6.4200/16.405 SP23 - Lab #5 Report: Determining a Robot's Orientation and Position in a Given Environment via Monte Carlo Localization

Team #20

Aaron Zhu
Jessica Rutledge
Chuyue Tang
Nihal Simha
Shara Bhuiyan

6.4200/16.405 Robotics: Science and Systems (RSS)

April 15, 2023

1 Introduction (Aaron)

Tasks such as traveling to a predestined target location that require a sequence of planned sub-tasks allow for more effective work to be done in the environment. Previously, the tasks we commanded our system to autonomously execute were relatively simple. Whether it was using LiDAR distances in the wall follower or color segmentation in the line follower, driving commands could be directly derived in real-time by minimizing sensed error feedback. As we continue our pursuit to achieve increasingly complex tasks eventually through motion planning, we are currently met with tackling perception. Perception goes beyond sensing to create a more coherent internal model for understanding the world. In this lab, we specifically examine the problem of localization, which is to determine the estimated pose of our race car given sensor data and a known map of the environment.

However, localization in general is challenging because of its need to model the uncertainty that exists in the real world. First of all, localization relies on imperfect, noise-containing sensor data which easily leads to erroneous estimates of the robot's pose. In addition, localization's reliance on a given static map makes it unable to account for any dynamic changes that may occur in the environment. Last but not least, there exists a large trade-off in localization between computational speed and accuracy. All such challenges must be addressed in order to achieve a localization module that is robust to various real-world disturbances that may hinder the intended performance of our race car.

This paper demonstrates the following contributions:

- Our Monte Carlo Localization (MCL) algorithm is designed to utilize both proprioceptive (internal) and exteroceptive (external) sensor measurements to estimate the pose of our race car in a given environment.
- Through experimental analyses, we show both in simulation and on the real race car that our MCL algorithm, after sufficient tuning of parameters, is able to successfully localize the car with relatively high accuracy.
- We extend beyond the lab requirements to construct our Simultaneous Localization and Mapping (SLAM) algorithm, drawing upon concepts from our MCL to do so. In simulation, we show that our race car is able to build a map representation of the environment while localizing itself within it at the same time.

We detail the technical implementations of our MCL algorithm in Section 2 and evaluate our experimental results in Section 3. We briefly discuss our extra credit implementation and analysis of SLAM in Section 4. Our conclusions are presented in Section 5 and our personal lessons learned are described in Section 6. Videos and images associated with this lab are available here.

2 Technical Approach (edited by Aaron)

This section details our Monte Carlo Localization (MCL) algorithm. At a high level, MCL combines information from both a motion model and sensor model that utilize odometry and LiDAR data respectively to build a particle filter. By updating the poses of previously-initialized particles located in the given environment, an estimate for the race car's pose can be determined. We begin by detailing the theory behind the motion and sensor models separately before discussing how the particle filter constructs the premise of the MCL algorithm.

2.1 Motion Model (Nihal)

The motion model obtains proprioceptive measurements of the rotational speed of the drivetrain using internal encoders which provide the odometry of the car. The odometry is represented by the vector $[\Delta x,\ \Delta y,\ \Delta\theta]$ which provides the respective x, y, and θ changes in position from the body frame of the car. Figure 1 summarizes the motion model calculations.

Given an initial pose $[x_W^t,\ y_W^t\ \theta_W^t]$ at time t, Equation 1 dictates how the pose of the car can be propagated to $\left[x_W^{t+1},\ y_W^{t+1}\ \theta_W^{t+1}\right]$ at a time $t+1$. A 2D rotation matrix converts the odometry to the map frame, and it is then simply added to the previous pose at time t. However, measurement values obtained from the sensors are not perfectly accurate on the actual car due to various sources of noise inherent with such sensing. To model such phenomenon, noise in the x, y, and θ dimensions are further added as Gaussian distributions centered around a mean of 0. Standard deviation values are then tuned to capture the spread of noise present in the data. Tuning of the noise standard deviation values is discussed later in Section 3.

$$\begin{bmatrix} x_W^{t+1} \\ y_W^{t+1} \\ \theta_W^{t+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta_W^t) & -\sin(\theta_W^t) & 0 \\ \sin(\theta_W^t) & \cos(\theta_W^t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \Delta x + N(0, \sigma_x) \\ \Delta y + N(0, \sigma_y) \\ \Delta\theta + N(0, \sigma_\theta) \end{bmatrix} + \begin{bmatrix} x_W^t \\ y_W^t \\ \theta_W^t \end{bmatrix} \quad (1)$$

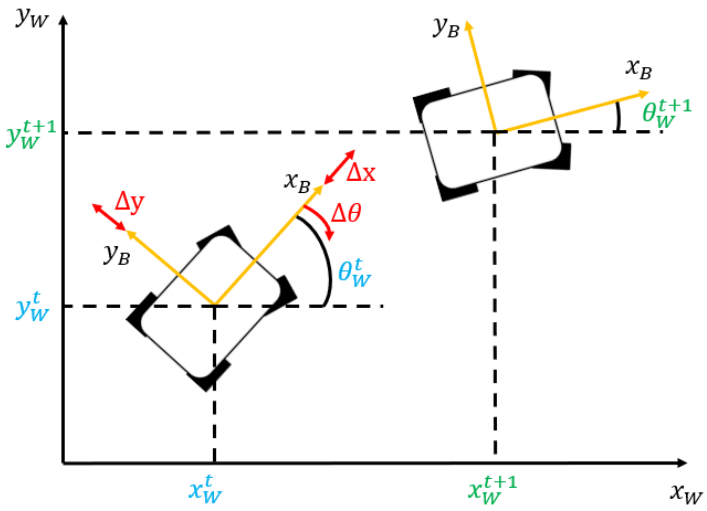


Figure 1: Graphical summary of the odometry data and the motion model calculations. The sensed change in the x, y, and θ directions from the odometry data is used to estimate the pose of the race car at the next time step. As with all sensor data, however, the measurements obtained can not be guaranteed to be fully accurate. As a result, we account for such uncertainty by adding random noise values sampled from a Gaussian distribution. Respective standard deviation values were experimentally determined (see Section 3.1). (Figure Credit: Nihal)

Algorithm 1 outlines the optimized process of propagating the pose of the particles using the odometry of the car over each timestep.

Algorithm 1 Motion Model

Require: PARTICLE-FILTER, SAMPLE-NORMAL-DIST, SIN, COS, MAT-MUL

function MOTION-MODEL(*particles*, $[\Delta x,\ \Delta y,\ \Delta\theta]$)
  for $particles_i$ **in** *particles* **do**
    $R_b^W \leftarrow [\text{COS}(\theta_W^t),\ -\text{SIN}(\theta_W^t),\ 0;\ \text{SIN}(\theta_W^t),\ \text{COS}(\theta_W^t),\ 0;\ 0,\ 0,\ 1]$
    $noise_x \leftarrow$ SAMPLE-NORMAL-DIST$(\Delta x,\ \sigma_x)$
    $noise_y \leftarrow$ SAMPLE-NORMAL-DIST$(\Delta y,\ \sigma_y)$
    $noise_\theta \leftarrow$ SAMPLE-NORMAL-DIST$(\Delta\theta,\ \sigma_\theta)$
    $particles_i \leftarrow$ MAT-MUL$(R_b^W,\ [noise_x,\ noise_y,\ noise_\theta]) + particles_i$
  return *particles*

2.2 Sensor Model (Chuyue)

The sensor model utilizes data from the LiDAR scans to compute the likelihood of each particle being the actual pose of the race car. Through modeling various types of probabilistic uncertainty and writing computationally-efficient code, the sensor model is able to obtain likelihood values that are used to update the MCL algorithm.

2.2.1 Beam-based Probability

Given the particle's pose $\mathbf{x}$ and the map model m, the sensor model computes the likelihood of a sensed scan measurement $\mathbf{z}$. This is done by first using ray casting to obtain $\mathbf{d}$, the ground-truth scan that the particle would see if it had pose $\mathbf{x}$. Then, the beam-based sensor model computes the total likelihood of the scan measurement as the product of each independent beam's likelihood. Assuming the laser scan data contains K beams, this calculation is expressed more formally as the following:

$$\mathcal{P}(\mathbf{z}|\mathbf{x}, m) = \mathcal{P}(\mathbf{z}|\mathbf{d}) = \prod_{k=1}^{K} \mathcal{P}(z_k|\mathbf{d}) = \prod_{k=1}^{K} \mathcal{P}(z_k|d_k) \tag{2}$$

If the sensor model were to not take any uncertainty into account, it would simply give an indicator for whether or not each scan measurement z_k exactly matches the corresponding ground-truth measurement value d_k (as previously obtained through ray casting):

$$\mathcal{P}(z_k|d_k) = \mathbb{I}\{z_k = d_k\} \tag{3}$$

However, such a naive model is too strict to be useful. Especially with inherently noisy measurements, it would be rare to find instances where scan values would match up exactly. Rather than only returning a non-zero value for an exactly matching scan, it is more beneficial to have a probabilistic model that returns a non-zero likelihood whose magnitude is greater when the measurement is closer to the ground-truth value. To accomplish this, a Gaussian distribution centered around the ground-truth scan value d_k is used to broaden the probability distribution for each sensed scan value z_k:

$$\mathcal{P}(z_k|d_k) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k - d_k)^2}{2\sigma^2}\right) \tag{4}$$

The sensor model further accounts for many other sources of uncertainty that exist in the real world. Specifically, we model four possible types of uncertainty:

- $\mathcal{P}_{hit}$: measurement uncertainty when the scan hits the obstacle (η is the normalization factor since the meaningful value range is $[0, z_{max}]$ and z_{max} is the maximal value that the laser can bounce back to give the measurement in limited time)

$$\mathcal{P}_{hit}(z_k|d_k) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k - d_k)^2}{2\sigma^2}\right) & \text{if} \quad 0 \le z_k \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

- $\mathcal{P}_{short}$: environment noise when the scan hits some unknown obstacle before hitting the expected obstacle

$$\mathcal{P}_{short}(z_k|d_k) = \begin{cases} \frac{2}{d_k}\left(1 - \frac{z_k}{d_k}\right) & \text{if} \quad 0 \le z_k \le d_k \quad \text{and} \quad d_k \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

- $\mathcal{P}_{max}$: maximal value when the obstacle exceeds the maximal measure range of the laser (in programming, discretization is used and $\epsilon = 1$)

$$\mathcal{P}_{max}(z_k|d_k) = \begin{cases} \frac{1}{\epsilon} & \text{if} \quad z_{max} - \epsilon < z_k \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

- $\mathcal{P}_{rand}$: laser noise when the laser produces some random value

$$\mathcal{P}_{rand}(z_k|d_k) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_k \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

However, when uncertainty enters the sensor model, there is no definitive way to classify which type the noise originates from. As a result, the sensor model uses a linear combination of the four aforementioned probabilistic models:

$$\begin{aligned} \mathcal{P}(z_k|d_k) = \alpha_{hit} \cdot \mathcal{P}_{hit}(z_k|d_k) + \alpha_{short} \cdot \mathcal{P}_{short}(z_k|d_k) + \\ \alpha_{max} \cdot \mathcal{P}_{max}(z_k|d_k) + \alpha_{rand} \cdot \mathcal{P}_{rand}(z_k|d_k) \end{aligned} \tag{9}$$

where the condition $\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1.0$ must be satisfied. The final values chosen for the weights are $\alpha_{hit} = 0.74, \alpha_{short} = 0.07, \alpha_{max} = 0.07$, and $\alpha_{rand} = 0.12$ as was given in the lab description. A visualization of the sensor model's probability distribution can be seen in Figure 2.

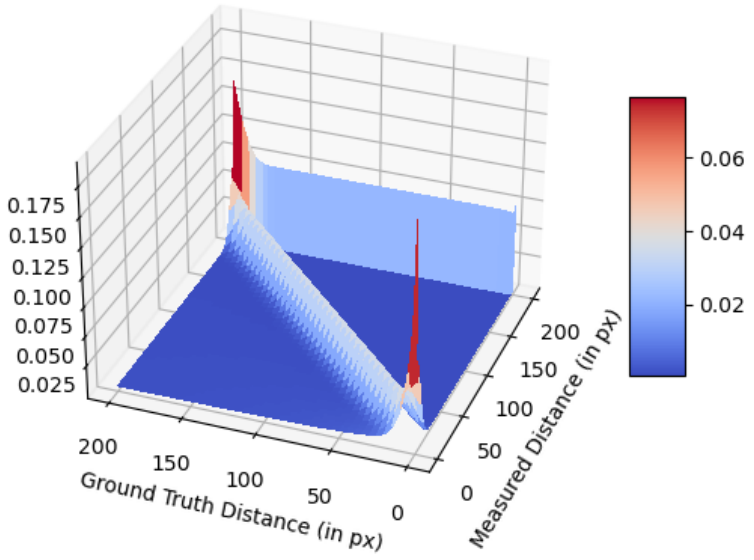


Figure 2: 3D visualization of the sensor model probability distribution with values of $\alpha_{hit} = 0.74, \alpha_{short} = 0.07, \alpha_{max} = 0.07$, and $\alpha_{rand} = 0.12$. The sensor model is a linear combination of four different probabilistic models that attempt to account for certain types of uncertainty. By precomputing the probabilities into a lookup table, values can be quickly accessed for computations by indexing with the given ground-truth distance and the measured distance values. (Figure Credit: Chuyue)

2.2.2 Code Efficiency

A challenge with the sensor model is its need to process a large number of computations over a large number of particles. Being able to obtain the current pose of race car instantaneously can be crucial for certain implications (for instance, collision detection),

especially with a fast-moving race car whose pose may change drastically within a short timestep. Thus, maximizing the efficiency of the code is imperative to the success of any localization algorithm. The lab requires that particles are able to be updated at a frequency of at least 20 Hz. To achieve this frequency, three methods are used to efficiently calculate the likelihoods for a large number of particles.

First, the probability function is discretized into a precomputed lookup table. Since probability values will be retrieved a large number of times (which grows linearly with the number of particles, the number of beams, and the running time), the ability to quickly lookup a value reduces computational time significantly. Probabilities can be simply retrieved by indexing during execution time:

$$\mathcal{P}(z_k = i | d_k = j) = LT[i, j], \quad i, j \in \{0, 1, \cdots, z_{max}\} \tag{10}$$

The lookup table LT is computed with the usage of `np.meshgrid` and matrix masking. Figure 3 details specific examples of the matrix masking process in code.

```
def precompute_sensor_model(self):
    rang = np.arange(z_max+1)
    #obtain two matrices where each corresponding position
    #represents a possible value assignment to z_k and d_k
    Z, D = np.meshgrid(rang, rang, indexing='ij')
    # More code here! (Not shown)

    # Case 4: random value (One case shown for simplicity)
    mask4 = np.multiply((Z >= 0), (Z <= z_max))
    P_rand = mask4 * (1.0 / z_max)
    # More code here! (Not shown)

    #linear combination
    self.sensor_model_table = P / P.sum(axis=0, keepdims=1)
```

Figure 3: Overview of the implementation of `precompute_sensor_model()`. Matrix masking along with various numpy functions are used to efficiently construct the precomputed lookup probability table. (Figure Credit: Chuyue and Shara)

Next, numpy arrays are used to serve as an index into the table to retrieve the likelihoods for all beams and for all particles. The observation from the car is a 1D array of length b while the particle scans constitute a 2D matrix of shape p x b (where b is the number of beams and p is the number of particles). Observations and scans are matched up with the use of `np.repeat` to then serve as indices into the `self.sensor_model_table` (precomputed probability lookup table). Figure 4 details the process in pseudocode.

```
def evaluate(self, particles, observation):
    scans = self.scan_sim.scan(particles)
    # More code here! (Not shown)
    # downsampling and rescaling
    observation = observation[np.newaxis,:]
    observation = np.repeat(observation, N, axis=0)
    # More code here! (Not shown)
    # rounding and clipping
    probability = self.sensor_model_table[observation, scans]
    probability = np.prod(probability, axis=1)
    probability = np.power(probability, 1.0/2.2)
    return probability
```

Figure 4: Pseudocode for implementing the `evaluate()` function. The observations and scans are directly used (after rounding and clipping process) as indices to index into the precomputed lookup table to obtain the needed probabilities. (Figure Credit: Chuyue and Shara)

Lastly, to further increase computational efficiency, downsampling is introduced. The LiDAR scan that the race car receives contains more than 1000 beams. Since many of these beams offer relatively redundant information (due to the small difference in angle measured from), a downsampling procedure occurs which only retains one beam out of every given fixed interval (i.e. 1 beam out of every 10 beams). Ultimately, we choose to downsample to use only 100 beams, as recommended in the lab description. With less beams, less probabilities need to be looked up to compute the likelihood for each particle, ultimately reducing the time needed to fully execute the sensor model.

In all, the three methods utilized help to ensure computational efficiency, which serves essential to the overall success of the localization algorithm.

2.3 Particle Filter (Aaron & Chuyue)

The particle filter utilizes both the motion and sensor models to execute the actual localization process of the MCL. We detail how the particle filter uses both models separately to update the particles, and how an estimate for the race car's pose is extracted [1]. A discussion of our usage of threading locks follows. The particle filter is summarized in Figure 5.

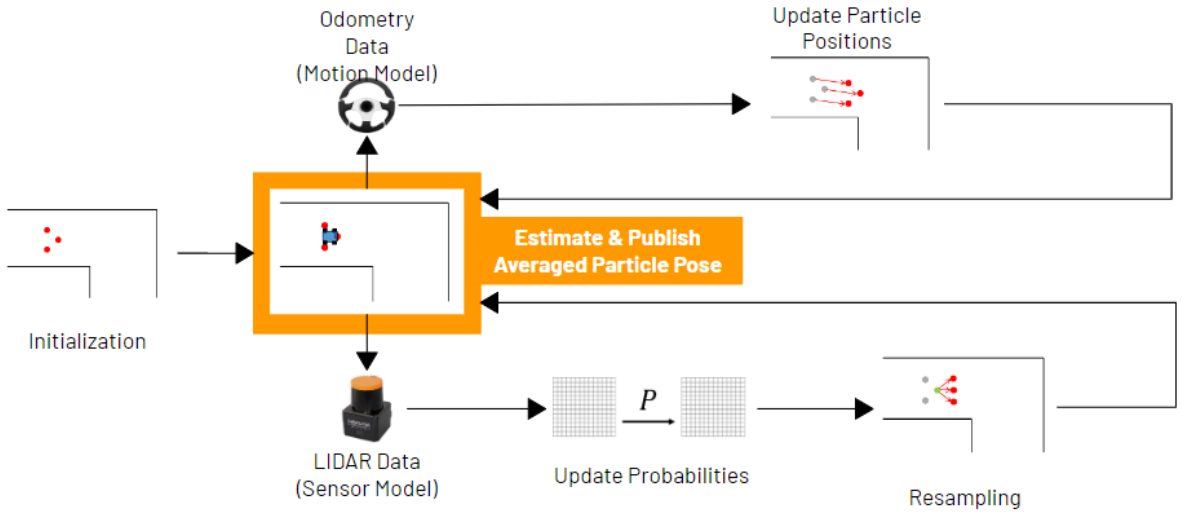


Figure 5: Illustrative summary of the particle filter and MCL algorithm. Starting with an initialization of particles around a given initial pose, either the motion model or sensor model is called one at a time depending on if odometry or LiDAR data is received, respectively. The motion model updates the particle poses, and the sensor model calculates each particle's probabalistic likelihood before particles are resampled. Every time particles are updated, an estimate for the race car's pose is determined via averaging the particle poses, which is published to the race car. (Figure Credit: Aaron)

2.3.1 Particle Updating and Pose Estimation

First, the initialization process is evoked by the `/initialpose` topic subscriber callback. The estimated pose is initialized with a provided starting pose $\mathbf{x_0} = [x_0, y_0, \theta_0]$. Particles are then initialized with a Gaussian distribution centered around the starting pose. The standard deviation of each dimension is empirically tuned in experiments. Equation 11 mathematically captures how particle poses are initialized with the Gaussian noise.

$$\mathbf{x} = \mathbf{x}_0 + \mathcal{N}_3(0, \sigma) \tag{11}$$

Once the particles have been initialized, the particle filter draws upon the motion model or the sensor model, depending on which data type is received at that instant in time, to update the localization.

When odometry data is received, the motion model is called. The pose change vector $\Delta\mathbf{x} = [\Delta x, \Delta y, \Delta\theta]$ is first calculated from the odometry data. This is accomplished by only using the `twist` component of the odometry data representing velocity $\mathbf{v} = [v_x, v_y, v_\theta]$. The velocity is transformed into the relative pose change in the body frame of the car by multiplying the elapsed duration of the timestep:

$$\Delta\mathbf{x} = \mathbf{v} \cdot \Delta t \tag{12}$$

$\Delta\mathbf{x}$ is then used as the input into the motion model, and the particle poses are correspondingly updated (as shown in Equation 1).

On the other hand, when LiDAR data is received, likelihoods for each particle are calculated via the sensor model. Then, particles are resampled based on the probabilities.

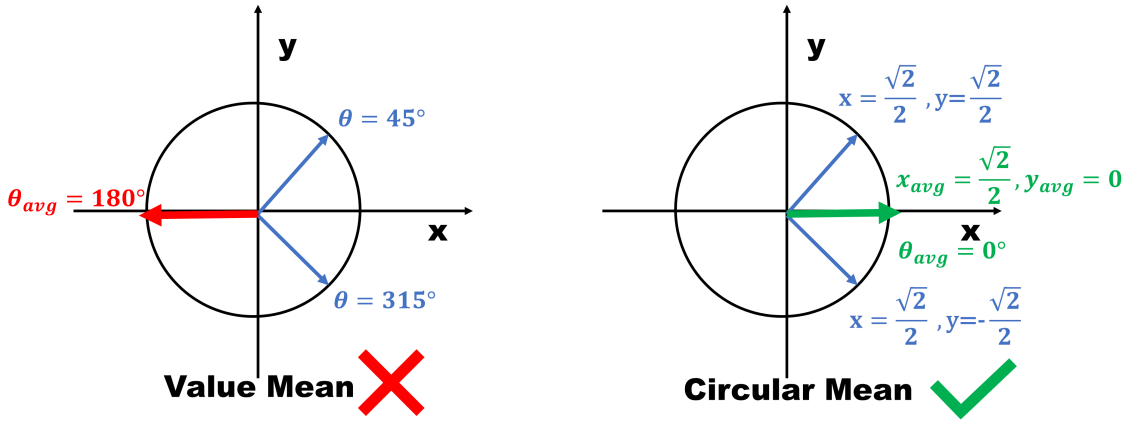


Figure 6: A comparison between two ways of computing average heading direction θ. Assume that we have two particles with heading directions 45° and 315°. The regular value mean will produce an average $(45 + 315)°/2 = 180°$, which deviates a lot from the expected heading direction. The circular mean will represent each direction as a vector on a unit sphere, take the average of the vector, and compute the direction $\theta = \arctan(\frac{y}{x}) = 0$.

Following the execution of either the motion or sensor model (in general, after any updates to the particles are made), the car's most current pose is updated and published to the car as a transform. The car's pose is effectively determined by an averaging of the particle poses. More specifically, the x and y values are calculated as an average of each particle's x and y positions, weighted by their probabilities. This allows particles with higher likelihoods (as calculated through the sensor model) to be more influential in determining an estimate for the race car's current pose. θ is estimated by using a weighted circular mean to account for the periodic value of θ, as shown in Figure 6. In short, the final heading direction is determined by using the weighted (by the particle's likelihood) average of the Cartesian x and y representations of each particle's θ value projected onto the unit circle.

2.3.2 Thread Locking

A key feature related to the implementation of our particle filter is the usage of threading locks. Due to our motion model and sensor model each attempting to update the particle array separately in real-time, it is possible that conflicts may arise during executions; for instance, the motion model may attempt to update particle poses while the sensor model is currently calculating particle probabilities, which will lead to erroneous results. To prevent this issue from occurring, a thread lock is acquired when either the motion model or sensor model is called. The lock is only released upon full execution of the corresponding model. While a thread is locked, any attempts to execute another process on the same thread is blocked. This ensures that each model can safely complete their corresponding calculations and updates to the particle array without any conflicts occurring. Figure 7 illustrates the thread locking process.

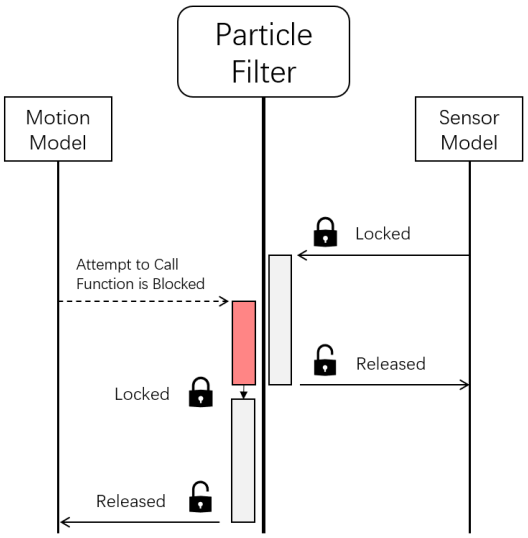


Figure 7: Graphical illustration of thread locking and process execution (middle rectangles). In this example, LiDAR data is received first. This calls the sensor model. A lock is acquired at the beginning of the sensor model execution. While the sensor model is running, it is possible that odometry data may be received by the particle filter. This would trigger the motion model to be called; however, the attempt to execute the motion model is blocked by the lock previously acquired via the sensor model. The motion model is only able to be executed after the sensor model is run to full completion, as the lock is then released. The motion model begins as well by first acquiring a lock. Thread locking ensures processes are able to execute fully to avoid conflicts. (Figure Credit: Aaron)

3 Experimental Evaluation (edited by Aaron)

To evaluate our localization algorithm, we first conduct tests in simulation. Testing in simulation allows us to debug our code and obtain a general understanding of what each parameter does in an optimized, noiseless environment. Once our tests in simulation are successful, we implement the MCL algorithm onto the real race car. Using the general knowledge obtained from the simulated tests, qualitative and quantitative measures are then used to help further tune parameters to make our localization run as accurately as possible in the real world. In this section, we begin by discussing our testing conducted in simulation and then our testing done on the real car to ultimately assess the success of our localization algorithm.

3.1 Simulation Testing (Jessica & Chuyue)

The main objective of testing our localization algorithm in simulation is to finalize our algorithm and prepare it to be implemented onto the physical race car. To conduct testing in a variety of environments, both the wall follower and the parking controller modules from previous labs were integrated into our simulation testing procedures. A variety of qualitative and quantitative data was used to assess the algorithm in simulation. RViz ("ROS Visualization") was used to visualize the odometry data and the localization algorithm's behavior, leading to qualitative analyses. Additionally, accuracy was quantitatively determined by comparing the car's ground-truth location in the world frame and to the robot's position calculated by our localization algorithm. After imple-

menting these strategies with 1000 particles and a publishing rate of 45Hz, the algorithm is finalized and prepared to be tested on the physical race car.

3.1.1 Tuning Particle Initialization Noise

First, the initialization of particles was examined. In simulation, a Gaussian distribution for the positions and directions of the particles is used. Through trial-and-error procedures, general trends regarding the magnitude of noise added in the particle initialization step were noticed. Values for the standard deviation of noise that were too small caused particles to converge too quickly and not cover a wide-enough hypothesis space to maintain robustness in our algorithm. On the other hand, values for the standard deviation that were too large often took too long to converge, which was exacerbated with additional motion noise. While keeping such trade-offs in mind, empirical testing in simulation led to the values of $\sigma_x = 0.4m$, $\sigma_y = 0.4m$, and $\sigma_\theta = 0.3$ rad to be used.

3.1.2 Determining Feature Reliance through the Wall Follower

Our wall follower was used to begin evaluating the simulated performance of our localization algorithm in a more continuous driving scenario. As seen in Figure 8, the car is able to successfully localize itself as it drives parallel to the x-direction, makes a 90 degree turn, and continues parallel to the y-axis (where coordinate axes are given in the world frame). The red odometry arrow in front of the car and the purple path generated via the localization algorithm are able to overlap the actual path of the car in simulation. Graphs of the x, y and θ errors, as shown in Figures 9a and 9b, demonstrate the overall low accuracy error between the estimated localized position and the actual position of the car during the path driven.

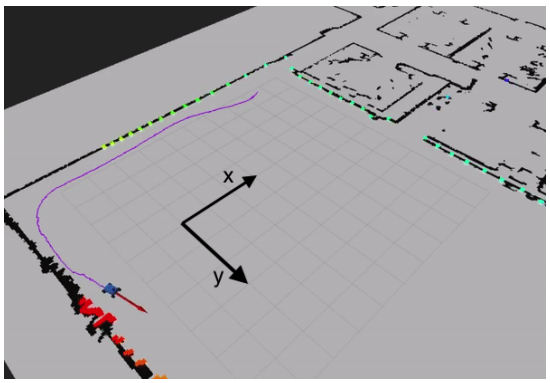


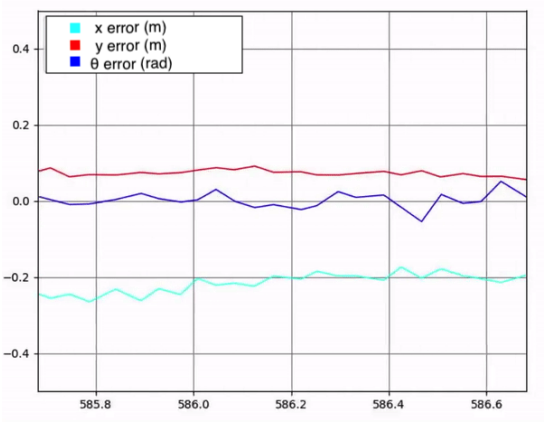
Figure 8: Testing in localization with the wall follower. The red arrow in front of the car is produced by the odometry data from the particle filter. The purple path is also generated as a path trajectory composed of all previous estimated car poses. The x and y coordinate system according to the world frame is also depicted, corresponding to the x and y error recorded in the Error Graphs. (Figure Credit: Jessica)

Further analysis of the error graphs allowed us to gain insight on the strengths and weaknesses of our localization algorithm. In Figure 9a, the y and θ error are both quite minimal; however, the absolute x error converges to a relatively significant value of approximately 0.2m. We hypothesized that this increased error was due to the lack of distinguishable environment features in the x-direction, as there are no clear indications that can be used to verify the correct pose of the race car. Meanwhile, in the y-direction,

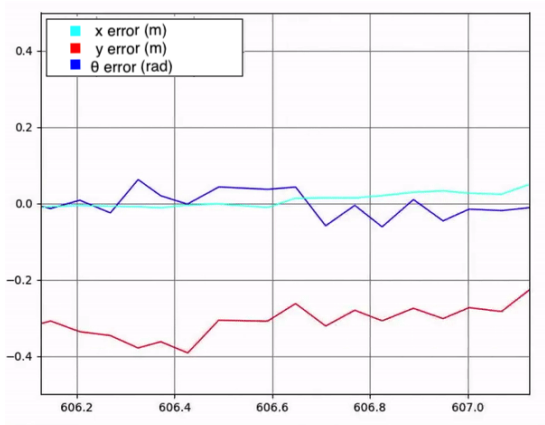
there are more distinguishable features as the car is driving directly next to the wall. This allows the car to localize itself the correct distance away from the wall, resulting in a y-error closer to 0m. This hypothesis was supported after the car turned the corner. As the car now continued driving the the y-direction, there became more distinguishable features in the x-direction (and less in the y-direction). As shown in in Figure 9b, the absolute y-error increased significantly to approximately 0.3m while and the x-error converged to nearly 0m in error.

3.1.3 Ensuring Accuracy with the Parking Controller

After the localization algorithm was successfully implemented using the wall follower, it was further tested with the parking controller to ensure the algorithm remained accurate despite the absence of a clearly recognizable feature (i.e. wall) directly next to the car. The path and final location of this test is shown in Figure 10 with the same red odometry arrow, purple path, and world coordinate system as the previous wall follower test. A portion of the test's $x, y,$ and θ error is also shown in Figure 11. The θ error remained at around 0 radians throughout the entire test. This was most likely because there was hardly any sizeable odometry yaw input received, as the path taken by the car did not require any excessive turning. Both the x and y error ranged between 0m and 0.2m despite the lack of a noticeable feature such as the wall being directly next to the car. The minimal error that resulted from this simulated test, in addition to the successes observed in our previous simulation testing, provided enough confidence in the robustness of our MCL algorithm to be implemented on the physical race car.



(a) Errors when initially driving parallel to the x-axis in the world frame. The θ error is approximately 0 radians, the y-error approximately 0.1m, and the x-error approximately -0.2m. We hypothesize the larger absolute x-error is due to the lack of noticeable features in the x-direction.



(b) Errors when driving parallel to the y-axis in the world frame following the turn. The θ error is approximately 0 radians, the y-error approximately -0.3 m, and the x-error approximately 0 m. The reduction of the x-error and the sharp increase in the y-error supports our hypothesis.

Figure 9: Graphs of the $x, y,$ and θ errors of the MCL algorithm being tested in simulation with the wall follower. An analysis of the error before and after the turn emphasizes the importance of distinguishable features in the environment when trying to accurately localize the race car. (Figure Credit: Jessica)

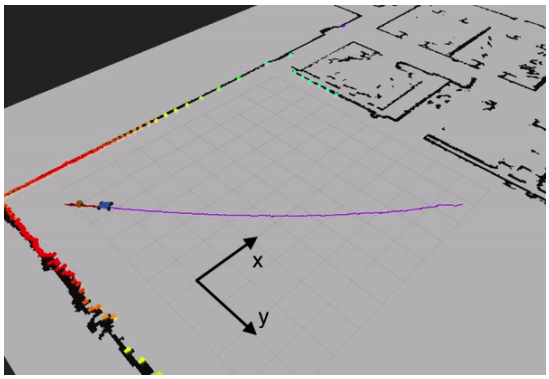


Figure 10: Testing the MCL algorithm in simulation with the parking controller. The x and y coordinate system according to the world frame is depicted, which corresponds to the x and y error recorded in the error graphs (as shown in Figure 11). (Figure Credit: Jessica)

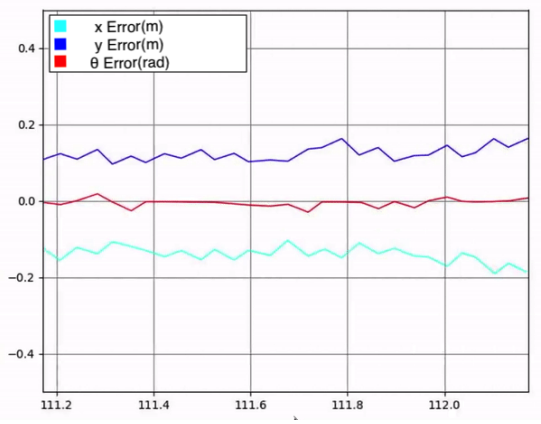


Figure 11: $x, y,$ and θ errors of the MCL algorithm being tested in simulation with the parking controller. The θ error is approximately 0 radians, the y-error approximately 0.1m, and the x-error approximately -0.1m. The relatively minimal errors in all three dimensions provide confidence in the abilities of our localization algorithm to be implemented on the physical race car next. (Figure Credit: Jessica)

3.2 Testing on the Car (Nihal & Jessica)

Two primary objectives were considered while testing the performance of our MCL algorithm on the car: 1. to tune our noise parameters using qualitative observations of performance in various testing environments, and 2. evaluate the success of our algorithm against justified criteria using both qualitative and quantitative measures. We first discuss the tuning of parameters before analyzing the two measures used to evaluate the algorithm's success: accuracy and convergence.

3.2.1 Tuning Parameters and Qualitative Analysis

First, the noise parameters were tuned on the car in order to ensure localization worked successfully on the physical race car. To do this, it was important to first understand how each parameter affected the performance of the particle filter in the real world. The car was tested repeatedly along the same path in the Stata basement, which is depicted in Figure 12. Through extensive testing, the following trade-offs were noticed for each tunable parameter:

- **Standard deviation of noise in the x-axis (σ_x)**: A higher value allows the car to self-correct its pose when noticeable features exist in front of it (in the car's x-direction). In scenarios such as long, featureless hallways, however, increasing this value may cause the filter to become stuck on a single significant feature, as the particles are largely spread out and continue to detect sensor scans at this particular feature. Reducing σ_x on the other hand causes the filter to trust the odometry data more. This is beneficial in long, featureless hallways as the particles can start to spread out in the x-axis; however, this may make it more difficult for the car to self-correct its pose in more complex, feature-rich environments.

- **Standard deviation of noise in the y-axis (σ_y)**: The car is able to localize and correct itself better along the width of the hallway it is driving down for larger values of σ_y. Increasing σ_y too far however can be detrimental, as the algorithm begins to devalue odometry inputs. A balanced reliance on both odometry and LiDAR data is needed for the localization algorithm to work effectively in all environments. As this value was decreased, the filter struggled to recognize turns into new hallways and was not accurate in localizing along the width of the hallway.

- **Standard deviation of noise in θ (σ_θ)**: Largely increasing this value causes there to be too many particles with different θ headings. Since the angle average is not weighted by probability, this skews the accuracy of the pose. Conversely, largely decreasing this value meant that the heading angle would not be tracked well while inputting a maximum steering angle input. This results in an incorrect pose during maximum steering angles.

- **Number of particles**: Increasing the number of particles allows for the averaged pose estimate to be more accurate; however, it greatly increases the computation time and thereby the publishing rate. Reducing the number of particles creates a larger error in the pose estimate because the filter is unable to explore more potential race car positions. As previously stated, this is traded off with decreasing the computation time and thus a faster publish rate.

- **Squashing Factor**: The squashing factor is a parameter that is used to flatten the sensor model output probability. Increasing the squashing factor causes the probability distribution to be more peaked and leads to the filter becoming stuck on a certain feature. Reducing the squashing factor leads to a less peaked distribution causing the filter to find it difficult to localize itself and the estimated pose to jump between different positions on the map.

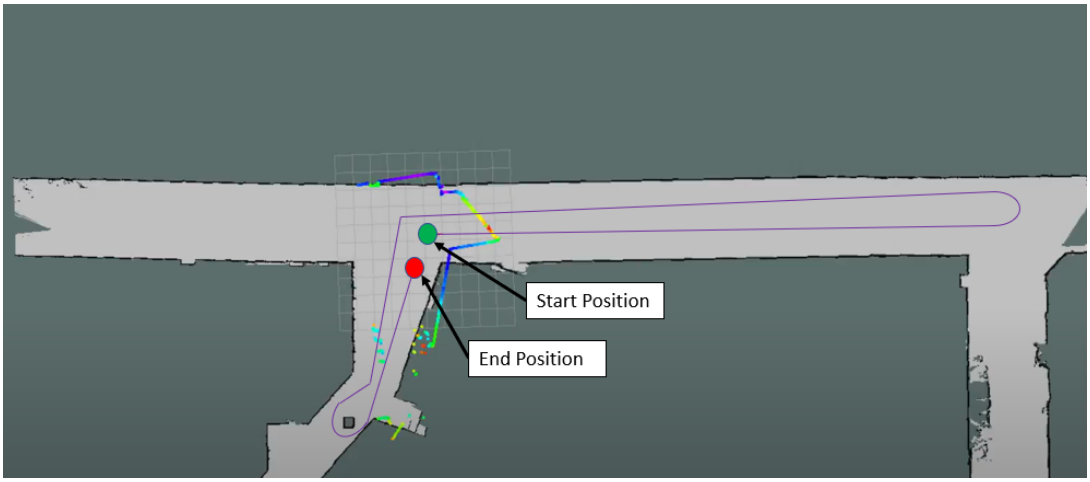


Figure 12: Map depicting the experimental path used for parameter tuning. The start position represents the origin of the map frame and is a convenient location to start the test as it allows us to test in a relatively difficult section of the Stata basement. The particles are initialized via a Gaussian distribution centered around this pose with larger-than-usual standard deviation values. (Figure Credit: Nihal)

Armed with these understandings, parameters were individually tuned in order to provide the best possible performance, balancing the aforementioned trade-offs. The noise

parameters chosen were $\sigma_x = 0.01$ m, $\sigma_y = 0.05$ m, and $\sigma_\theta = 0.04$ rad. These values allowed for the odometry to be trusted more in the x-axis yet rely on wider particle spreads in the y-axis and θ directions. Ultimately, this allowed our car to be able to localize itself at higher-speeds and account for various turns well. A squashing factor of 0.28 was chosen as the optimal value for an optimal distribution. Finally, 300 was chosen as the number of particles. This was the maximum number of particles that could be used without the publish rate falling below 20 Hz, a frequency sufficient enough for the sensor model to be able to accurately correct the odometry.

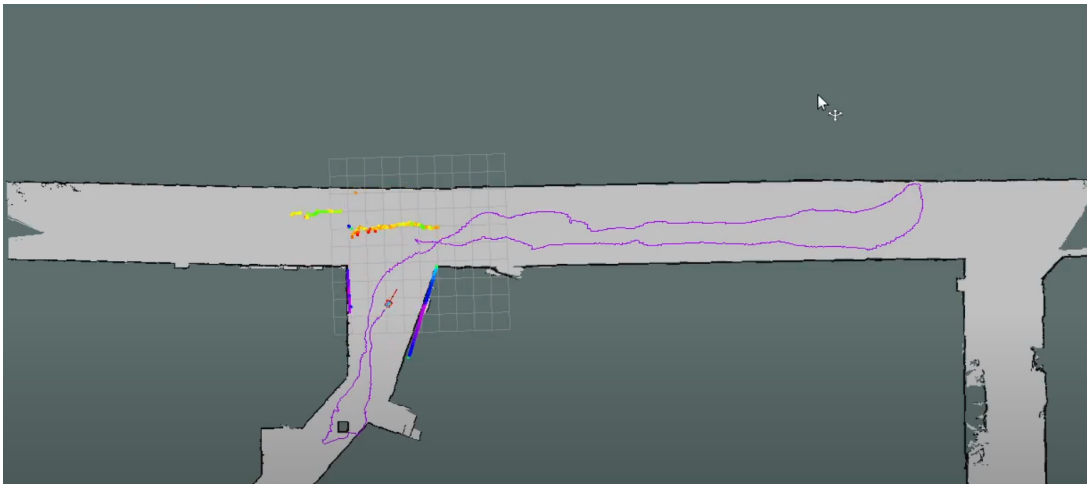


Figure 13: The estimated path obtained via the localization algorithm while running the same path as in Figure 12. The purple path represents the trajectory of all previously estimated poses and the red arrow shows the car's current odometry data. The jagged nature of the line is due to challenges in driving the car manually in a smooth path. Noticeably, the localization algorithm incorrectly perceives the car's 180 degree turn (at the right-most intersection) earlier than when the turn was actually executed in the real world (which was in the middle of the hallway intersection). We also hypothesize this is due to the amount of boxes and clutter that was present in the Stata hallway that was not depicted in our given map. Furthermore, the localized path demonstrates that the car is also able to handle large steering angle inputs without losing sight of correct heading angle. (Figure Credit: Nihal)

Using the aforementioned parameters resulted in the localized trajectory shown in Figure 13. In general, this trajectory coincides with the used testing path quite well. The car demonstrates its robust ability to continue localizing itself on the map through various obstacles in its path. The only region where our particle filter qualitatively fails is towards the end of the right-most hallway intersection. Ideally, the localized path should depict the car executing the 180 degree turn in the middle of the hallway intersection, as that is what was manually commanded to it during the test path; however, the localization path obtained shows the car executing the turn much earlier than intended. We hypothesize this may be due to a lack of particles which limits the spread of possible poses the sensor model is able to use to correct and reduce the error on the pose estimate.

3.2.2 Particle Filter Accuracy

To quantify the accuracy of our localization algorithm, a test was conducted to compare the localized pose estimates to empirically-obtained poses in the real world when the car runs in a perfectly straight path. In Stata basement, we discovered a black line on the ground emanating from a spot on the ground that coincided with the origin point

of the map frame. Empirical measurements for the poses along the line could then be obtained relative to the map frame origin. In this test, the car is initialized at the map frame origin and driven along this straight black line. Accuracy can then be quantified by calculating the error between the estimated poses returned from the localization algorithm and the empirically-measured ground truth pose values. Figure 14 illustrates the testing environment.

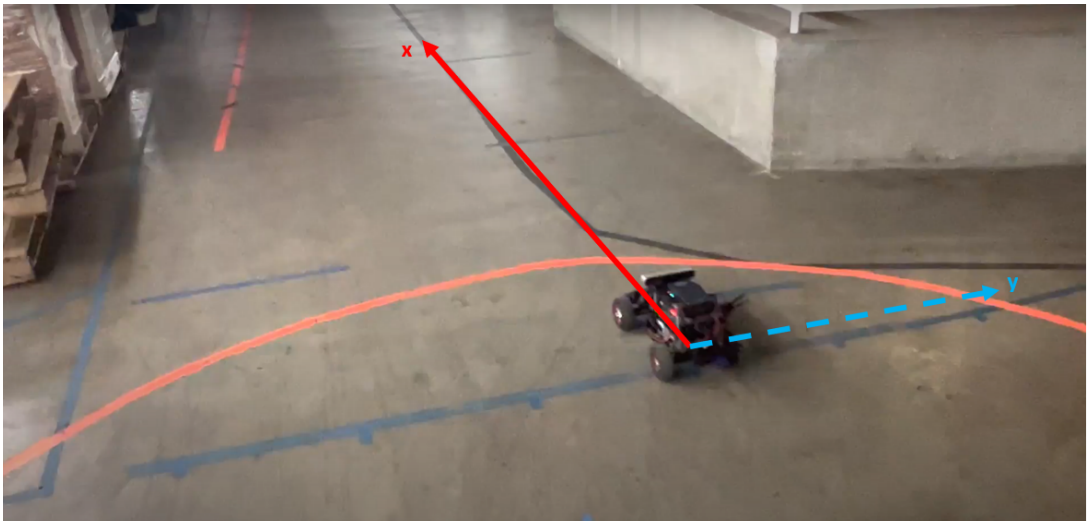


Figure 14: Experimental setup to quantify the accuracy of the localization algorithm. This setup takes advantage of a straight black line found on the ground emanating from the map frame's origin, for which it is easy to obtain empirical measurements relating to the ground-truth poses we should expect to see. The car is initialized at the map frame origin and travels along the line which is also simply the x-axis. The boxes to the left act as a proxy wall which aren't present in the map provided. (Figure Credit: Nihal)

Figure 15 shows the error between the estimated and the measured ground truth $x, y,$ and θ. The x error remains at around 30 cm throughout the test which can be attributed to the test being conducted along a long, featureless hallway. For the y error, the error seems exhibit more oscillatory behavior. This is perhaps due to the presence of obstacles such as boxes along the left side of the hallway, which can be seen in Figure 14. These features are not present in the provided map and caused the filter to think that the boxes/clutter constituted the left wall when in reality they weren't. Finally, the error along the heading direction maintained consistent around 0.1 rad. This confirmed the accuracy of the heading angle just as it was seen previously in the qualitative testing of the localization algorithm.

3.2.3 Convergence

Convergence is another important metric used to evaluate the success of our MCL algorithm. Ideally, as MCL runs, particles should converge closer together around the actual ground-truth pose. To test convergence, the same experimental setup as previously used for the accuracy testing was used. The car is run along the same straight line and the standard deviation of the particles is measured. Figure 16 illustrates the principle behind using the standard deviation as the convergence metric, as it can quantify both how long and to what radius around the car the particles generally converge to.

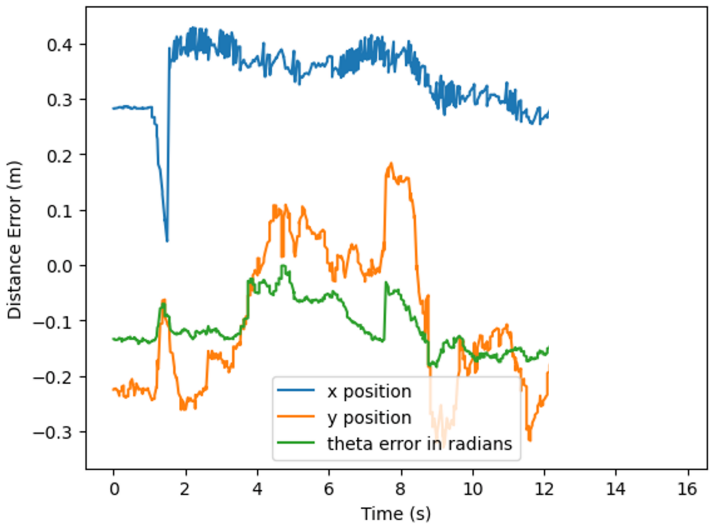


Figure 15: Plotting the x, y, and θ errors in the estimates provided by the particle filter. The x error is larger than y and θ due to the long, featureless hallway along which this test was conducted. (Figure Credit: Shara)

The particles are first converted from Cartesian to polar coordinates to provide the radial distance values of each particle relative to the base link of the car. The standard deviation of these values are then taken and plotted over the duration of the testing path, as shown in Figure 17. It can be seen that the standard deviation spread of the particles somewhat converges to a relatively steady-state value of 15 cm as it continues on the path. We interpret 15 cm as an excellent threshold for convergence, as 15 cm is half the width dimension of the car. Statistically speaking via the empirical 68-95-99.7 rule, this means that approximately 68% (a majority) of particles are within the car's width; in other words, the particles are able to converge tightly enough to provide a sufficiently accurate pose estimate. This serves as an important implication as we move towards motion planning, where our localization's ability to converge and obtain relatively accurate pose estimates will be essential in protecting against object collisions at high speeds.

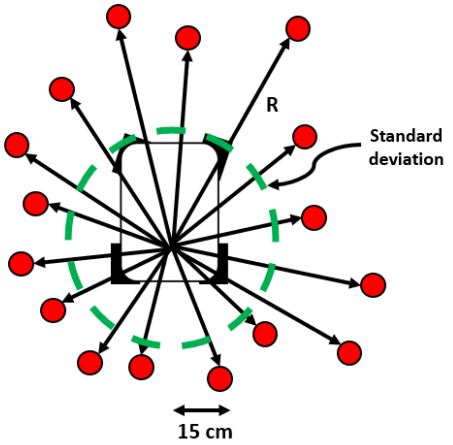


Figure 16: Graphical representation of the standard deviation calculation metric for convergence. The standard deviation represents a circle around the base link of the car within which a majority of the particles are most likely to fall. It is important for this radius to be below 15 cm as this is the width of the car. Having a tighter convergence of particles leads to having pose estimates closer inside the car's frame, means that the car can navigate using motion planning algorithms without needing to worry about hitting a wall due to an inaccurate pose estimate. (Figure Credit: Nihal)

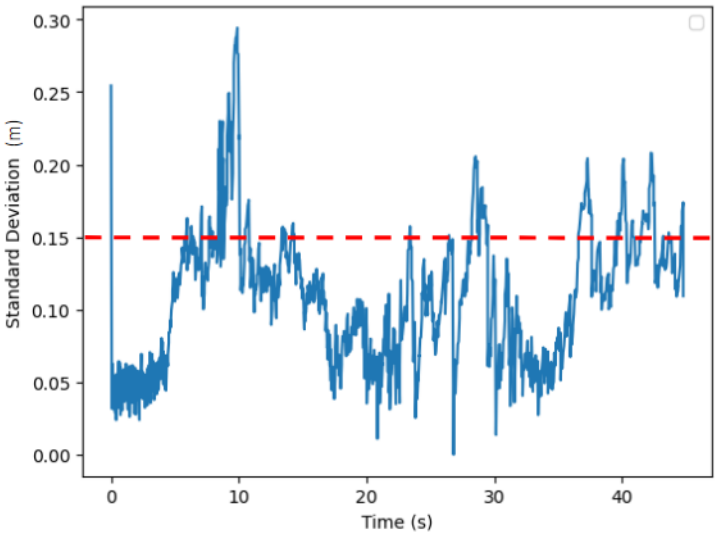


Figure 17: Graph plotting the standard deviation of particle distances toe the base link of the car in the straight-line test. After initialization, the particles only take a few milliseconds to converge to a 5 cm radius around the car which is very quick. As the car continues to drive, usual convergence rates of importance are on the order of 1-5 seconds as can be seen from when the peaks fall to the steady state value of around 15 cm. (Figure Credit: Shara)

4 Simultaneous Localization and Mapping (Shara)

Simultaneous localization and mapping (SLAM) is a technique that enables robots to use sensor data to create a map of an unknown environment while also using localization at the same time to figure out where it is on the created map. SLAM helps to solve common problems faced in autonomous systems by providing the necessary framework for these systems to navigate changing environments autonomously. As covered in

the analysis of our MCL algorithm, localization is a key part of autonomous navigation because a robot's awareness of its position and orientation enables it to successfully complete location-dependent tasks. We go beyond the lab and build on the importance of localization to implement SLAM, where the accuracy of localization is further essential to construct an accurate map of the environment.

To implement SLAM using our localization algorithm in simulation, we use Google's Cartographer system. Cartographer helped to visualize and create the map of the area our robot traversed while our localization algorithm was running. The accuracy and layout of the map is largely dependent on the performance of our localization algorithm.

A primary challenge faced in evaluating our SLAM algorithm was determining the exact transformations between the map frame and the cartographer map frame. At first, the transformation between the two frames was not being published properly. Although a relatively accurate map was being formed as our robot was traveling, the robot was initially unable to be located on the newly formed environment as seen in Figure 18.

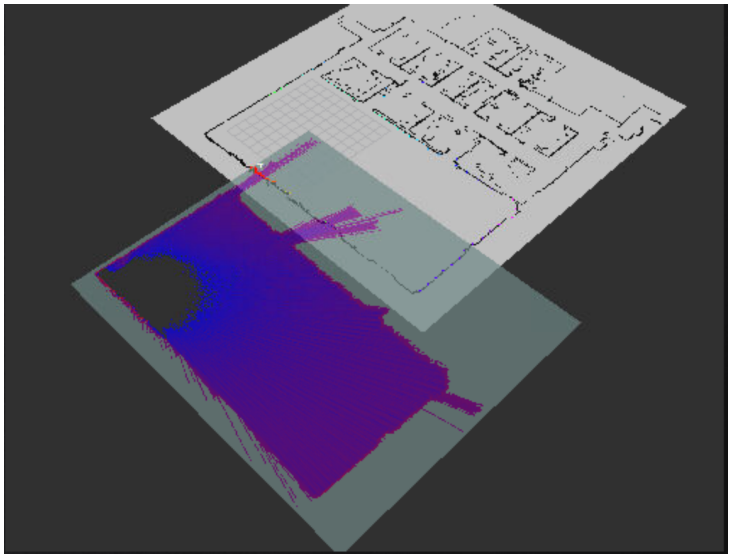


Figure 18: Initially constructed cartographer map via SLAM with incorrect transform. This constructed map correctly shows the area in which the robot has traversed but the robot does not also show up on the constructed map. This is because erroneous transformations between the cartographer map frame and the normal map frame were used at this time. The incorrect transformation prevented the robot from knowing where it is on the constructed map.(Figure Credit: Shara)

By investigating the Transformation Tree ("tf-tree"), the correct transformation is obtained which successfully localizes the robot on the newly-constructed map, as shown in Figure 19. Notably, however, the orientation of the map changes as the car continues to drive, despite the robot appearing in the correct position on the constructed map relative to where it is actually driving. This constant adjustment is largely due to the fact that the map is being created as the car travels. Cartographer simply projects the most probable map representation given the data seen so far at that current update step. As the robot traverses the area multiple times, SLAM is more confident in determining the correct representation of the map. Additionally, SLAM's performance is largely dependent on the race car's ability to localize. A more robust and accurate particle filter would reduce the time needed for the constructed map to be in the correct orientation and have the

correct features. These challenges could perhaps be addressed via the implementation of map matching, leaving a possible avenue for future work to improve the quality of our SLAM algorithm.

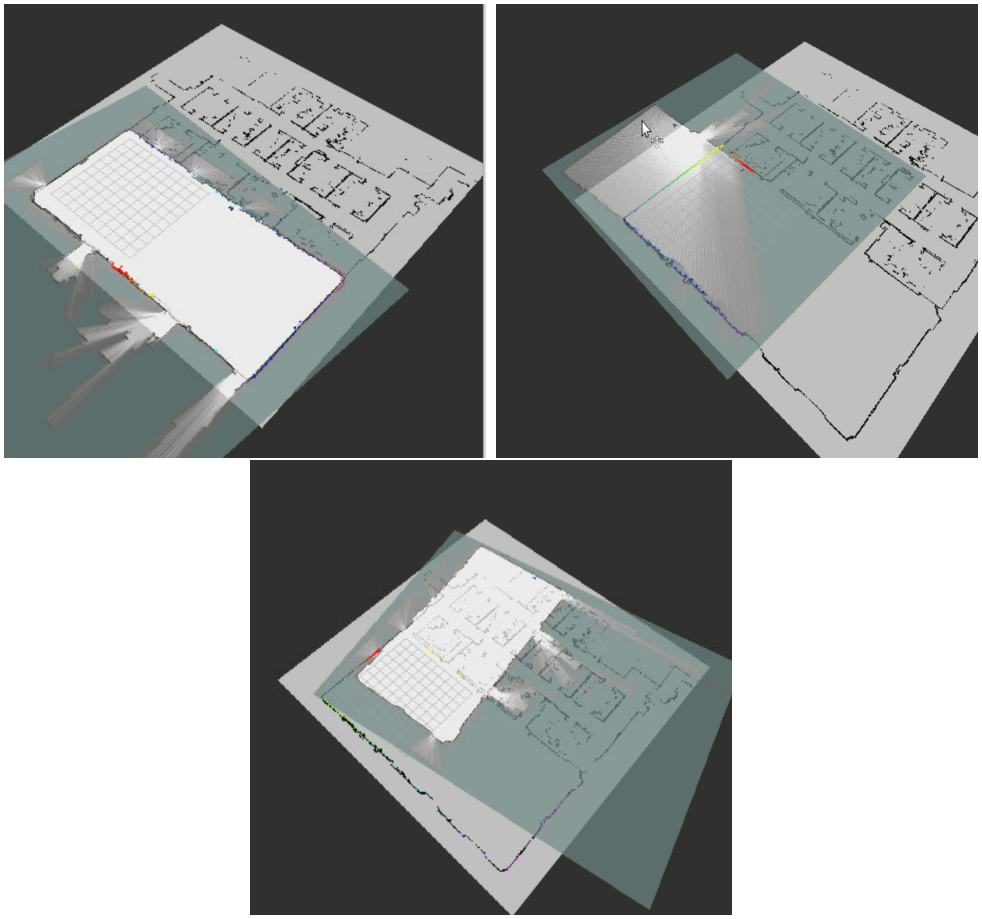


Figure 19: Illustration of various instances of SLAM working in simulation. While the constructed maps demonstrate the correct transformations between the cartographer and normal map frames, the orientation of the maps are different due to SLAM continuously updating the map as the robot travels. This may result in instances where there is a correct overlap of the maps but a flipped orientation (top-left), a correct orientation but incorrect overlap (top-right), or an incorrect orientation and overlap i.e. when the robot finishes turning a corner (bottom). In all such instances, however, the car's localization in the constructed map is relatively accurate. The blue car indicating the actual location of the robot on the constructed map in general matches up well with the robot's position on the actual map, determined by the colored laser scan data that appears on the underlying map (where red laser scan data indicates the wall that the car is positioned parallel to). (Figure Credit: Shara)

5 Conclusion (Shara / edited by Aaron)

In this lab, we successfully designed, implemented, and tested a module that allows our robot to estimate its current position and orientation. Various uncertainties present in the world had to be accounted for, such as noisy sensor data and a static map that does not account for environmental changes. To overcome such challenges, we implement our Monte Carlo Localization (MCL) algorithm consisting of three main components. The motion model updates the robot's current pose via odometry data. The sensor model evaluates the probability of the robot having a certain pose via LiDAR data. The particle filter then brings the predictions of these two models together utilizes particles as a wider

representation of possible poses that the robot could have.

Experimental data, both quantitative and qualitative, were first used to debug our algorithm in simulation before being eventually used to further tune and assess its abilities to localize in various real-world settings. From our testing procedure, we recognized key trade-offs related to computational speed and accuracy inherent with each parameter. In the end, we justify our choices for our parameters in demonstrating how the car can be successfully localized in various paths, ranging from a simple straight line to a long turn-filled route, with high accuracies and tight conversion rates.

In addition, we extended our implementation of localization to implement Simulatenous Localization and Mapping (SLAM), which furthers the ultimate goal of autonomous navigation by simultaneously creating a map of a dynamic environment as it is explored. We demonstrate our SLAM algorithm's ability to accurately localize the race car while it continuously constructs a map of its environment, while noting areas for future improvements via map matching.

Localization is a key component of autonomous navigation as it enables more complex tasks to be executed such as map making and path planning. As many tasks are location dependent and require robots to adapt correspondingly to dynamic environments, an accurate and fast localization algorithm is needed to ensure such autonomous tasks are executed with accuracy and safety. The goal of creating autonomous systems provides sufficient motivation in continuing to develop methods that tackle the problem of perception and being able to model the complex and uncertain world.

6 Lessons Learned

Aaron: On the technical side, I again continued to learn about how much noise and uncertainty truly exists in the world and why creating autonomous systems in the real world is such a complex task. In simulation, we could logically follow our intuition in creating our algorithms, knowing that the "world would be perfect" in terms of measurements and error values. When putting things onto the race car, we witnessed how much uncertainty in our algorithm calculations are added with every step. Small sources of uncertainty from the noise inherent in sensor measurements to the boxes being in the hallways of Stata can compound quickly and significantly impact the performance of any autonomous system. While probability introduces a novel way to begin counteracting such phenomenon, this taught me to be more aware of possible sources of uncertainty – they may arise from sources least expected. On the communication/organizational side, the structure of this lab emphasized how important it is to be on the same page and keep pace with the rest of the group. Especially with spring breaks occurring during this project, group members often advanced the project independently in different areas. In order for each member to be able to contribute effectively in the end, we had to make sure that we had a good understanding of each other's work. It is important to also allocate sufficient time during group projects to educate others and ensure the group is up to speed with the project's direction. This "push-and-retract" idea of how we advanced with our project required frequent communication and built-in sessions to our group meetings designed

specifically to ensure everyone had a clear understanding of each concept addressed so far.

Jess: I learned the importance of testing in different environments. Originally, we were only testing in a hallway where there were lots of boxes which deviated from the map, causing the sensor model to be unreliable. We, therefore, tuned our values to largely reduce the noise and rely mainly on the odometry data. However, in areas that were more represented by the map, the sensor data was much more reliable and much more effective. Therefore, by testing in different environments, we had to find tuned values for noise that worked for both situations. Additionally, because of spring break, some team members continued to work while others were unavailable. I learned how important it was to post updates on our shared document and ask questions on the group chat. This way ideas could be bounced off other team members and everyone was up to date and ready to work when they returned.

Chuyue: I learned the process of parameter tuning, and understanding why algorithms might fail in complex environments. During my teammates' spring break, I also learned how to work with the car independently as well as communicate important details with my teammates because Harvard was not on break. I tuned the algorithm parameters in simulation to get it working, and it turned out that it didn't work quite well in the real world at first. So my teammates and I tried to tune the parameters in different environments. One thing that I think I could improve is about tuning the parameters in the sensor model. Although the model takes into consideration of faraway points (which occur quite often when we are driving down the hallway) and unknown obstacles (like boxes stacked in one hallway), it did not concur with these problems perfectly. I tried to tune them but did not get enough results due to the time limit.

Nihal: I learned about the importance of tuning parameters and understanding how different variables interact with each other. This was especially crucial as this lab has a lot of variables which can affect the precise working of the particle filter such as the noise standard deviation, the number of particles etc. Furthermore, developing testing procedures to quantitatively assess the filter needed thorough thinking and planning so being able to figure stuff like this out earlier would help in a big way. In terms of communication, I learned the importance of having to communicate difficult concepts that I understand well but others in the team might not in order to get them on the same page as me. Furthermore, I learnt the importance of keeping constant communication even though not everyone was working during spring break to be able to bounce off ideas better.

Shara: This lab made me realize how complex making an autonomous system is, there are so many environmental factors that impact performance of these systems and there isn't precise method of guaranteeing that you are recreating and considering environment factors properly. There is a lot of probability and tuning involved in attempting to "perfect" these systems and that although your system may be working well in one environment, it may not do the same in another. This lab also taught me more about how important it is to extensively test and test in different environments. Sometimes the algorithm would work very well at shorter distances but once we arrive in a different part of the environment that may have unexpected features the behavior of the algorithm changes. I also was able to work on SLAM towards the end of this lab and I found it really cool how we could dynamically account for changes in our environment, this

was very interesting after having worked with a static map for the other parts of the lab. Additionally, this lab also fell partially during spring break and it taught me more lessons on communication and asynchronous working because different people had to pick up where others left off and we had to learn to communicate technical details and problems and solutions we have faced. It was also sometimes challenging to speak to each other about some issues without having access or experiencing the issues firsthand, some of the issues and errors had to do with optimizing and the code really transformed a lot as we attempted to optimize. Github was a really nice tool for this asynchronous work because we could track and view changes and catch up on things as we changed a lot of code rapidly.

References

[1] Dieter Fox Sebastian Thrun, Wolfram Burgard. Probabilistic robotics, 2005. URL https://docs.ufpr.br/~danielsantos/ProbabilisticRobotics.pdf.