Lab 5 Report: Monte Carlo Localization using LiDAR and Odometry Data

Team 10

Cynthia Cao
Trey Gurga
Grace Jiang
Toya Takahashi
Jonathan Zhang

RSS

April 10, 2024

Contents

1 Introduction

(Authors: Jonathan Zhang)

Localization is an essential tool when working with any autonomous systems, especially those tasked with executing precise maneuvers in uncertain environments. From robots that deliver food to customers in restaurants, to those exploring new planets, each needs to have an understanding of its operating environment before it can make any decisions. This is especially true for our robot as well, as many of the actions our robot needs to implement for future labs relies on a strong conceptual understanding of where it is relative to the map. In this lab, our group worked to implement localization on our robot using the Monte Carlo Localization (MCL) Algorithm.

Our implementation of the MCL Algorithm can be thought of as split between three main modules, the Motion Model, the Sensor Model, and the Particle Filter. The Motion Model takes an input of particles and odometry data, and updates the particle's locations after a pose transformation with the odometry. The Sensor Model takes an input of particles, sensor data, and the map, and returns a set of probabilities representing the likelihood of of each particle being the same location as the robot. Finally, the Particle Filter brings everything together, initializing the particles and utilizing the motion and Sensor Model in a continuous feedback loop (see Figure 1) to get continuous live updates of the robots location.

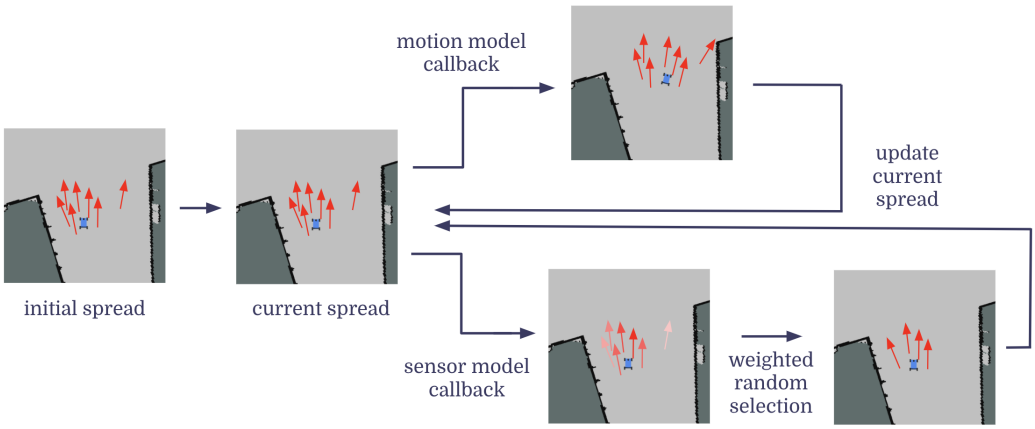


Figure 1: The Particle Filter initializes the spread of particles which continuously get updated from the Motion Model and Sensor Model callbacks

In the rest of this paper, we go over our technical approach of the localization problem and an in depth analysis of our experimental results, both in simulation and the real world, and conclude with a summary and reflection of our work.

2 Technical Approach

(Authors: Trey Gurga, Cynthia Cao, Grace Jiang)

The Particle Filter is a sequential Monte Carlo Localization method used for estimating the state of a system where the true state cannot be measured directly. It is particularly useful in robotics for

tasks such as localization, where a robot must infer its position in a given map using indirect measurements. The algorithm involves a set of stages that collectively enable a robot to estimate its pose with a certain level of confidence, despite uncertainties and noise in sensor readings and movement commands.

The process begins with an initial set of particles estimating the robot's pose. Each particle carries a weight indicating the likelihood of that particle representing the actual pose of the robot. As the robot moves and receives new sensor data, these particles are updated and resampled in a way that progressively leads to a more accurate estimate of the robot's position and orientation.

This section elaborates on the key components of the Particle Filter algorithm as applied to robotic localization: the Motion Model, which predicts the next state of the particles based on the robot's movements; the Sensor Model, which adjusts the particles' weights based on how well their predicted measurements match with actual sensor data; the Resampling step, which selects a new set of particles from the current distribution based on their weights; and the Publishing step, where the estimated pose is derived and visualized on the map. Additionally, considerations for implementing this algorithm in real-world applications are discussed, showcasing the adaptability and robustness of Particle Filters in varied and unpredictable environments.

2.1 Motion Model

(Authors: Trey Gurga)

In order to estimate the robot's pose in the given map, odometry must be combined with LiDAR data to identify the location with high probability. The Motion Model implements the odometry integration by taking the particles' poses, adding the odometry at each timestep, and outputting the updated particle poses. If a particle was initialized at the robot's exact pose and updates with exact odometry at each time step, the particle would deterministically track the robot's location as it traversed the map; this is referred to as dead reckoning. However, due to discrete odometry data and linear approximations between sampling, many particles must be used and noise must be added so that the pose can still be approximated closely. Because different noise is added to the odometry for each particle, the poses will have slightly different positions and orientations at each time step and over time will drift apart from each other. With a sufficient number of particles and tuned Gaussian noise parameters, there is high likelihood that some particles will closely align with the ground truth position and rotation. Each particle is assigned a probability of being the correct pose utilizing the Sensor Model as described in (2.2).

2.1.1 Mathematical Formulation

The propagated particle's pose utilizing odometry is laid out in (1), where $\boldsymbol{x}_k$ represents the pose at time k, $\boldsymbol{u}_k$ represents the given odometry, and $\bar{\epsilon}_u$ represents the noise parameter in normal distribution $\mathcal{N}(0, \sum_u)$.

$$\boldsymbol{x}_k = f(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k, \bar{\epsilon}_u) \tag{1}$$

The pose $\boldsymbol{x}_k$ is a 3-dimensional array $[x_k, y_k, \theta_k]$ that represents the position and orientation in the map frame. The odometry $\boldsymbol{u}_k$ is a 3-dimensional array $[\Delta_x, \Delta_y, \Delta_\theta]$ applied to the robot in the body frame at time step k. In order to apply this odometry to particle at $\boldsymbol{x}_{k-1}$, we must transform the pose from the map frame to the body frame using a transformation matrix T_{k-1}(2). We then add Gaussian noise to the odometry, yielding $\boldsymbol{u}'_k$ (3). Last, we multiply T_{k-1} by the noisy odometry transformation matrix T_Δ to obtain T_k. This process is laid out in (4). From T_k we can calculate $\boldsymbol{x}_k = [x_k, y_k, \theta_k]$ shown in (5).

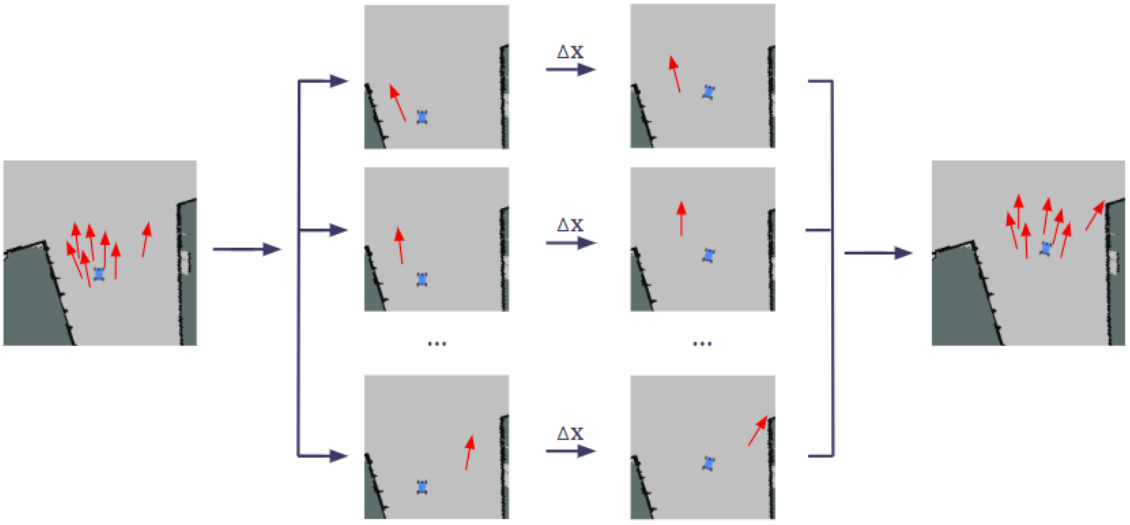


Figure 2: Visualization of Motion Model applied to particles. Noisy odometry is added to each pose at every time step and the particles disperse over time.

1. The odometry reading is perturbed by Gaussian noise to account for uncertainty, resulting in a noisy odometry $(\Delta x', \Delta y', \Delta\theta')$.

2. The initial pose of the particle is converted to a transformation matrix, T_{k-1}:

$$T_{k-1} = \begin{bmatrix} \cos(\theta_{k-1}) & -\sin(\theta_{k-1}) & x_{k-1} \\ \sin(\theta_{k-1}) & \cos(\theta_{k-1}) & y_{k-1} \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

3. A transformation matrix for the odometry, T_Δ, is constructed:

$$T_\Delta = \begin{bmatrix} \cos(\Delta\theta') & -\sin(\Delta\theta') & \Delta x' \\ \sin(\Delta\theta') & \cos(\Delta\theta') & \Delta y' \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

4. The new pose transformation matrix, T_k, is computed by multiplying the particle's transformation matrix with the odometry transformation matrix:

$$T_k = T_{k-1} \cdot T_\Delta \tag{4}$$

5. $\boldsymbol{x}_k$ is computed by extracting x, y, θ values from the transformation matrix T_k.

$$[x_k, y_k, \theta_k] = \begin{bmatrix} T_k[0,2] & T_k[1,2] & atan2(T_k[0,1], T_k[0,0]) \end{bmatrix} \tag{5}$$

2.2 Sensor Model

(Authors: Grace Jiang, Cynthia Cao)

Given particle positions generated by the Motion Model, we must determine which particles actually correspond to the current location of the robot. We use the Sensor Model to prune particle positions that are inconsistent with a laser scan reading on the map from the point of view of that particle.

The final goal, achieved using the Particle Filter, is for particles to be pruned following Sensor Model calculations as the Motion Model tries to spread them out.

To determine which particle positions are inconsistent with the LiDAR data, we calculate the probabilities that the current LiDAR scan reading is observed from each of the given particle locations, following the specifications in the lab reading. We examine those calculations as follows.

2.2.1 Using Particle Locations as Hypothesis Positions

Let z_k be the sensor reading from a hypothesis position $h_k = (x_k, y_k, \theta_k)$ in a known, static map at time k. The sensor reading is composed of n indices $z_k^1, ..., z_k^n$, each of which correspond to the LiDAR range reading at a scan angle. In simulation, we use 100 range readings, which is the whole LiDAR range vector produced in simulation. In real life, we downsample the LiDAR to increase computational efficiency.

Using the given ray-tracing code, we simulate a laser scan reading, d_k, for h_k. The ray-tracer takes in as a parameter the number of range measurements it should output in the simulated scan. Similarly to z_k, d_k is composed of n indices $d_k^1, ..., d_k^n$, where each of d_k^i correspond to the same angle at which the range reading z_k^i is taken in z_k. Through the method specified in Section 2.2.2 below, we compare the simulated range measurements, d_k, to the observed range measurements, z_k, to evaluate the probability that the robot is at the location h_k.

2.2.2 Calculating Probabilities of Hypothesis Positions

To calculate the probability that the LiDAR sensor data z_k was measured as the simulated scan d_k, we take the product of the probabilities at each index of the range measurements, or

$$\prod_{i=1}^{n} p(z_k^i | d_k^i) \tag{6}$$

Given hypothesis position h_k, which expects to see scan d_k^i and observation z_k^i for i in range $1, ..., n$, we calculate the total probability $p(z_k^i | d_k^i)$ using four separate probability components:

1. Probability $p_{hit}(z_k^i | d_k^i)$ describes the probability of detecting a known obstacle in the map using the following equation:
$$p_{hit}(z_k^i | d_k^i) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp - \frac{(z_k^i - d_k^i)}{2\sigma^2} \tag{7}$$
where σ equals 8.0.

2. Probability $p_{short}(z_k^i | d_k^i)$ represents the likelihood of a short measurement in the case of internal LiDAR reflections, hitting the robot itself, and unknown obstacles.
$$p_{short}(z_k^i | d_k^i) = \frac{2}{d_k^i}(1 - \frac{z_k^i}{d_k^i}) \text{ if } z_k^i \leq d_k^i \wedge d_k^i \neq \tag{8}$$

3. Probability $p_{max}(z_k^i | d_k^i)$ describes the likelihood of a missed measurement
$$p_{max}(z_k^i | d_k^i) = 1 \text{ if } z_k^i = z_{max} \tag{9}$$

4. Probability $p_{random}(z_k^i | d_k^i)$ reflects the probability of a completely random measurement using:
$$p_{rand}(z_k^i | d_k^i) = \frac{1}{z_{max}} \tag{10}$$
where $z_{max} = 200$.

Given the previous four probabilities, we then find the total probability that the robot expects d_k^i and sees z_k^i with

$$p(z_k^i|d_k^i) = \begin{bmatrix} \alpha_{hit} \\ \alpha_{short} \\ \alpha_{max} \\ \alpha_{rand} \end{bmatrix}^T \begin{bmatrix} p_{hit}(z_k^i|d_k^i) \\ p_{short}(z_k^i|d_k^i) \\ p_{max}(z_k^i|d_k^i) \\ p_{rand}(z_k^i|d_k^i) \end{bmatrix} \tag{11}$$

where $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{rand} = 0.07$, $\alpha_{rand} = 0.12$. These values were originally pulled from the lab reading; through testing in simulation and real-life, we find that these parameter values allow our robot to localize accurately, so they remain unchanged.

2.2.3 Implementation Details

As instructed in the lab reading, to save on computation time complexity, we precompute a lookup table to obtain the values $p(z_k^i|d_k^i)$. To look up each $p(z_k^i|d_k^i)$ in a table, we first must discretize LiDAR range measurements into integers. By dividing each range measurement by `map_resolution * LiDAR_scale_to_map_scale` and clipping distances < 0 and > 200 as instructed in the lab, we discretize the continuous measurements z_k^i, d_k^i to a discrete integer pixel $\in [0, 200]$. We precompute the table following the method presented in Algorithm (1).

Algorithm 1 Pre-computed Table Algorithm

```
Initialize p_hit, p_short, p_max as array of zeros
Set p_rand equal to array of 1/z_max
for z_k^i in range [0, 201] do
    for d_k^i in range [0, 201] do
        Calculate p_hit(z_k^i|d_k^i)
        if d_k^i ≥ 0 and z_k^i ≤ d_k^i then
            Calculate p_short(z_k^i|d_k^i)
        end if
        if z_k^i ≤ z_max then
            Set p_max(z_k^i|d_k^i) to 1
        end if
    end for
end for
Normalize p_hit along each column (corresponding to d_k^i
Take weighted sum of above arrays to find table containing all p(z_k^i|d_k^i) values
Normalize p(z_k^i|d_k^i) table along each column (corresponding to d_k^i
```

Our final lookup table, T_{sensor} has dimensions of 201 by 201 and is formatted in the following manner, where $T[a][b]$ represents the probability $p(z_k^i = a|d_k^i = b)$:

$$T_{sensor} = \begin{bmatrix} p(z_k^i = 0|d_k^i = 0) & p(z_k^i = 0|d_k^i = 1) & ... & p(z_k^i = 0|d_k^i = 200) \\ p(z_k^i = 1|d_k^i = 0) & p(z_k^i = 1|d_k^i = 1) & ... & p(z_k^i = 1|d_k^i = 200) \\ \vdots & \vdots & & \vdots \\ p(z_k^i = 200|d_k^i = 0) & p(z_k^i = 100|d_k^i = 1) & ... & p(z_k^i = 200|d_k^i = 200) \end{bmatrix}. \tag{12}$$

For each current particle, we compute its likelihood as a hypothesis position through lookups in T_{sensor}. The particles are then resampled following our methodology in the Particle Filter as described in Section 2.3. In simulation, we track 200 particles currently, but code has computational efficiency to increase this to approximately 1000 if needed.

2.3 Particle Filter

(Authors: Trey Gurga)

The Particle Filter consists of the following main steps: initialization of particles, prediction of pose, measurement update, and resampling. The process is described in Algorithm (2) and each step is explained in the sections below.

Algorithm 2 Particle Filter Algorithm

Initialize particle set $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ where each p_i is a particle representing a possible state
Initialize weights $w_i = \frac{1}{N}$ for each particle p_i
while true **do**
  if new odometry data received **then**
    Apply Motion Model to each particle based on odometry
    $p_i \leftarrow \text{MotionModel}(p_i, \text{odometry data})$
  end if
  if new laser scan data received **then**
    Update weights for each particle based on Sensor Model
    $w_i \leftarrow \text{SensorModel}(p_i, \text{laser scan data})$
    Normalize weights w_i
    Resample particles based on updated weights
    $\mathcal{P} \leftarrow \text{Resample}(\mathcal{P}, w_i)$
  end if
  if new initial pose data received **then**
    Initialize $\mathcal{P}$ around the given pose with some variance
    Reset weights w_i
  end if
  Compute estimated robot pose as weighted mean of particles
  Publish estimated pose and particles for visualization
  Compute and publish error between estimated pose and actual odometry
end while

2.3.1 Initialization

To initialize the particles an initial pose, or "guess", is provided by the user. Given the number of particles and the standard deviation for each x, y, θ, this function creates random poses following a Gaussian normal distribution around the initial pose. This step is crucial in the Particle Filter since in the real world the initial guess will almost always be incorrect, so generating particles randomly around the initial guess allows the Sensor Model to refine this guess. Additionally, since each initiated particle also receives noisy odometry by the Motion Model, a robot at rest can still localize itself if the initial guess is incorrect as particles that move toward the direction of ground truth will be assigned higher probabilities by the Sensor Model and resampling will occur. This process is covered in (2.3.4). The weights of the initial particles are initialized to all be equal.

2.3.2 Prediction

Each particle's pose is updated based on the Motion Model when new odometry is provided. The odometry provides the simultaneous linear velocity in the x and y directions and angular velocity corresponding to θ. To convert this odometry from velocity to $(\Delta x, \Delta y, \Delta\theta)$, odometry message is multiplied by the time between odometry messages Δt. The current particles' poses and the odometry message are passed through the Motion Model and the particles locations update with Gaussian noise added. The particles' poses are published.

2.3.3 Measurement Update

When the Sensor Model receives a new LaserScan message, it updates the probabilities associated with each particle pose. This is done by comparing the ray-tracing generated for each particle with the LiDAR scan, and computing updated probabilities for each particle. Each particle's weight is updated to reflect the likelihood of it being the robot's true position. Higher weights are assigned to particles whose predicted measurements are closer to the actual measurements. This likelihood is computed using a probabilistic model explained in (2.2).

2.3.4 Resampling

The resampling step aims to focus the computational resources on the most promising particles. Utilizing *numpy.random.choice*(), particles with higher weights are more likely to be sampled, while those with lower weights may be discarded. The number of particles does not change; instead, the particles are re-sampled around the highest probabilities, leading to convergence with noise. This process is crucial as it allows the Particle Filter to recover from initial poor guesses and converge towards the true pose of the robot.

2.3.5 Publishing

The Particle Filter estimates the robot's pose by calculating the weighted mean of all particles for x, y, and uses a circular mean for θ to properly account for the angular wraparound. The mean pose is used to update the transform between the map frame and the robot's base frame to provide estimate of the robot's position in the map.

2.4 Real-World Implementation

(Authors: Cynthia Cao)

The same Particle Filter algorithm as described above is run onboard the robot, with a few minor changes. The raw LiDAR reading is composed of 1081 beams which are down-sampled to $n = 99$ to optimize performance, as many beams are redundant. Furthermore, we use a lower noise level with standard deviations of $[0.05, 0.05, \frac{\pi}{24}]$, as there was already increased noise present in the real-world measurements. Finally, because our robot uses a flipped coordinate system, we negate the signs in our odometry.

3 Experimental Evaluation

(Authors: Toya Takahashi, Cynthia Cao)

We evaluate the performance of our Monte Carlo Localization algorithm both in simulation and in real-life on the robot. The procedures and results are detailed in the sections below.

3.1 Testing Procedure in Simulation

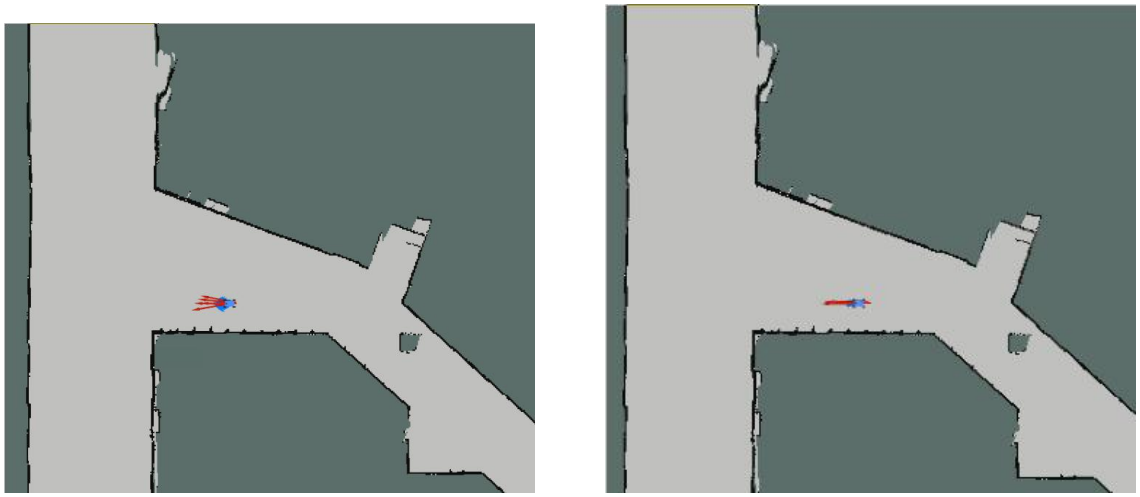
In order to test our Monte Carlo Localization algorithm in simulation, we simultaneously ran our Particle Filter with our wall follower. In RVIZ, we visualized our particles and estimated odometry using ROS2, PoseArray, and Odometry messages respectively. For a duration of 10 seconds after selecting the initial pose estimate, we collected x, y, and θ errors by calculating the difference between the ground truth and our estimated pose.

3.2 Performance Evaluation in Simulation

3.2.1 Error Evaluation

As explained in the Motion Model section, while dead reckoning can be an effective method of localization for a short period of time, accurate localization for an extended period of time requires carefully tuned noise parameters $\sum_u = [\sigma_x, \sigma_y, \sigma_\theta]$ where $\sum_u$ is the vector of standard deviations of the Gaussian noise added to the odometry. A larger standard deviation for the Gaussian noise leads to a larger spread of particles, leading to a greater probability of finding particles that closely align with the ground truth position and rotation. However, this is at the cost of greater noise in the estimated odometry: with more particles, the weighted pose average is subject to more jitter.

Figure 3 shows a visual comparison between two of the noise parameters we tested in simulation. We initially chose the noise parameter $\sum_u = [0.05, 0.05, \pi/12]$ which accurately localized the x, y position but caused noticeable jitter in the estimated orientation.

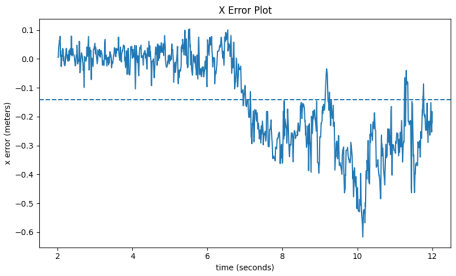


(a) $\sum_u = (0.05, 0.05, \pi/12)$ (b) $\sum_u = (0.05, 0.05, \pi/24)$

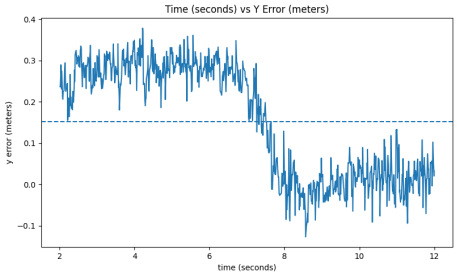
Figure 3: **Setting $\sum_u = (0.05, 0.05, \pi/24)$ lowers the variation in the estimated orientation of the robot, leading to a more accurate localization.** We found that these standard deviation values are a good compromise between drift rejection and jitter of the estimated odometry.

As shown in figure 4, on average, when making a left turn, the x position deviated from the ground truth by -0.14m, the y position by 0.15m, and the θ angle by -0.01rad. However, the estimated x and y position had a maximum deviation from ground truth of 0.61m and 0.37m respectively, while the θ angle had a maximum deviation of 0.68rad or 39°.

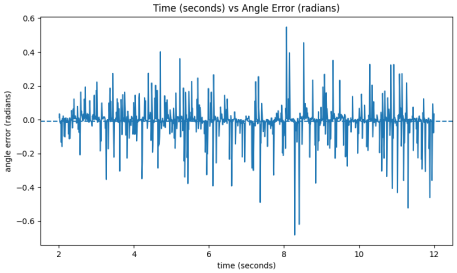
To reduce our θ error, we halved σ_θ for a noise parameter of $\sum_u = [0.05, 0.05, \pi/24]$. As shown in figures 3, 4, and 5, the difference was noticeable both visually and numerically. While the maximum deviation from ground truth of the x and y positions remained at 0.63m and 0.37m respectively, the maximum θ error reduced to 0.29rad or 16.6°. After experimenting with various values, we decided that this particular noise parameter strikes a good balance between rejecting odometry drift and having a noisy pose estimate.



(a) x error

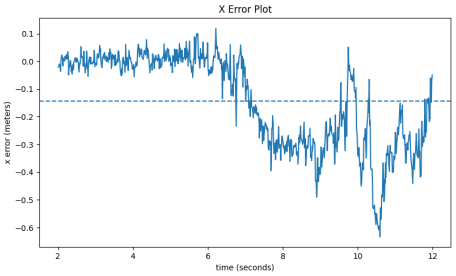


(b) y error

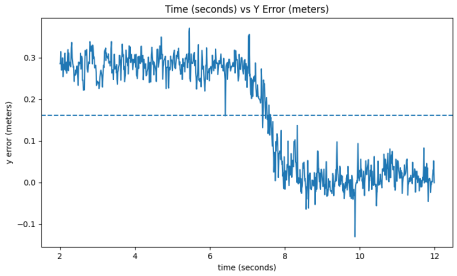


(c) θ error

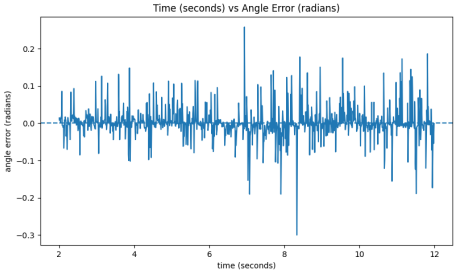
Figure 4: **Choosing a noise parameter of $\sum_u = (0.05, 0.05, \pi/12)$ causes the maximum angle error to be undesirably high**. While our initial noise parameter accurately localizes or robot position, it struggles to find a consistent heading.



(a) x error



(b) y error



(c) θ error

Figure 5: **Modifying the noise parameter to $\sum_u = (0.05, 0.05, \pi/24)$ limits the maximum angle error to 16.6°, accurately localizing our robot**. Halving σ_θ significantly reduced the heading error of our robot, leading to our chosen noise parameter.

3.2.2 Convergence Evaluation

We measured the time for particle convergence, which we defined as our array of particles containing ≤ 5 unique estimated poses under three odometry noise levels. The average of 50 trials is depicted in Figure 6 below.

Odometry noise levels	**Time for convergence (s)**
$[0.0, 0.0, 0.0]$	0.020
$[0.05, 0.05, \pi/24]$	0.262
$[0.1, 0.1, \pi/12]$	0.096

Figure 6: **Our model converg**

For our chosen noise parameters, the average convergence time was 0.2 seconds, which is sufficiently quick to have an accurate localization in both the simulation and in the real world.

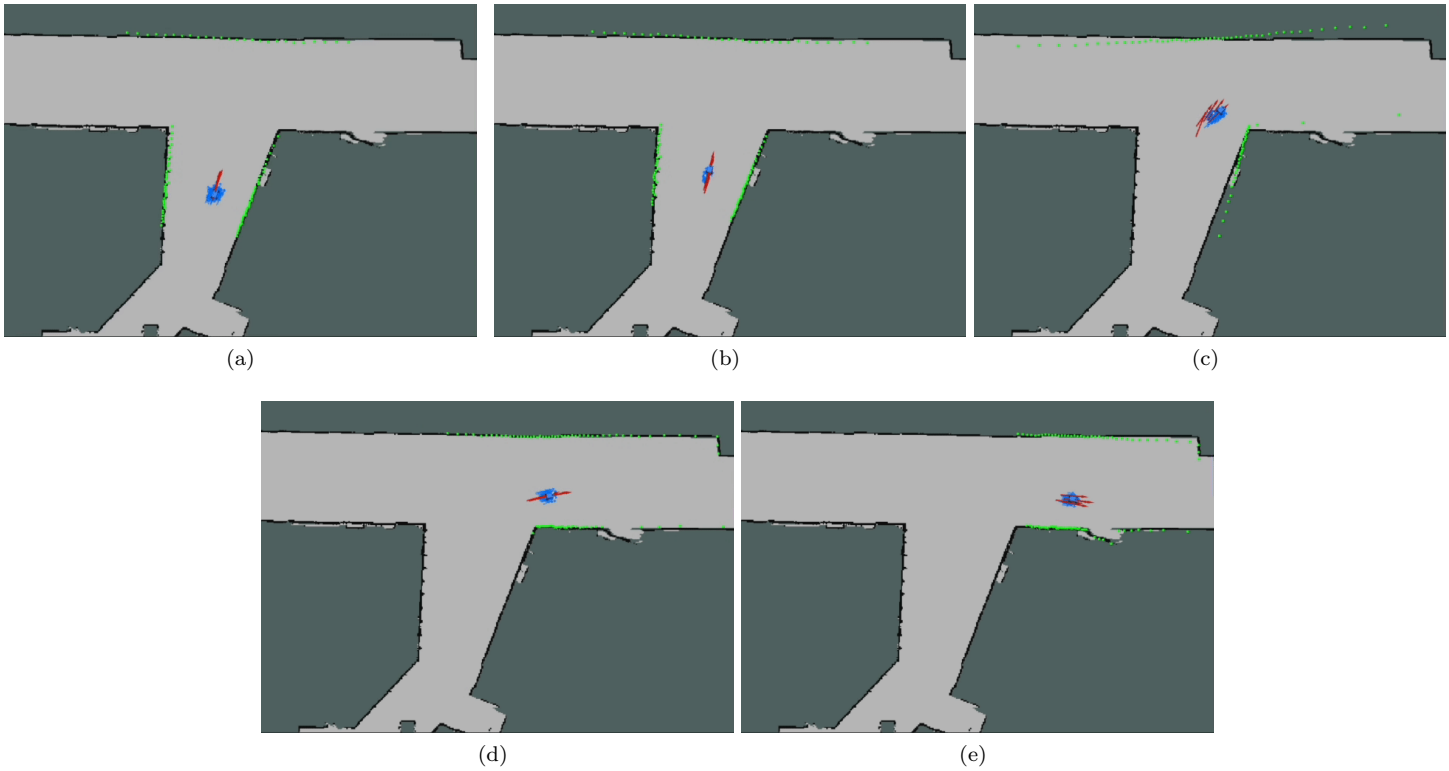
3.3 Testing Procedure in the Real World

In order to test robot localization performance in the real world, we devised three manually controlled trajectories in the Stata basement: turning right (Figure 7), driving in circles (Figure 8), and turning left (Figure 9), starting from the location shown in Figure 3. Each trajectory uses a noise parameter of $\sum_u = [0.05, 0.05, \pi/24]$. We choose these trajectories as they provide unique environments for our robot to localize in; furthermore, these trajectories entail rapid changes in pose between time-steps compared to driving in a straight line, and show our algorithm's robustness to changes in the environment. In order to test our Particle Filter algorithm's robustness to noise, we repeated the last

trajectory, the left turn, with noise parameters of $\sum_u = [0.1, 0.1, \pi/12]$ and $\sum_u [0.2, 0.2, \pi/6]$, as shown in Figure 10 and Figure 11

Although numerical evaluation of the robot's cross track error is unfeasible in the real world, qualitative analysis can be done through Rviz, which shows where the robot thinks it is. In each frame of the figures 7, 8, 9, 10, 11 shown below, the five most recent estimated poses are shown as red arrows, the current sampled particles as blue arrows, and the ray-tracing generated from the current estimated pose as the green scan. We evaluate performance by comparing the trajectory shown in Rviz with the robot's location in real life and checking how well the green scan aligns with the walls of the map in the simulated environment.

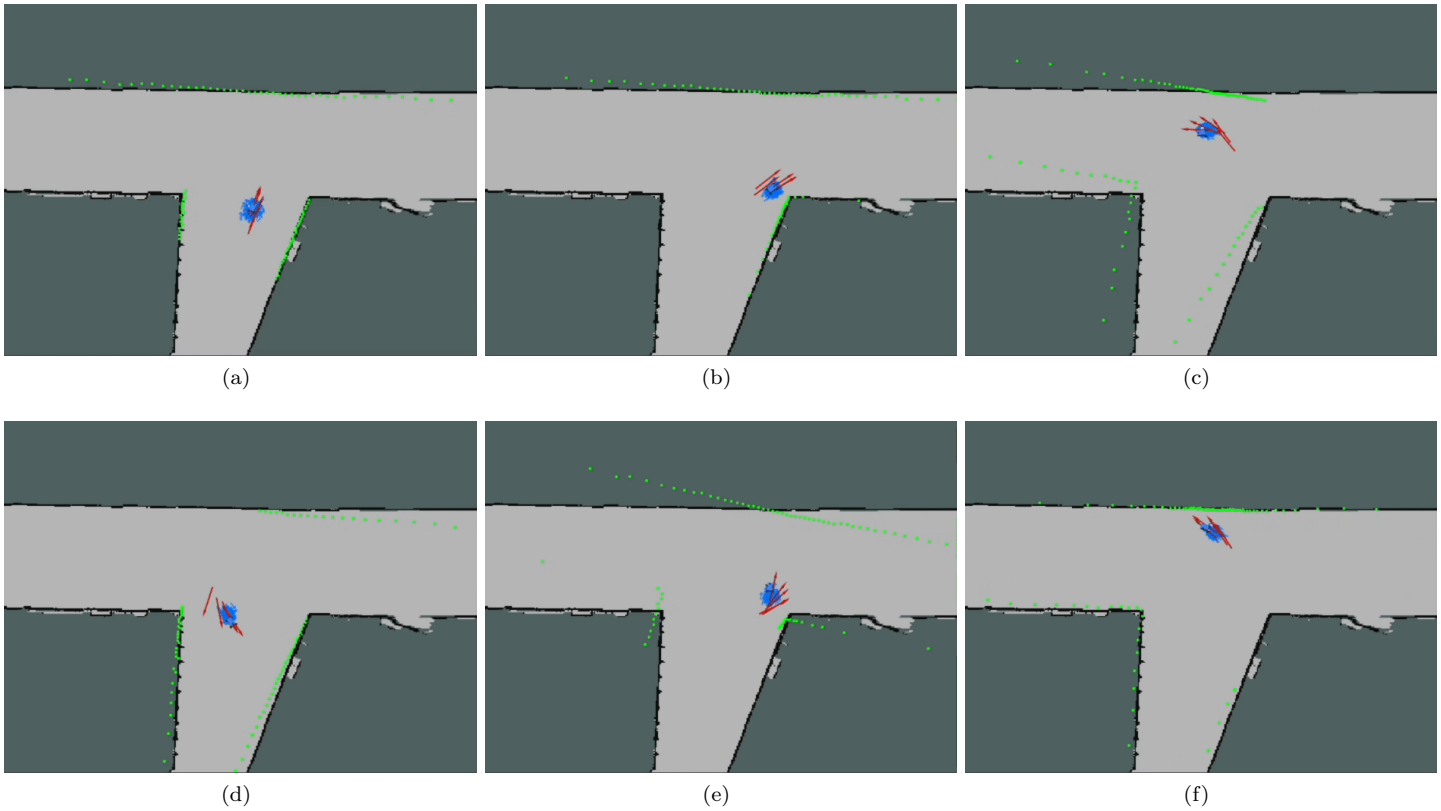
3.4 Performance Evaluation in the Real World



(a) (b) (c)

(d) (e)

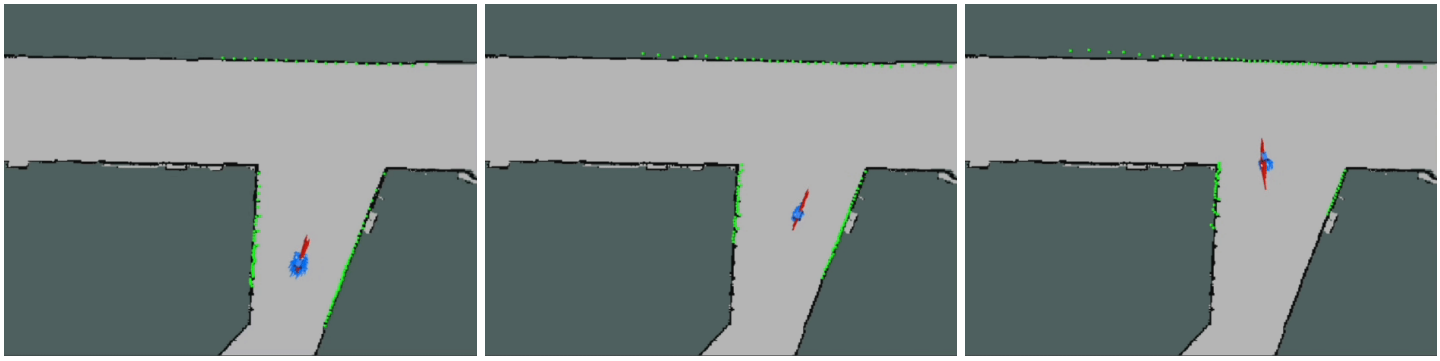
Figure 7: **The green scan aligns closely with the walls of the simulated Stata basement as we drive the robot through a right turn.** We see minor drift in localization in the middle of the turn as a result of the steep changes in the θ component of the pose. The drop in the number of blue particles, as seen from frame (a) to frame (b), confirms the Particle Filter is resampling correctly. Similarly, the increase in blue particles from frame (b) to frame (c) show the Motion Model adding noise as expected.



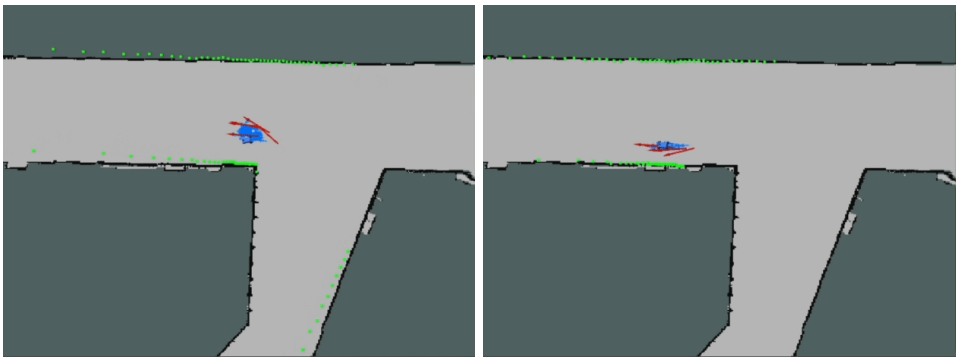
(a) (b) (c)

(d) (e) (f)

Figure 8: **When driven in circles, which entails rapid changes in pose, the robot continues to localize with high accuracy**. Specifically, the green scan continues to align with the simulated walls, except for minor deviations when θ is changing the quickest. We can see the red estimated poses fanning out in along a curve behind the robot in frames (c) and (e), confirming our expectations for how the algorithm will localize.

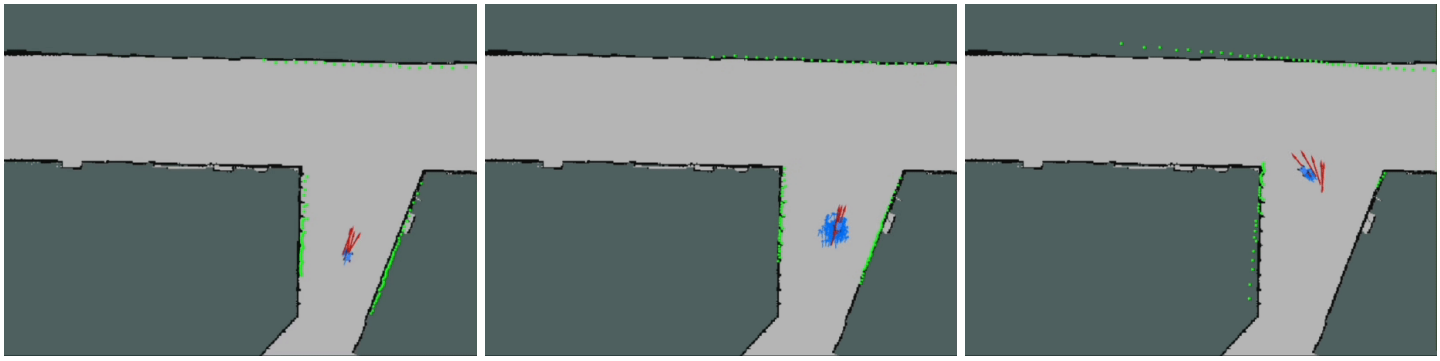


(a) (b) (c)

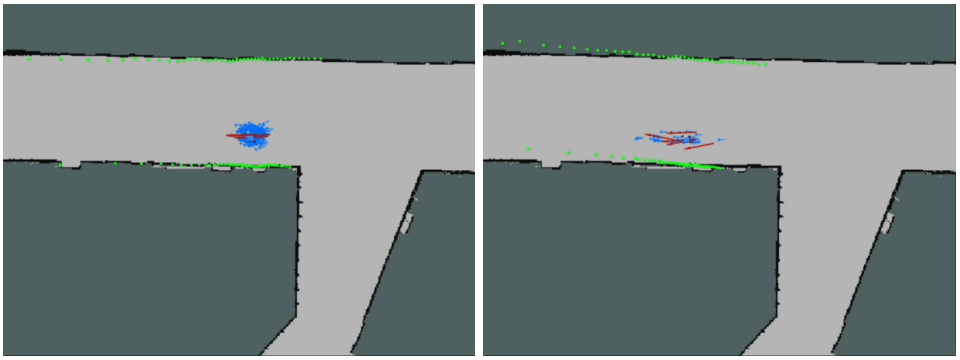


(d) (e)

Figure 9: **Similarly to Figures 7 and 8 shown above, this figure shows good localization as the robot turns left around a corner in Stata basement.** The same analysis as detailed above applies here as well.



(a) (b) (c)



(d) (e)

Figure 10: **The robot follows the same trajectory as in Figure 9 above, driving with a noise parameter of $\sum_u [0.1, 0.1, \pi/12]$.** The spread of blue particles are wider than in Figure 9, but less wide than in Figure 11, to reflect the changed noise parameter. In addition, the convergence of the Particle Filter can be seen from (b) to (c) and from (d) to (e) as points are resampled. The green LiDAR scan continues to show accurate localization, even with the increased noise

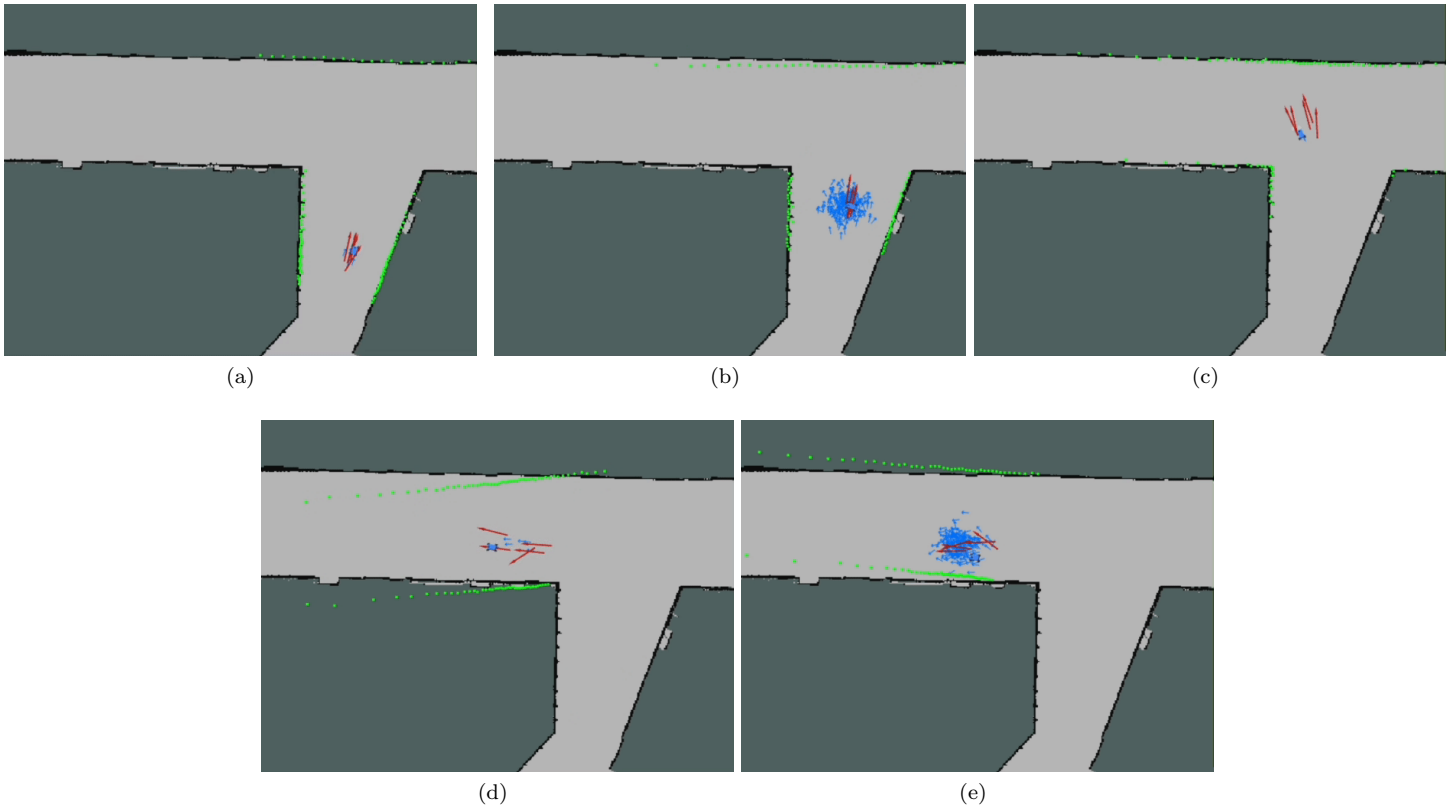


Figure 11: **The robot localizes as it drives with noise parameter $\sum_u [0.2, 0.2, \pi/6]$ being fed by the Motion Model to the odometry.** The wide spread of blue particles and red poses in (b) and (e) reflect the high level of noise, while the simulated LiDAR scan in green shows greater deviations from the correct location. The convergence of the Particle Filter is visible here from (b) to (c). This Rviz trajectory, even with the increased noise, closely matches the actual trajectory of the robot.

4 Conclusion

(Authors: Cynthia Cao)

This report presents a successful implementation of the Monte Carlo Localization algorithm on real-life hardware. A Particle Filter composed of a Motion Model and Sensor Model allows the robot to predict its current location. First, a set of N particles that serve as predictions for the pose are randomly initialized; as the algorithm iterates, each particle is updated by the Motion Model using odometry data. Concurrently, the Sensor Model computes the probability that the robot is at each given particle position using LiDAR data. The Particle Filter then resamples the particles and estimates a final pose that iteratively approaches the ground-truth location of the robot.

Testing in simulation highlights the accuracy and robustness of our algorithm. Average cross-track error values calculated by running our algorithm in simulation show low error between the estimated pose and the ground-truth pose, even when the car is maneuvering through a turn. We find that the algorithm converges at an average rate on the order of centiseconds without noise, and deciseconds with noise added. Fast convergence and low cross-track error allow for a robot that accurately localizes even as the environment changes.

Experimental results from real-life implementations confirm results found in simulations. Figures 7, 8, 9 show the robot effectively localizing in real time while being maneuvered through a right turn, in circles, and through a left turn, respectively. Figures 7, 8, 9 show our localization algorithm's robustness as noise increases.

Potential modifications to improve performance include tuning the α and σ values in Equations (7), (8), (9), and (10) for the Stata basement as opposed to the default values we used. The current design uses a simple probabilistic weighted average of the sampled particles to generate the estimated pose; an alternative method could ensure better performance in the case of multi-modal distributions. We further find that thread locking is unnecessary for this lab, but plan to add it to ensure no conflicts between the Motion Model and Sensor Model in future labs. Finally, given more time, we would vary the number of particles used by the robot at a given time to identify an optimal that increases accuracy while maintaining the speed of the algorithm.

The next phase of our work will use localization for path-planning and trajectory following. In order to successfully plan a trajectory to a goal pose and follow it while avoiding obstacles, the robot must accurately predict its current pose with respect to the map. As such, it is key to implement Monte Carlo Localization successfully as shown in this report.

5 Lessons Learned

Cynthia Cao Over the course of this lab, I learned how to implement Monte Carlo Localization both in simulation and on a real-life car. Through working on the Sensor Model, I gained experience with writing efficient code using numpy functions and slicing. Working with Grace, I was able to develop my collaborations skills when it came to code, especially when it came to talking through the problem, planning how to write the code, and debugging our implementation of the Sensor Model. Furthermore, the time I spent implementing our algorithm on the robot gave me a better understanding of the differences between simulated environments and the real world. Not only am I better equipped to handle similar issues when they come up in the future, but I also learned the importance of writing code with a focus on real-life hardware. Additionally, I learned a lot in terms of communication, especially when it came to taking our individual code modules and combining them together into a final product. Finally, I've spent a lot of time learning how to best put together briefing slides, so this report has helped develop my ability to explain technical issues through text.

Trey Gurga This lab allowed me to really comprehend how localization works after some confusion in lecture and allowed me to collaborate at a much higher level than other labs. Part A gave me a really good refresher on transformations and hands-on practice with the probability portion of MCL which I did not grasp at first. This experience well-equipped me to work with Toya and Jonathan to implement the Motion Model with ease. We began by utilizing a 'for' loop and had fun trying to eliminate it which enhanced our skills with numpy. I took the initial crack at the Particle Filter which exposed me to all parts of MCL and allowed me to interface with the other portion of our team and we had a full-team effort on the implementation which was the first lab that had all 5 of us editing the script at the same time. This gave us some valuable lessons in version control. Similar to other labs, we broke up towards the end into focusing on simulation data, robot implementation, and deliverables planning and this gave me the opportunity to continue to dive deep into working on the intersection of hardware and software. We experienced many more bugs with the real-world implementation this time around and I am much more equipped now to debug complicated programs directly onto the robot and isolate what has changed from our working simulation model and real-world, which is insanely valuable as I will be working with hardware in the future.

Grace Jiang Before beginning on the coding portion, I learned about the fundamentals of Monte Carlo Localization algorithm. Afterwards, I worked on the actual implementation of the Sensor Model more than other parts of the algorithm, so I understood the Sensor Model in more depth. When talking through the Sensor Model with Cynthia, she was the one coding it, as it didn't make sense for two people to code the same thing twice. This lab was perhaps my first experience coding together with someone without doing any actual typing. Similarly to Lab 4, we split up individual code modules then combined efforts to integrate them together to run on the real robot. Later, as Cynthia and I debugged our algorithm on the real robot, we eventually found an issue in way we do downsampling of the observed LiDAR scans. For us, finding this bug was a moment of realization, since I remember thinking the day before that one of the only differences between simulation and real life was the number of points in the LiDAR range data. Debugging with someone else was easier than what I imagine it would be by myself, and I learned that debugging together can be more efficient than working individually.

Toya Takahashi The localization lab allowed me to develop both technical and communication skills. Working together as a team, I initially learned how Monte Carlo Localization works in a high-level. Part A of the lab was a good refresher on rigid transformations and was useful to understanding how both the Motion Model and the Sensor Model can be implemented in the code. Working with Trey and Jonathan on the Motion Model, I also practiced pair programming to prevent mistakes and write code efficiently, and I gained a deeper understanding of the Motion Model. Finally, I learned that integration of code in the real-world is key, as there are always differences between performance in the simulation and the real-world. Through the process of developing the Particle Filter and debugging the robot, I learned how to work effectively as a team to tackle a challenging problem and how to implement a localization program on a real-world robot.

Jonathan Zhang From this lab, I developed a much more in depth understanding of localization. Implementing the motion model and particle filter allowed me to better understand the content from lectures that I was confused about, and making the slides and reports really caused that knowledge to sink in. Working with numpy was also a great refresher. It's always a fun challenge to code without using for loops and instead with matrices.
I also grew not only technically, but also as a team member. Learning to code with pair programming was definitely a challenge from me, as many of my computer science classes are individually based, but after getting used to it I found that I quite enjoyed it. I furthermore had more practice with distributing work as a team while still learning the material that was distributed to another team member. Our communication keeps improving, and our team keeps

(Editors: Cynthia Cao, Trey Gurga, Grace Jiang, Toya Takahashi, Jonathan Zhang)