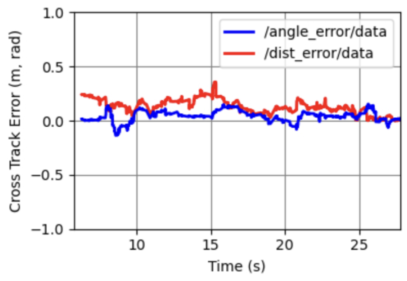

**Fig. 6:** Cross-track error while following the orange line. The blue line represents angular error, and the red line represents distance error. The distance error appears to be centered around 0.15 meters, which we attribute to the offset between the LiDAR and the robot's base link.

The red line plots the position error, while the blue line shows the angular error (difference between estimated and actual orientation). Both the mean position error and the angular error remained within a reasonable bound throughout the run. We attribute the consistent offset in position error to a known misalignment between the LiDAR and the robot's base link frame. Other sources of error may include:

- Imperfect human driving while following the tape
- Inaccurate hand-annotation of the reference line in the map
- Approximation limitations inherent in the particle filter model

Despite these factors, the error remained bounded and consistent, suggesting that our system is resilient to small observation and motion inaccuracies.

## 4 Conclusion

*Author: Cristine Chen*
*Editor: Monica Chan*

In Lab 5, we implemented a Monte Carlo Localization (MCL) system using a particle filter framework to estimate the pose of a racecar robot in a known environment. Our approach combines a motion model driven by odometry with a sensor model based on LiDAR observations, iteratively applying the prediction and update steps of the Bayes filter to maintain a probabilistic belief over the robot's pose. Through simulation testing and both qualitative and quantitative evaluations, we demonstrated that our particle filter implementation is capable of robustly tracking the robot's position over extended runs.

Our qualitative results, supported by RViz visualizations, showed strong consistency between the robot's true path and its inferred trajectory, while our cross-track error analysis confirmed that the estimated pose stayed within the ground truth reference. Importantly, we found that tuning the noise parameters in the motion model was critical to achieving a balance between accuracy and robustness. These findings validate the effectiveness of our design choices under realistic noise and motion uncertainty.

However, several areas remain open for further investigation and refinement. First, our current sensor model uses precomputed likelihoods based on parameters given in the lab handout. We plan to evaluate the sensitivity of localization performance to these parameters, particularly the alpha values and the Gaussian model used in $p_{\text{hit}}$, with the goal of achieving more principled tuning or even learning these parameters from data. Second, we used a fixed number of particles (100) in our experiments; future work should explore the trade-off between computational cost and localization accuracy as a function of particle count. Finally, our motion model assumes symmetric noise in the $x$ and $y$ directions, but the racecar's movement is primarily forward-dominated. Introducing anisotropic noise or even learning motion model parameters from actual trajectory data may further improve realism and performance. By building on this foundation, we aim to create a more adaptive, data-driven localization system that generalizes well across environments and remains resilient to real-world uncertainties.

## 5 Lessons Learned

### 5.1 Monica Chan

#### 5.1.1 Technical

I learned how to publish coordinate transforms in ROS2, which significantly improved our ability to visualize the racecar's position during testing. By broadcasting these transforms—specifically between the map, odometry, and base_link frames—we could accurately track the car's location and orientation relative to the environment in real-time. This visualization made it much easier to debug and verify that our localization and motion models were working correctly, especially when preparing for the actual racecar deployment. The visual feedback in RViz helped us identify any discrepancies between expected and actual behavior, which was crucial for ensuring reliable performance.

#### 5.1.2 CI

I learned that setting specific goals for each work session and reevaluating them frequently can lead to much more productive and focused work, rather than diving in without a clear direction. By defining concrete, achievable objectives beforehand, we can create a roadmap for what we want to accomplish, which helps maintain motivation and measure progress. Regularly revisiting and adjusting those goals also allows us to respond to unexpected challenges or new insights, keeping our work aligned with our larger project timeline and ensuring that each session contributes meaningfully toward our end goal.

### 5.2 Cristine Chen

#### 5.2.1 Technical

I learned the importance of optimizing code performance, particularly in the context of implementing the motion model. Initially, I used a basic for-loop to compute the predicted pose of each particle, apply noise to the odometry, and then append the updated particles to a dynamically growing array. While this approach was straightforward, my teammate pointed out that it would become inefficient as the number of particles increased, especially given the computational demands of real-time localization. Taking this feedback into account, I refactored the code to leverage NumPy's vectorized operations, which allowed for more efficient batch processing of particle updates. This change significantly improved the performance of the motion model, making it better suited for large-scale particle filtering and real-time applications.

#### 5.2.2 CI

I learned the value of setting a clear agenda for meeting times, including outlining what tasks should be completed before the meeting, what we aim to accomplish during the meeting, and any follow-up actions that need to be taken afterward. Establishing this structure helped our team stay focused and make the most of our limited collaboration time. It also ensured that everyone came prepared, which made discussions more productive and helped us move forward efficiently with our project goals. This practice fostered better communication, clearer expectations, and a stronger sense of accountability among team members.

## 5.3 Alex Franks

### 5.3.1 Technical

I learned how to work with ROS2 launch files, which are essential for managing and automating the startup of multiple nodes and configurations in a robotics system. By defining components such as nodes, parameters, and topic remappings within a launch file, I was able to streamline the process of setting up our localization pipeline. This work not only saved time during testing but also made our setup more reproducible and easier to share with teammates. Understanding how to structure and customize launch files gave me greater control over how different parts of the system interact and run together, which is especially valuable in more complex robotics applications.

### 5.3.2 CI

I learned how to effectively integrate into a new team, which involved actively listening, being open to feedback, and quickly adapting to the team's workflow and communication style. At first, it was important for me to understand the existing dynamics and technical foundations the team had already established. By asking questions, contributing where I could, and showing a willingness to learn, I gradually built trust and became a more confident and collaborative team member. This experience taught me that successful integration isn't just about technical skills—it's also about communication, empathy, and being proactive in finding ways to support the team's goals.

## 5.4 Kyle Fu

### 5.4.1 Technical

I learned how to efficiently tune parameters both in simulation and on the actual racecar, especially given the limited time we had for testing and experimentation. These constraints required a strategic approach—carefully selecting which parameters to adjust, using structured experiments, and relying on simulation data to narrow down viable options before transitioning to real-world tests. In addition, I gained hands-on experience profiling Python code to identify performance bottlenecks, and I practiced integrating optimized modules written in Cython. This change allowed us to maintain the flexibility of Python while benefiting from the speed improvements of compiled code, which was crucial for meeting real-time constraints in our localization system.

#### 5.4.2 CI

I learned how to effectively welcome and onboard a new team member, which involved providing guidance on team processes, tools, and expectations, and ensuring they felt supported and integrated into the group. Additionally, I'm practicing advocating for task splits that everyone is happy with, while being mindful of balancing individual workloads and strengths. This process requires open communication and a thoughtful approach to ensure that tasks are distributed fairly, that team members are set up for success, and that the overall workload doesn't become overwhelming. It's a valuable skill for fostering collaboration and maintaining team efficiency.

### 5.5 Asa Paparo

#### 5.5.1 Technical

I learned how to write clean and efficient code for running the sensor model and motion model threads concurrently, which was crucial for maintaining real-time performance and accuracy in our localization system. To ensure the threads could operate without interfering with each other, I implemented locks to prevent race conditions, ensuring that shared resources were accessed in a thread-safe manner. This feature not only improved the stability of the system but also helped maintain the integrity of data being processed in both threads. By applying these principles, I was able to optimize the system's concurrency and make it more robust for real-time applications.

#### 5.5.2 CI

I learned how important clear communication is, especially when expressing my availability to the team. By proactively letting them know when I was unavailable, my team was able to adjust their schedules accordingly, ensuring that our workflow remained smooth and efficient. This practice helped prevent delays, avoided misunderstandings, and allowed everyone to plan their tasks better. It also reinforced the idea that transparent communication fosters a more collaborative and organized environment, where everyone's time and contributions are respected.