# Lab 5 Report: Localization

Team 13

Ra Mour (Ronald Alvarez)
Srinath Mahankali
Nandini Thakur
Jorge Tomaylla
Sarah Zhang

6.4200: RSS

April 14, 2023

## 1 Introduction

Author: Ra Mour

To perform many tasks autonomously, a robot must have a sense of its location within its environment. For efficient navigation using path planning, the robot must know from where it's beginning and whether it's making proper progress along the specified path. If a robot finds itself in a completely unfamiliar environment, it must gradually build a representation of the world and simultaneously locate itself within that representation. If we can provide the robot with a prespecified representation of the world—a map—it can combine known features of the map, perceptual inputs, and information about its own movement over time to estimate pose.

In this lab we implement the **Monte Carlo localization** algorithm to iteratively predict the position of our racecar. We use odometry to perform "dead-reckoning" estimation of the change in vehicle position. By taking into account noise in the measurement of odometry, we generate a set of particles that represent potential poses (the motion model). We then simulate observations and compare them with the racecar's LIDAR data to assign likelihood weights to each particle. This *particle filter* approach increases the influence of particles that are more representative of the true pose while pruning particles that are unlikely to be correct. At each iteration, the particle filter takes an average of the weighted particles to propose a pose estimate, and the algorithm repeats

from that pose. We achieve excellent results with our implementation of MCL in simulation and achieve reasonable performance in real-world tests.

# 2 Technical Approach

## 2.1 Motion Model

Author: Nandini Thakur

### 2.1.1 Deterministic Model

A model of the racecar's motion was created to roughly estimate its pose over time, based on its odometry and pose in the previous timestep. First, we assumed a perfect world with no error in the odometry to create a deterministic motion model. The equations for updating the racecar's pose $p_t$ based on its odometry and previous pose $p_{t-1}$ were as follows:

$$\text{odometry}_{world} = \begin{bmatrix} \Delta x_w \\ \Delta y_w \\ \Delta \theta_w \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta x_r \\ \Delta y_r \\ \Delta \Theta_r \end{bmatrix}$$

$$p_t = p_{t-1} + odometry_{world}$$

The odometry in the racecar frame was multiplied by a rotation matrix to find the odometry in the world frame. The rotation matrix is the standard matrix for a rotation around the z-axis, since that is the only way our racecar can rotate. $\theta$ in this matrix is the rotation of the car in the world, since we want to convert points from this rotated frame to the world frame. Once we had the odometry in the world frame, we could easily add it to the racecar's previous pose since that is also expressed in the world frame.

This model is a deterministic model, where the car's new pose is purely a function of its old pose and the odometry we receive. This model only works if the odometry has no error. Unfortunately that can't be assumed, since its very likely that the robot will have moved slightly differently than what the odometry indicates. In scenarios where the car is moving very fast, it's almost certain that the odometry will have error in it, even if just by a few centimeters.

### 2.1.2 Non-Deterministic Model

We accounted for this by adding noise to the odometry to generate many possible odometries that could have occurred based on the one we received, and therefore many particles representing all the locations the racecar might be located. We had some challenges with creating this noise because initially we

added noise to particles after adding the world odometry to their positions. This resulted in noise in the "world frame" rather than in the "robot frame," which meant noise wasn't correlated with the magnitude of robot movement, but rather was just a random amount which didn't make much sense.

We solved this with a new method of adding noise, which was to create $N$ copies of the robot odometry and then add a different amount of noise to each copy. Each version of the odometry is multiplied by a different rotation matrix based on its corresponding particle, which is now effectively the robot for that specific odometry. This method generates $N$ particles of possible racecar locations with noise being correlated to the direction of car movement.

```
def motion_model(robot_odometry):
    multiple_odometry = make_n_copies(robot_odometry)
    noisy_odometry = add_noise(multiple_odometry)
    world_odometry = rotate(noisy_odometry)
    particles = particles + world_odometry
```

Noise is sampled from a normal distribution centered around 0 so that most noise values don't change the odometry much. The noise in every direction $(x, y, \theta)$ is sampled independently from a distribution with different standard deviations, which is necessary because the amount of noise necessary is directly influenced by the magnitude of motion in that direction. Larger movements over one timestep correspond to more likelihood of error in the odometry, and therefore a larger range of possibilities for the actual odometry. The standard deviation of each distribution is calculated by a scale factor multiplied by the magnitude of the motion in that direction. The equations for standard deviation of noise in each of the directions, and the new motion model equations for a single particle $i$ after incorporating noise are presented below:

$$\begin{aligned} std_x &= 2 \cdot \Delta x \\ std_y &= .02 \cdot \Delta y \\ std_\theta &= 1 \cdot \Delta\theta \end{aligned}$$

$$\text{odometry}_{world,i} = \begin{bmatrix} \Delta x_w \\ \Delta y_w \\ \Delta\theta_w \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta x_r + noise_x \\ \Delta y_r + noise_y \\ \Delta\theta_r + noise_\theta \end{bmatrix}$$

$$p_t = p_{t-1} + odometry_{world}$$

The $x$ direction has the largest standard deviation for its noise distribution because that is the direction the robot drives in. The $y$ direction is never driven in, so not much noise is necessary. We keep a small amount of noise to account for cases where the racecar turns and the particles get separated from the car. The $\theta$ noise standard deviation is also slightly larger to account for potential

errors in turn angle. Figures 1, 2, 3 demonstrate the different ranges of possible noise values.

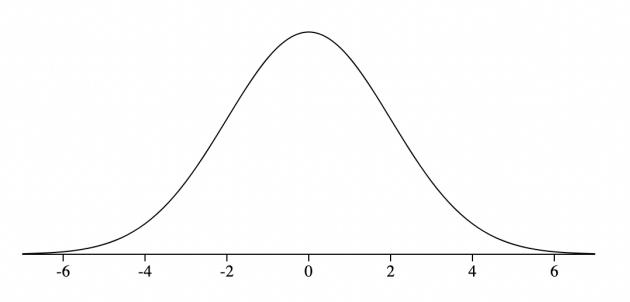


Figure 1: Noise Distribution for X

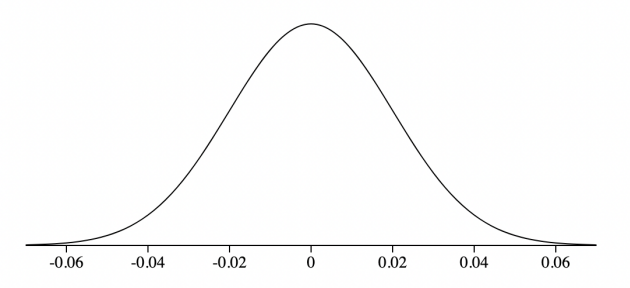


Figure 2: Noise Distribution for Y

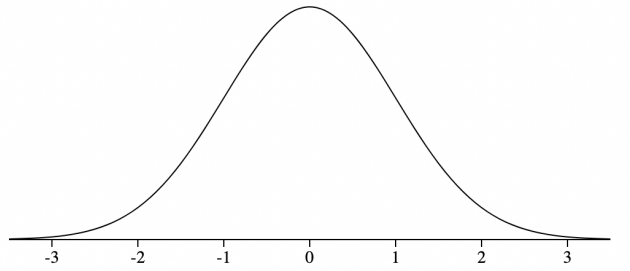


Figure 3: Noise Distribution for Theta

The final non-deterministic motion model has $N$ particles which spread out over time as the position of the racecar becomes more uncertain. This is demonstrated in Figures 4, 5, 6. This spread of the particles will be limited with implementation of the sensor model.

Figure 4: Particles are close together before car starts moving

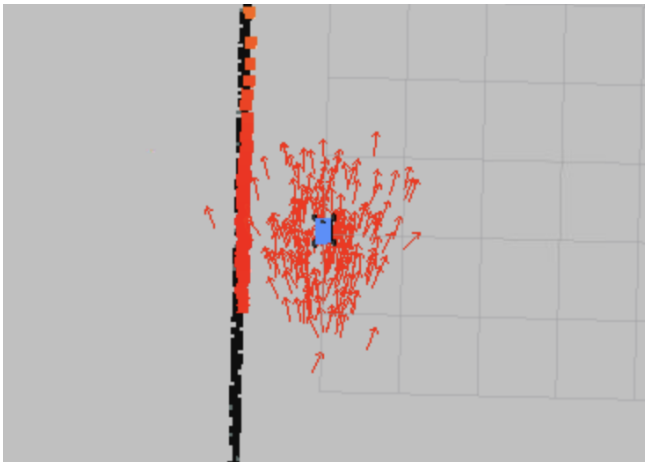

Figure 5: Movement of the car causes particles to spread out due to added noise

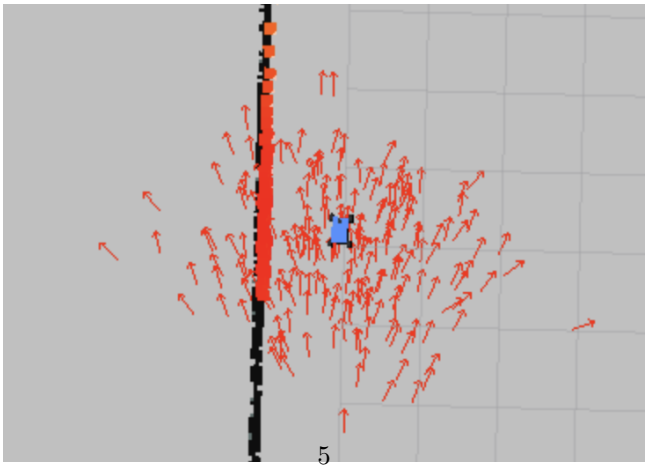

Figure 6: Car has moved a far distance. Location of car has lots of uncertainty

## 2.2 Sensor Model

Author: Ra Mour

### 2.2.1 General Description

Once the motion model has generated particles, we must determine how likely they represent the true location of the vehicle. We use two sources of information about the external world to achieve this: LIDAR data from the vehicle at time k $z_k$ and a static map of the environment. LIDAR data gives us a relatively accurate sense of the vehicle's surroundings. With the map, we can simulate a ray-cast scan $d$—analogous to a LIDAR scan—for each particle. We are interested in answering the question: how much does the ray-cast scan of a particle resemble the LIDAR scan $z_k$? We compare the observations from each particle with the LIDAR scan, and each particle is assigned a weight according to how closely its observation resembles the LIDAR scan. Particles that have a similar view of the environment resembling the racecar observation are assigned higher weights and are carried forward as likely candidates in the location estimation process.

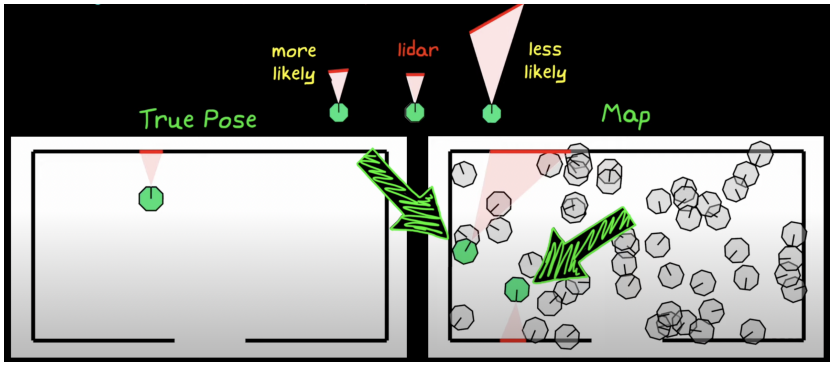


Figure 7: Example of sensor model observation comparison. On the left, there is the vehicle and its LIDAR information. On the right, we have proposed particles for the current time step. Two particles are highlighted and we see their corresponding observations. The particle on the bottom is assigned a higher weight by the sensor model because its observation more closely resembles the LIDAR data.
Source: Understanding the Particle Filter

### 2.2.2 Assigning weights to the particles

Monte Carlo localization uses recursive bayesian estimation (also known as a Bayes' filter) to efficiently estimate the distribution over vehicle states (i.e. particles). The Bayes' filter specifies that the probability of a particle should be updated in proportion to the likelihood of the particle observation matching the LIDAR scan:

$$p(x_k|z_{1:k}) \propto p(z_k|x_k, m)p(x_k|z_{1:k-1}, m)$$

For each particle we have generated at time step $k$, we index into the probabilities table using each element pair from the LIDAR and ray-cast pixel arrays to get weights for each orientation of the scans. Finally, we obtain the total likelihood for a particle by taking the product of likelihoods of all scan orientations:

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

The final likelihoods are used in the particle filter to combine the particles into an "average" estimated pose.

### 2.2.3 Defining the likelihood function

In computing the likelihood function for a LIDAR beam, we take into account certain properties of LIDAR data and of the arrangement of objects in the environment:

1. The measured distance of an object in the environment is normally distributed around its true distance from the vehicle.
2. Potential defects in the LIDAR sensor (such as scratches) can generate short distance readings.
3. Some fraction of measurements will be at the maximum range (indicating that the LIDAR beam didn't return to the sensor), not because there was no object in that direction but due to the reflective properties of the obstacle.
4. There may be other small, unaccounted for sources of noise in the LIDAR sensor.

Each of these assumptions is represented by a component likelihood function:

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

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

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

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

We then take a weighted average of the components to define the likelihood for a single LIDAR beam.

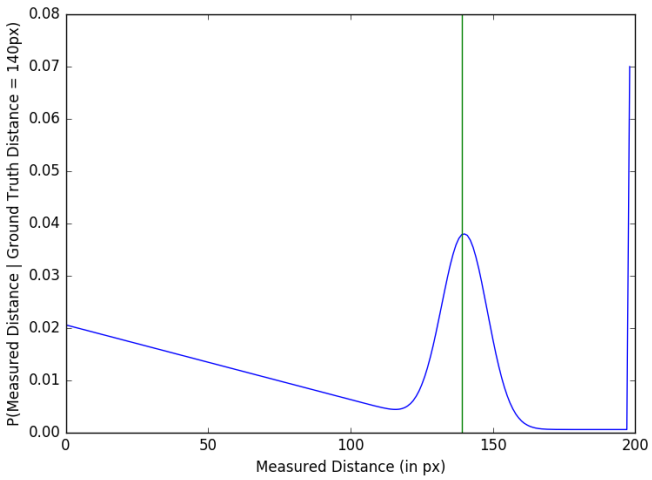


Figure 8: Likelihood as a function of measured distance (for a given true distance value).

#### 2.2.4 Precomputing the probabilities table

LIDAR and ray-cast distances are continuous values measured in meters. Computing the likelihoods from scratch for the infinite possible number of input values would be too slow to be useful in real-time localization. Instead, we bucket the distances into 201 discrete pixels, with distance zero corresponding to the 1st pixel and the max scan distance corresponding to the 201st pixel. Then we build a $201 \times 201$ table of probabilities for each (LIDAR, Ray-cast) distance pair. We can therefore store the likelihoods and efficiently access them as we receive new LIDAR scans and particle observations.

### 2.3 Particle Filter

Author: Srinath Mahankali

Here, we will describe the process of designing the particle filter and key details of its implementation. The overall goal of the particle filter is to estimate the pose of the racecar using the odometry data and LIDAR data, which is essential since these are the only observations available to the racecar. When the racecar is deployed in the real world, it does not have access to its ground truth position!

We first designed our particle filter in simulation, setting ourselves up for iteration by testing its performance using the wall follower module we previously

constructed.

**Integrating the Motion Model and Sensor Model in Simulation** Periodically, the racecar receives both odometry and LIDAR data, after which we update the particles using the motion model and the sensor model, respectively. At a high level, our particle filter updates its particles every time it receives this feedback, and publishes an "average" pose based on the poses and likelihoods of all particles.

**Integrating the Motion Model** Every time the racecar receives odometry data, we update the particles as described in Section 2.1. We made sure to do this in a thread-safe way using Python's `threading` library, as shown in our following code snippet:

```
self.lock.acquire()
self.particles = self.motion_model.evaluate(self.particles,
                    np.array([dx, dy, dtheta]))
self.lock.release()
```

We update the particles based on the odometry measurement at a frequency of `self.rate`, we also normalized the odometry data as following:

```
dx = linear.x/self.rate
dy = linear.y/self.rate
dtheta = angular.z/self.rate
```

We observed that without this normalization, the particles would move forward at a much faster rate than the actual racecar. After this normalization, the movement of the particles was highly correlated with that of the racecar.

**Integrating the Sensor Model** Every time the racecar receives LIDAR data, we obtain the likelihood estimates using the sensor model as described in Section 2.2 and save these probabilities to our particle filter using the command `self.probs=probs`.

We resampled the particles based on these probabilities in order to preserve particles which are more likely to represent the racecar's pose based on the LIDAR data. However, instead of directly sampling according to the probabilities `probs`, we sampled according to the probabilities `probs**.75`. This makes the probability distribution closer to uniform, which allows some noise to remain in the particles. We perform this in a thread-safe way as shown in the following code snippet:

```
probs = probs ** .75
self.lock.acquire()
self.particles = self.particles[np.random.choice(
                    np.arange(self.num_particles),
                    size=self.num_particles,
```

```
                    p=probs/probs.sum())]
self.lock.release()
```

We also only perform this resampling every third time the racecar receives LIDAR data, since we aimed for the resampling process to occur more slowly compared to the motion model.

**Estimating the Racecar's Pose** Before receiving any LIDAR data, we initialize the probability distribution to be uniform over all the particles, i.e. the estimated probability that a given particle represents the actual pose of the racecar is $\frac{1}{\text{number of particles}}$.

As the motion model and sensor model are used, the particle filter updates the pose of all the particles along with the probabilities each particle represents the racecar's actual pose. We estimate the position of the racecar as a weighted average of the positions of the particles, which uses their probabilities to compute the weights. Specifically, we compute these weights as follows:

```
probs = probs/probs.sum()
probs = probs ** 2
```

Note that we square the probabilities before computing the weighted average - this further prioritizes the particles which have higher likelihoods as determined by our sensor model. To estimate the direction the racecar is facing, we take the circular mean of all particles, as described in `https://en.wikipedia.org/wiki/Circular_mean`.

**Smoothing Pose Estimates with Exponential Moving Average** We observed that estimating the pose of the racecar this way led to an estimated pose that was slightly ahead of the ground truth pose in simulation. Based on this, we adjusted our estimate of the position of the car to take the previous estimated position into account. This is shown in the following code snippet using an Exponential Moving Average (EMA):

```
new_x = (1-tau)*new_x + tau*self.prev_x
new_y = (1-tau)*new_y + tau*self.prev_y
```

This has the effect of decreasing the rate at which the estimated position changes, and our goal was to prevent the estimated pose from being too far from the ground truth pose. This also smooths the estimates of poses due to the decreased change in estimated position.

After this, we began deployment of our particle filter on the racecar.

**Tuning the Particle Filter on the Real Racecar** Deploying the particle filter on the real robot in the Stata basement led to different settings than in

simulation. We noticed that the racecar could not move too far from the router, decreasing the scale of experiments we could perform. Furthermore, due to lag, we found that the particle filter could not keep track of the car's pose if it was somewhat far from the router or when it was performing U-turns. We tuned the motion model to have slightly higher noise to allow the particle filter to recover even if most of the particles have a drastically incorrect estimate of the pose. We also resampled particles at a more uniform rate to preserve some of the noise in the particles. Finally, we removed the EMA discussed in the previous section since we felt that there was too much lag in estimating the car's pose.

# 3 Experimental Evaluation

Author: Sarah Zhang

We evaluated our localization model primarily through testing in simulation. After achieving satisfactory performance in simulation, we migrated to the physical car to test in the real-world environment. In this section, we first discuss how we tested our model in simulation and then discuss how we evaluated it in the real world.

## 3.1 Testing in Simulation

We test our localization model using two primary simulation environments: (1) by conducting tests using our wall-following algorithm locally, and (2) by running it on the course-provided online Gradescope test cases.

### 3.1.1 Testing Locally with Rviz and Wall-Follower

We tested our localization model locally in Rviz by running our Lab 3 wall-follower algorithm in the MIT Building 31 basement map that comes with the racecar-simulator package. In this environment, we mainly tested two conditions: (1) running wall-follower along a straight wall and (2) running wall-follower along a corner. The two test paths are illustrated in Figure 9.

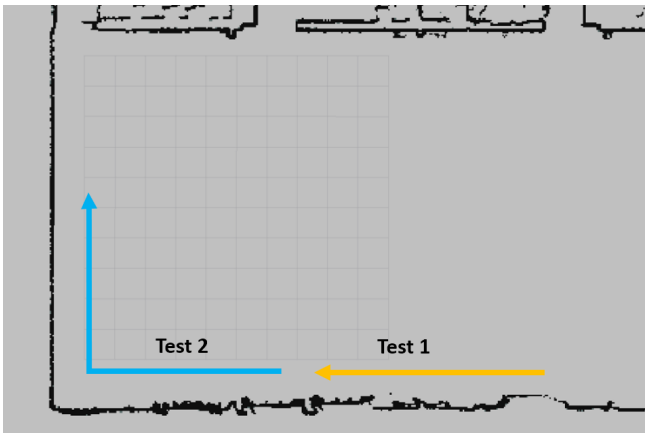


Figure 9: Illustration of two test paths used when testing localization model in simulation.

The metric we measured localization performance with is the difference between the robot's perceived pose $p_M = (x_M, y_M, \theta_M)$ and actual pose $p_A = (x_A, y_A, \theta_A)$. Specifically, we measure and analyze the position error by taking Euclidean distance,

$$e_p = \sqrt{(x_A - x_M)^2 + (y_A - y_M)^2}$$

and for each test case, calculate the average $e_p$ over the duration of the test. We run each test with a small amount of noise in all dimensions in the motion model.

Figure 10 plots the error in Euclidean distance (in meters) and theta (in radians) for each test. We find that the straight line test (Test 1) had an average error of 0.016 m, while the corner turn test (Test 2) had an average of 0.047 m. Observing the error plot for Test 1, we see it stays mostly flat with $e_p$ between 0 and 0.4 m, which reflects desirable performance. Note the error plot for test 2 includes a large plateau around the 3 second mark while the car is turning where $e_p$ rises up to around .15 m, but after the car finishes the turn and starts following the straight wall again, the $e_p$ drops back down to a relatively stable and constant value. For both tests, the theta error stays very close to 0, indicating that the localization model does very well at modeling the direction that the robot is facing. Collectively, these tests indicate our localization model is effective in both straightforward situations and those where it must recover from more uncertain situations.

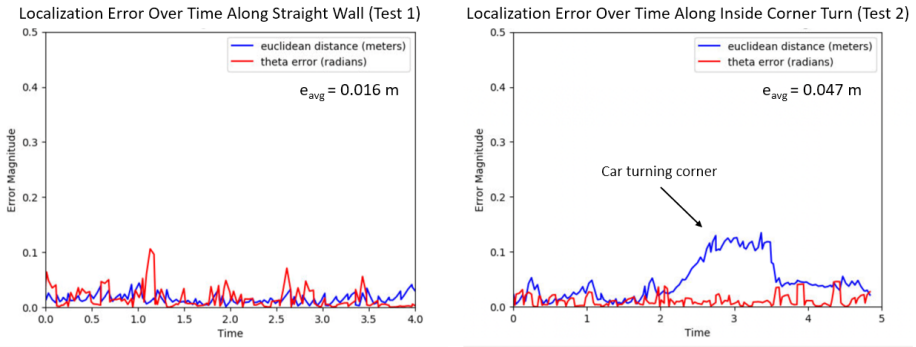


Figure 10: Plots of localization error over time for two wall-following tests. The plot on the left shows the test of running wall-follower along a straight wall and the plot right shows the test of running wall-following along an inside corner.

### 3.1.2 Testing Robustness to Odometry Noise

In order to test our localization model's robustness to odometry noise, we use three course-provided test cases, which simulate moving the car along a pre-set trajectory under three different settings: (1) no odometry noise (Test 3), (2) some odometry noise (Test 4), and (3) more odometry noise (Test 5). The ground truth trajectory test is shown in blue on the top left graph in Figure 11. Test 3 does not apply any odometry noise to the robot's odometry measurements, which is the "perfect world" scenario where changes in motion that the robot reports are true to what the robot actually experiences. Test 5 is a more extreme case where the data our model receives is very noisy, and good performance on this test case demands the motion and sensor model be able to work through this noise.

We evaluate performance on these test cases both quantitatively via the time-average deviation from the trajectory in meters, $d_{avg}$, as well as by qualitatively comparing the localized path with the ground truth path.

Our model achieves a minimum $d_{avg}$ of 0.270 m in the no-noise test case, and a maximum $d_{avg}$ of 0.276 m in the more-noise test case. The more-noise test's $d_{avg}$ is only 2.22% higher than the score of the no-noise test, demonstrating our model is indeed robust to a range of added odometry noise. The results are summarized in Table 1 and illustrated in Figure 11.

| Test | $d_{avg}$ score (m) |
|---|---|
| Test 3, No Noise | 0.270 |
| Test 4, Some Noise | 0.273 |
| Test 5, More Noise | 0.276 |

Table 1: Results of the three course-provided tests applying varying amounts of odometry noise.

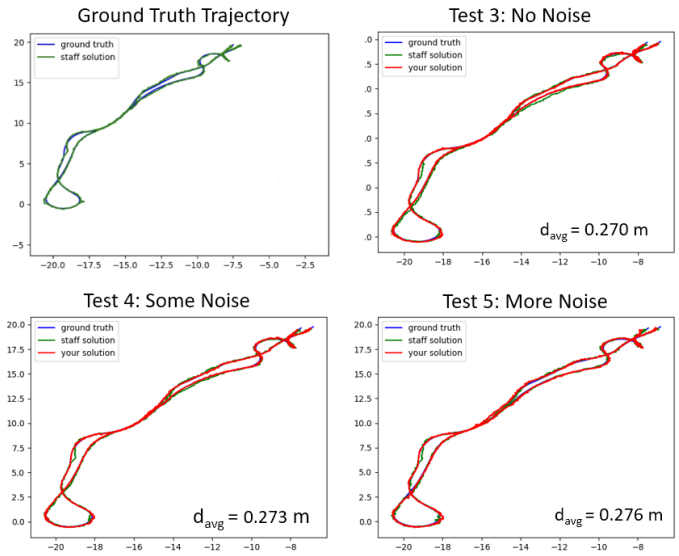


Figure 11: Trajectories generated by course-provided tests. The ground truth is shown in blue and our model's trajectory is shown in red.

Comparing the trajectories from each test, we observe as more odometry noise is added, our model's localized path gets slightly more "spiky" but still for the most part lies very close to the actual path. This confirms that our model can successfully localize the car under both ideal and much noisier situations.

## 3.2 Testing on the Racecar

Evaluating our localization model's performance in real life was much more difficult, as the car does not have access to its ground truth position and so cannot measure the distance between its perceived position to the ground truth position. This means we cannot measure $e_p$. Thus, we rely on plotting the car's path in real time in Rviz and visually comparing it to where we are moving the car in real life. We evaluated our car on two paths: down a straight but somewhat cluttered hallway (Test 6), and a less uniform hallway with several turns (Test 7). The specific paths we tested our robot on are illustrated below

in Figure 12.

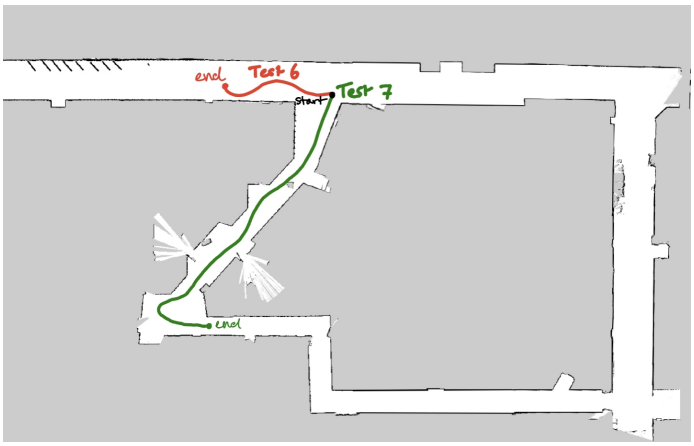


Figure 12: Sketch of paths we drove the car on in the Stata basement for Tests 6 and 7. The black dot indicates the starting position and the red and green dots indicate the end positions of Test 6 and 7, respectively.

**Test 6: Uniform Corridor.** This test evaluates if the car can localize itself in a straight hallway, and we drove the car in a slightly wavy pattern to add variation. The trajectory our localization model produced is shown in green below in Figure 13. We found that the path our model produced is rather noisy (as seen in the spikiness of the path), but relatively accurately reflects the path that we drove the car on.

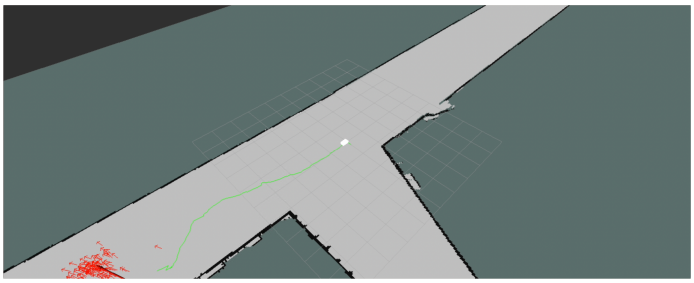

Figure 13: Test 6 results, with the localized trajectory shown in green. The yellow cluster shows the particle sample and the red arrow shows the latest robot pose.

**Test 7: Long Varied Corridor With Turn.** This test evaluates if the car can localize itself through a longer, more varied environment, and included a near-full U-turn. As visible in the trajectory plotted in Figure 14, we observed that the perceived position derived by the localization model appeared to follow

the path we drove the robot on very tightly, again with the caveat of the path being spiky.

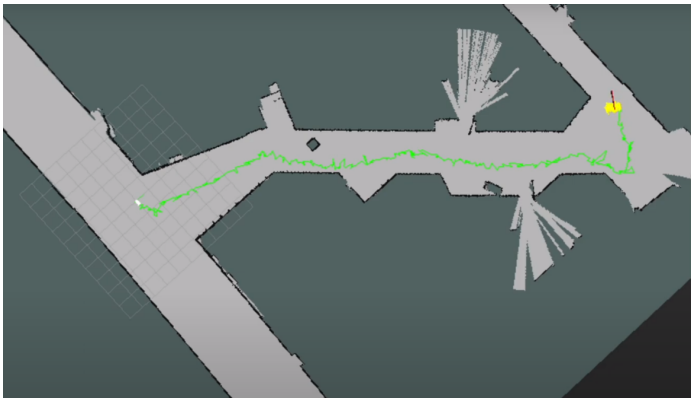

Figure 14: Test 7 results, with the localized trajectory shown in green. The yellow cluster shows the particle sample and the red arrow shows the latest robot pose.

These tests reinforced that our model performs well in noisy environments, as while the hallways are empty in the reference map we provide to the model, in real life there were random objects like pallets, tables, backpacks, chairs, and people scattered about. However, our real-life tests introduce some unpreventable human error because running them is reliant on people placing the car at the precise initial position marked in the rviz simulation. Another large challenge we had to deal with was network issues: often times while trying to run tests in the State basement hallway, we would seem to exit the range of our router and the visualization would lag significantly, making it difficult to evaluate.

# 4 Conclusion

Author: Jorge Tomaylla Eme

To summarize our findings, in this lab we first implemented a robust motion model to estimate the pose of the car over time using odometry readings and data from previous timesteps; then, we implemented a sensor model that helps to measure how likely the estimated location of the vehicle is using visible LIDAR data as well as a static mapping with precomputed probability values. Together, the motion and sensor model combine to create a particle filter that generates the most accurate pose estimation based on odometry and LIDAR data. We also incorporated a cartographer that maps the car's environment in real time in simulation. To complete the simultaneous localization and mapping (SLAM) we will need to run the cartographer in the car, which will prove useful in

the competition challenges. Our results in this lab put us in a great place to design complex path planning algorithms that will allow our car to circumvent obstacles and find the quickest routes to a goal in the next lab.

# 5 Lessons Learned

## 5.1 Ra Mour

I worked with Sarah and Jorge to develop the sensor model. We wrote an initial implementation, began testing our code, and immediately ran into unexpected issues with an early stage of the model. We couldn't even begin testing the second stage until we had corrected that issue. This meant that, by the time the motion model was finished, our portion of the code wasn't ready for integration. The rest of the team moved on with other tasks and had to trust that we would resolve the problem and have our portion ready for final integration. We spent a significant amount of time debugging before we converged on the solution. At that point the problem seemed trivial, and the debugging process felt like a waste of time. In analyzing this feeling I genuinely understood, however, that there was simply no way to immediately arrive at the correction from the very beginning. The possibility of fixing our problem and meeting the need of our team was contained in the tedious effort we made to meticulously review and modify our code.

## 5.2 Srinath Mahankali

I worked on creating the motion model and integrating the motion model and sensor model to design the particle filter in this lab. Through this lab, I learned a lot technically about how to determine the location and direction of the robot despite not knowing the ground truth of these values. One thing that I found counterintuitive was the idea of adding noise into the motion model, and similarly the idea of sampling points at a less aggressive rate in the overall particle filter. It was interesting to me that adding this noise was what allows the particles to "escape" situations where they are all stuck in a bad solution. Similarly, resampling particles less aggressively was what allowed the noise to be preserved.

## 5.3 Nandini Thakur

This lab showed me how helpful simulation can be in robotics to debug and refine the algorithm. We had many issues with generating noise while in simulation, and if these had been immediately tested on the robot then much more time would've been wasted with setup and accounting for external issues. Integration onto the robot was made much faster thanks to our simulation testing. I also learned how to manage time better in this lab, so we weren't rushing on the last day to finish it. Our team picked days early in the week for working on the robot so we had sufficient time after implementation to collect evaluation data and create the presentation.

### 5.4 Jorge Tomaylla

I worked on the sensor model and cartography in this lab. The sensor model was challenging for me because of its heavy load on probability and distributions. I learned to compensate by doing more research and preparation on the topic, as well as helping with other fields that I feel more capable in. I also noticed how hard it is for multiple people to work on the same file, as merging conflicts deter progress so it is better to have one person coding at a time and work on other tasks to maximize progress. From now on we will more efficiently allocate tasks so there is no bottle neck and so that no member of the team is overwhelmed.

### 5.5 Sarah Zhang

I focused on the sensor model and particle filter optimization in this lab. Overall, this experience reinforced the importance and usefulness of working in parallel, as we were able to break down the group into two subteams to work on the sensor and motion model, and when the motion model subteam finished their work before the sensor model subteam, they were able to get a start on implementing the particle filter. This meant that once the sensor model was also done, the entire team was able to efficiently come together again to test, tune, and optimize the entire localization model. Overall, I felt we managed our time well and completed the projected according to our initial timeline.