Lab #5 Report: Monte Carlo Localization (MCL)

Team #9

Karen Guo
Yohan Guyomard
Michael Kraus
Jaclyn Thi
Carlos Villa

6.4200/16.405 - Robotics: Science Systems

April 8th, 2024

1 Introduction (Carlos Villa and Karen Guo)

One of the most fundamental components for autonomous vehicles is localization; a robot must know its position and pose to be able to perform and plan tasks accurately and efficiently. When given a map of its environment, a robot is expected to deduce its location based on odometry measurements, its estimated previous position, and probability calculations. Through simultaneously running mapping (SLAM), autonomous vehicles can map unknown or dynamic environments. Common uses include using SLAM within home cleaning robots, like the Roomba, allowing for easy and efficient navigation and cleaning. Aside from mapping, localization can be used with a variety of projects. For example, Google Street View's localization creates immersive environments for virtual reality devices.

Our goal for this lab was to perform localization of our racecar using the Monte Carlo Localization (MCL) algorithm. Creating an internal representation of our external world and estimating our location in it will allow us to more accurately perform motion and path planning and fluid autonomous driving in future labs. Combined with a wall follower or line follower from previous labs, we can navigate a known environment, like Stata Basement, to complete tasks autonomously. We decided to use the MCL algorithm over the Kalman Localization algorithm. Kalman Localization assumes our system is Gaussian and linear, however our robot system, and most systems in robotics are nonlinear. Hence, MCL is advantageous because it can represent nonlinear systems with

its particle filter structure.

To approach this problem, we split our localization into three main parts. We have a motion model, a sensor model, and a particle filter. The respective parts are responsible for updating the robot position, updating the particle probabilities, and initializing and resampling particles. Using perception methods we have already become familiar with in previous labs, our racecar's LIDAR, state information, and odometry, we localize our robot. Taking in the odometry data, our robot will update its belief of its possible positions. With a map of its environment as a base reference and a collection of possible positions, the robot uses the LIDAR and odometry data to estimate which of these possible positions are the most probable. Then the possible positions are resampled with new probabilities influenced by our measurements. After some time, the algorithm converges to a most probable position and our robot localizes itself. Research on Monte Carlo Localization as seen in this lab aims to improve the precision of robotic navigation through probabilistic techniques, supporting the development of efficient and reliable autonomous systems.

Since we were not working with SLAM, we assumed that we would given a map of the environment. We also assume that the robot only depends on it's previous state, which allows us to use Monte-Carlo probabilistic models.

2 Technical Approach

2.1 Motion Model (Jaclyn Thi)

The motion model takes in the current odometry and a set of particles representing pose estimates for the robot. It then applies the odometry to each of the particles to generate updated position estimates based on the motion of the robot, returning a set of particles representing updated position estimates. This section goes into the implementation of the motion model as well as the design decisions that went into the implementation.

2.1.1 Implementing the Deterministic Motion Model

Our motion model f accepts a previous pose prediction x_{k-1} in the world frame and Δx in the robot frame and returns a new pose prediction x_k, which has been updated with the odometry information:

$$x_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = f(x_{k-1}, \Delta x)$$

The odometry of the robot (given with respect to the robot frame) is represented by

$$\Delta x = \begin{bmatrix} \delta x \\ \delta y \\ \delta \theta \end{bmatrix}$$

Given a set of N particles representing pose estimates in the world frame, each with previous pose $x_{k-1} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix}$, as well as odometry Δx as defined previously, we can update the position of each particle using the odometry using

$$T^{-1}\Delta x + x_{k-1} = x_k$$

which transforms the odometry vector from the robot frame to the world frame and adds it to the particle's previous pose vector. T is the transformation matrix from the world frame to the robot frame and is found using

$$T = \begin{bmatrix} cos(-\theta_{k-1}) & -sin(-\theta_{k-1}) & 0 \\ sin(-\theta_{k-1}) & cos(-\theta_{k-1}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ_{k-1} is taken from the particle's previous pose. This is the robot's rotation in the world frame, and thus $-\theta_{k-1}$ specifies the rotation needed to shift the robot frame to the world frame.

We apply this update to a matrix of N particles of the form

$$\begin{bmatrix} x_0 & y_0 & \theta_0 \\ x_1 & y_1 & \theta_1 \\ & ... & \\ x_N & y_N & \theta_N \end{bmatrix}$$

where each row represents one particle with pose x_{k-1}, and return the matrix of particles with updated poses from the provided odometry.

2.1.2 Injecting Random Odometry Noise

In order to account for any uncertainty in the odometry information, we need to inject random noise into our motion model function. The random noise is added to the odometry Δx takes the form

$$\epsilon = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_\theta \end{bmatrix}$$

Then our motion model with noise would be $T^{-1}(\Delta x + \epsilon) + x_{k-1} += x_k$.

We initially attempted to calculate the random noise to be proportional to the $\delta x, \delta y, \delta \theta$ from Δx using the following:

$$\epsilon = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_\theta \end{bmatrix} = \begin{bmatrix} sample(\alpha_1|\delta x| + \alpha_2|\delta\theta|) \\ sample(\alpha_1|\delta y| + \alpha_2|\delta\theta|) \\ sample(\alpha_3|\delta\theta| + \alpha_4|\delta x + \delta y|) \end{bmatrix}$$

Where $sample(b)$ draws a random number from a normal distribution with variance b. The reasoning for this was that the larger the odometry values for $\delta x, \delta y, \delta\theta$, the greater the potential error associated with the odometry measurements and the greater the potential noise. $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are coefficients that could be tuned.

However, this implementation for generating random noise proved to be ineffective, as our robot's estimated position began diverging wildly in simulation when used in our particle filter.

For our second implementation of generating random noise, we opted to use the following to add random noise to the odometry:

```
particles += max(0.75 * abs(odometry[2]), 0.0015) *
    np.random.normal(size=particles.shape)
```

where *particles* is the matrix of pose estimate particles, *odometry*[2] is the $\delta\theta$ from the odometry. This code uses the magnitude of the $\delta\theta$ from odometry times 0.75 (or 0.0015 if this value is sufficiently small), and multiplies that value with a random sample from a Gaussian distribution. Using this new noise allowed our robot's position estimates to track the robot position much better.

2.2 Sensor Model (Michael Kraus)

Once the odometry data is calculated and the position of all the particles is updated the sensor model calculates the probability of each particle quickly and accurately. This section will describe the general implementation of the sensor model and the design choices that were made to balance the accuracy and quickness of the algorithm.

2.2.1 Precomputation (calculating the probability of a single lidar measurement)

Since the sensor model will need to calculate the probability of thousands of lidar measurements repeatedly in a time-critical manner, a precomputed lookup table was implemented to eliminate the need to perform complex calculations each time the position of the particles updates.
When computing the probability of a lidar reading there are four cases that need to be considered:

1. The probability of detecting a known obstacle. This is given below:

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

where η is a normalizing constant, z_k is the measured distance, and d is the true distance.

2. The probability of a short measurement. This is given by:

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

3. The probability of a large (missed) measurement. This is given by:

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

4. the probability of a completely random measurement. This is given by:

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

where z_{max} is the maximum range that can be returned by the lidar scanner.

Then, once each of these individual probabilities is calculated the total probability of the lidar measurement can be found with the following equation:

$$p(z_k^{(i)}|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m) + \alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m)$$

Where $\alpha_{hit}, \alpha_{short}, \alpha_{max}$, and α_{rand} are chosen parameters and total to one. Figure 1 Shows a visualization of this look-up table using $\alpha_{hit} = 0.74$ $\alpha_{short} = 0.07$ $\alpha_{max} = 0.07$ $\alpha_{rand} = 0.12$ and $\sigma = 8$. These were the default values provided by the teaching staff, and they gave our sensor model and particle filter adequate performance. Due to time constraints and debugging issues in other parts of our project, we did not adjust these parameters.

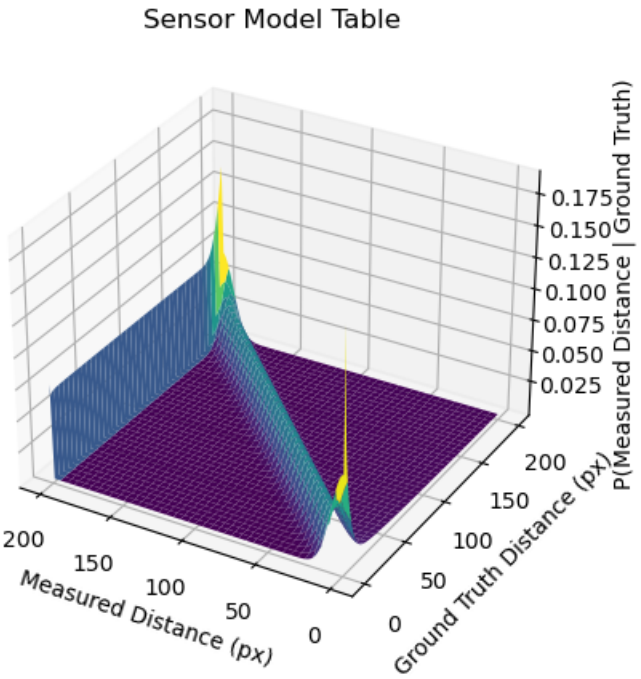

Figure 1: If the Ground Truth Distance (d) and measured distance z_k are known the the precomputed lookup table allows the probability of individual particles to be computed quickly. The height of the surface represents the probability of a particle for a given measured distance and a ground truth distance.

To help minimize runtime and keep the code clean, a fully vectorized (as opposed to using explicit loops) approach was used when writing the code. This is not critical because this function is only called once. However, it is still a good practice and keeps the code clean. The code for the precomputation is shown below:

```
def precompute_sensor_model(self):

        alpha_hit = self.alpha_hit
        alpha_short = self.alpha_short
        alpha_max = self.alpha_max
        alpha_rand = self.alpha_rand
        sigma_hit = self.sigma_hit
        eta = 1


        z_max = self.table_width-1 #assumes that the
            longest dist (in px) is the last entry in the
            table
        d_array = np.linspace(0.000001,z_max,self.
            table_width)
        z_array = np.linspace(0.000001,z_max,self.
            table_width)

        # Compute raw p_hit
        diff_squared = (z_array[:, np.newaxis] - d_array[
            np.newaxis, :]) ** 2
        exponent = - diff_squared / (2 * sigma_hit**2)
        constant = eta * 1 / (2 * math.pi * sigma_hit**2)
            **0.5
        p_hit = constant * np.exp(exponent)
        #normalize p_hit
        column_sums = np.sum(p_hit, axis=0)
        p_hit_normalized = p_hit / column_sums[np.newaxis
            , :]
        p_hit = p_hit_normalized

        #compute p_short
        constant = 2/d_array[np.newaxis,:]
        multiplier = np.where(d_array[np.newaxis,:]>
            z_array[:,np.newaxis],1-z_array[:,np.newaxis]/
            d_array[np.newaxis,:],0)
        p_short = constant*multiplier

        #compute p_max
        p_max = np.where(z_array[:,np.newaxis]==z_max
```

```
    ,1,0)

#compute p_rand
p_rand = np.full((self.table_width,self.
    table_width),1/z_max)

#compute total probability
p_table = alpha_hit*p_hit+alpha_max*p_max+
    alpha_rand*p_rand+alpha_short*p_short
#normalize the p_table by d value (columns i
    think)
total_column_sums = np.sum(p_table, axis=0)
p_table_normalized = p_table / total_column_sums[
    np.newaxis, :]
self.sensor_model_table = p_table_normalized
```

2.2.2 Evaluation (calculating the total probability of a particle)

Now that we have a table that we can use to look up the probability of a given lidar measurement, we can determine the probability of a given *particle*. That is, each particle will have several measurements, all with their own probability. We need to somehow total these probabilities to calculate the total particel probability. This is a reasonably simple task, however there are a few important challenges that need to be considered. To understand these challenges one must first understand how the probability of a particle is calculated.

To begin, the particle must be passed into a ray-casting algorithm. This algorithm uses the particle's pose and calculates a "mock" lidar scan (it's essentially just an array of ranges) using the given map, as shown in Figure 2. These measurements are essentially the true distance (d) that we used when calculating the probability of a given lidar measurement.

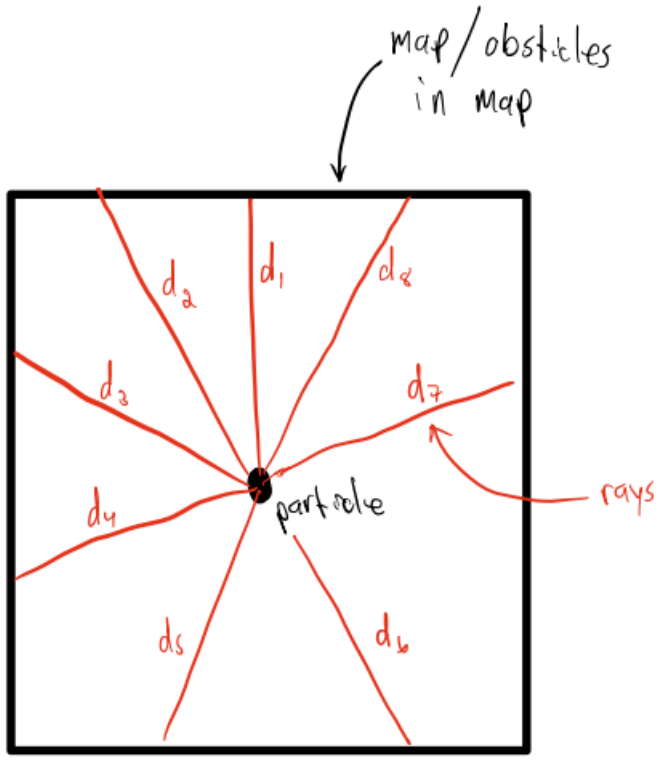

Figure 2: Essentially just a simulated lidar scan, ray casting will determine the true distance from the particle to objects in the map. Then these true distances can be compared distances that the robot's lidar scanner measures to determine the probability that the particle is the true position of the robot.

Then, using each of these true distances and a corresponding lidar scan (measurement) from the robot, we can use the lookup table to calculate the probability of each lidar measurement made by the robot.

Once we calculate the probability of each of the individual measurements we can calculate the total probability of the particle with the following formula:

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

This process is repeated for every particle.

One challenge with this process is ensuring that the ray-casting algorithm produces the same number of rays as there are lidar measurements in the lidar scan. One option to solve this issue is to write the ray-casting algorithm in such a manner that it produces the same number of rays that the lidar scanner measures. However, this is completely unnecessary; the lidar scanner takes thousands of measurements per scan, many of which are redundant. Thus, to help minimize the runtime of the code, it is far better to simply down-sample the number of lidar scans. That is, while our lidar scanner takes 1,000 measurements per scan, our ray-casting algorithm only generates 200 rays. Thus, we only use every fifth lidar measurement (for a total of 200) when we calculate the probability of a particle.

In addition to this design choice, we also wrote the code for the lidar evaluation function in a completely vectorized manner. As with the precomputed table, this produces cleaner, faster-running code. While explicit for loops are often easier for the novice programmer to read, the evaluate function of the sensor model is called repeatedly and needs to run quickly to ensure smooth and accurate results. Thus, despite being less intuitive to look at and debug, vectorized methods were used because the runtime of this function is absolutely critical. The code for the evaluate function is provided below:

```
def evaluate(self, particles, observation):
    if not self.map_set:
        return

    particle_scans = self.scan_sim.scan(particles)

    #convert scans from meters to pixels and round to
        the nearest int
    particle_scans_px = particle_scans/(self.
        resolution*self.lidar_scale_to_map_scale)
    particle_scans_px = np.round(particle_scans_px)
```

```
particle_scans_px = particle_scans_px.astype(int)
particle_scans_px = np.clip(particle_scans_px,0,(
    self.table_width-1))

#downsample the lidar
stride = max(1, observation.shape[0] //
    particle_scans_px.shape[1])

# Slice the array to select every `stride`-th
    element
downsmpled_observation = observation[::stride]

#convert the downsampled observation to px and
    round
px_observation = downsmpled_observation/(self.
    resolution*self.lidar_scale_to_map_scale)
px_observation = np.round(px_observation)
px_observation = px_observation.astype(int)
px_observation = np.clip(px_observation,0,(self.
    table_width-1))


#calculate the probability of each vector
particle_probs = self.sensor_model_table[
    px_observation, particle_scans_px]

#calc each particles probability
total_particle_probs = np.prod(particle_probs,
    axis=1)

return total_particle_probs
```

3 Particle Filter (Yohan Guyomard)

The purpose of the particle filter is to synthesize the motion and sensor models into a complete MCL algorithm. It maintains the collection of particles as an $N \times 3$ matrix where the three columns correspond to position (x, y) and orientation (θ). It is implemented as a ROS2 node with subscriptions to a source of odometry and laser scan data that is then fed to the motion and sensor models respectively. Finally, it averages the position and orientation of the particles and publishes the result; we expect the particles to converge to a single point which is the estimated location of the robot in space.

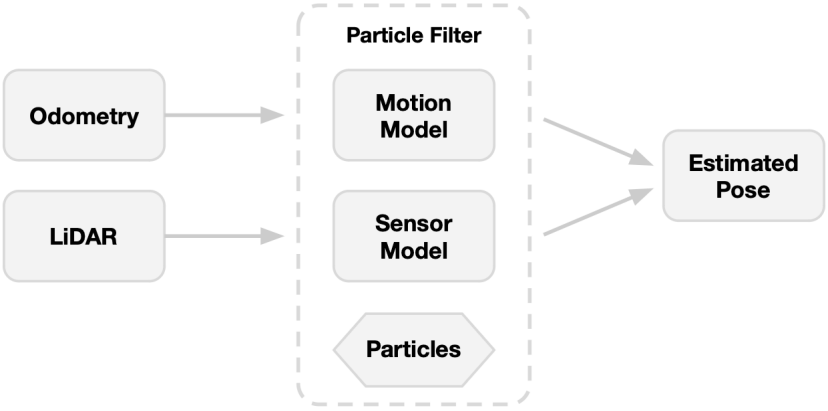

Figure 3: Overview of the Particle Filter in the context of ROS2 nodes.

3.1 Integration of the Motion Model

Built into our race car system (and simulation) is an odometry node. It uses the rotary encoders and known diameter of the car's wheels to estimate the instantaneous velocity and, using dead-reckoning, can estimate the car's location subject to some drift. Since dead-reckoning is generally unreliable and not needed for the MCL algorithm, we discard this estimation and depend only on the instantaneous velocity. Odometry packets are published relatively fast, so we compute the change in position $\Delta x = v_x \Delta t$ (likewise for Δy, $\Delta \theta$) where Δt is the difference in time stamps between this message and the one received immediately before it (msg.header.stamp). Next, Δx, Δy, $\Delta \theta$ are passed to the motion model where the particles' locations are updated. Whenever this happens, an updated estimation is also published as described in 3.3.

Unfortunately, the motion model has the same issue as dead-reckoning and will drift over time. Thus, we introduce the sensor model to correct these errors.

3.2 Integration of the Sensor Model

The sensor model casts rays in a simulated space and compares these results to rays cast in the real world. For these "ground-truth" rays, we are using a LiDAR scanner and the ROS2 LaserScan node. This publishes a vector of distances corresponding to each ray, which is then down-sampled and fed to the sensor model. The probabilities it returns are used to resample the particles, which works like this: a fresh new vector of particles is created (also $N \times 3$, as before) and each particle is populated by rolling a weighted die over the old particles' corresponding probabilities. That is, particles deemed "high probabality" (relative to the rest) by the sensor model will likely survive this resampling and likely be duplicated. Likewise, "low probability" particles will likely be dis-

carded. In practice, this is implemented using the numpy.random.choice utility.

This process is useful for eliminating particles which are unlikely to be at the same position as the robot, but over time every particle will converge (have the same x, y, θ) resulting in $N-1$ redundant particles (since they are all the same). In reality, this doesn't happen because the motion model introduces non-negligible noise which separates these duplicate particles.

Finally, we found that using the probabilities are returned by the sensor model gave undesirable results (inconsistent drifting over time). Our simple fix was the introduction of a **Temperature** parameter, where $p' = p^T$ for $T < 1$. This has the effect of flattening the peaks in the probability distribution and slowing down convergence, which yields better results.

As before, an updated estimation is published whenever the particles are filtered, as described in 3.3

3.3 From N Particles to 1 Location

We determine the estimated location of the robot by averaging every particle. For the position, this is straight forward and consists of taking the mean of x, y. Orientation is more subtle because, e.g. 360 deg and 0 deg would not have an arithmetic mean of 0 deg. Thus, the following formula is used:

$$\overline{\theta} = \arctan(\frac{\overline{\sin(p_\theta)}}{\overline{\cos(p_\theta)}})$$

Averaging in cartesian space gets rid of the issue. Finally, this is published as an Odometry message by our ROS2 node.

3.4 Race Conditions

The motion model, sensor model, visualization and publisher systems are all disjoint and may happen simultaneously. Since they all access or mutate the vector of particles, we utilize thread-safe code to guarantee exclusive-ownership over the particles. Using a Semaphore, only one model/system can act upon particles at a time and the others will stall until the lock is released. This approach eliminates race condition bugs where, e.g. particles are being moved as they are being filtered.

4 Experimental Evaluation (Karen Guo and Carlos Villa)

Our localization algorithm was tested multiple times with unit tests and simulation tests and graphs to ensure accuracy. For our experimental evaluation of the entire system, we focused on the convergence rate of our particles and

the absolute difference between our ground truth positioning and evaluated position. We decided on these metrics because they highlight the efficiency and accuracy of our algorithm. The absolute difference of our positioning versus the ground truth allows us to see how accurate our positioning is. By determining convergence rate we can identify how quickly our localization reaches an acceptable estimate and therefore when we expect our position estimate to be reliable. While we weren't able to perform tests in real life, we expect our evaluation methods to stay relatively similar, with some slight changes due to the fact that we do not have access to the ground truth.

4.1 Convergence rate

In simulation, we measured the convergence rate by determining how long it took the particles to converge to within a certain standard deviation of each other from their initial position. We chose an arbitrary threshold standard deviation of 0.07 meters since we felt that was sufficiently close in simulation and plotted that with respect to time. Additionally, we ran the tests for three cases: a stationary car, a car following a straight wall, and a car following a curved wall to see convergence rate for different cases.

From Figures 3, 4, and 5, we can see that as time passes the standard deviation lessens. This is within our expectations as our sensor model and particle filter should be filtering out less likely particles, which would in turn, decrease the distance between them. From Figure 3, we can see that the convergence rate is relatively fast compared to Figure 4, and Figure 5 (0.133 m/ms versus 0.024 m/ms and 0.014 m/ms), which would make sense since we expect lesser convergence time when the car is staying still. We can also see from Figure 4 and 5 and that the standard deviation decreases linearly, so if we wanted to estimate convergence, we could create a linear equation and solve for the expected time. While we ran out of time, if we were to run these tests real life, we would pick a higher threshold standard deviation, since there will be more noise within measurements. Otherwise, everything else for this test would stay the same.

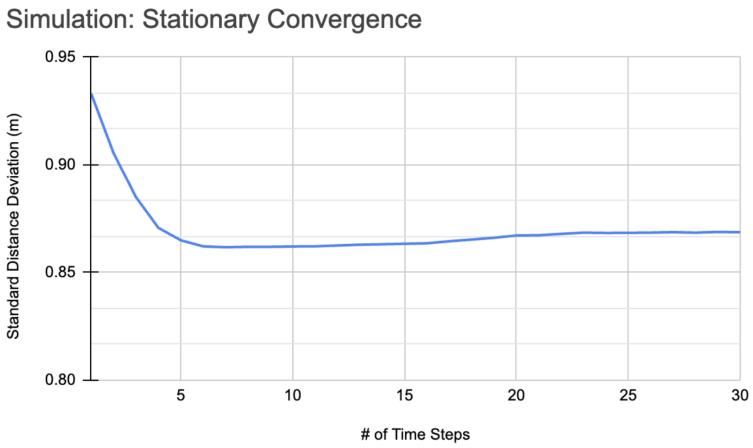

Figure 4: In this graph we plotted the standard distance deviation in meters versus the number of time steps where each time step is measured in a few milliseconds that have passed for a stationary car. We can see that the standard deviation decreases to around 0.86 in around 7 time steps before plateauing off. Thus our stationary car has a fast convergence rate at around $0.93/7 = 0.133$ m/ms (initial standard deviation over number of time steps).

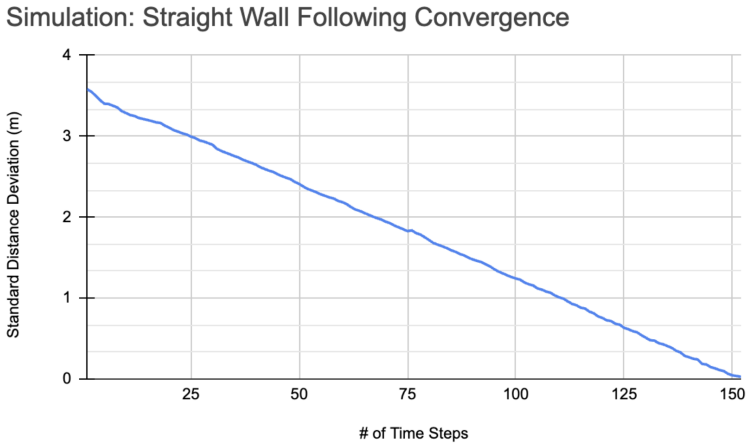

Figure 5: Likewise, we created a graph for when it is following a straight wall with the standard distance deviation in meters versus the number of time steps as axes. We can see that the standard deviation decreases to around 0 in around 150 time steps before plateauing off. Thus our stationary car has a convergence rate at around $3.6/7 = 0.024$ m/ms, which is slower than a stationary car, but faster than following a curved wall.

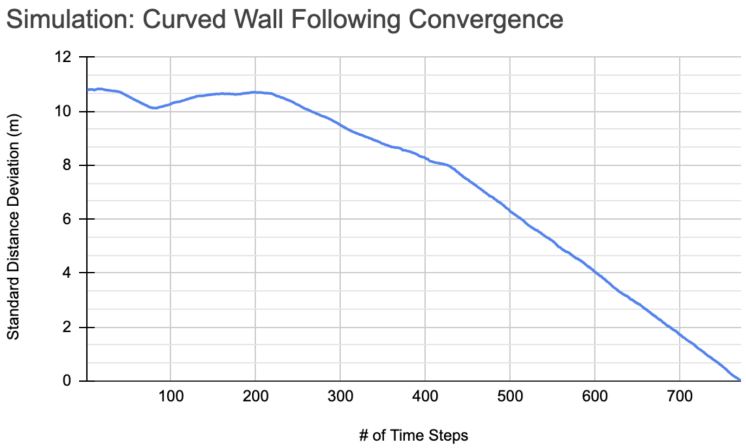

Figure 6: This graph displays standard distance deviation (m) versus the number of time steps for following a curved wall. The standard deviation decreases from around 10.8 to 0 in around 800 time steps. Thus our convergence rate is around 0.014 m/ms, the slowest of the three test cases.

4.2 Absolute Error

In addition to convergence rate, we compared the predicted x, y, and theta values with the ground truth while the racecar was stationary or moving to determine the accuracy of our evaluated odometry in simulation.

4.2.1 Simulation

We ran our MCL algorithm, then used RQT plot to plot our graphs for the x and y axes. Because the initialization in Rviz is slightly inaccurate, we change the initial pose and position of the car and take that section of the plot. We plotted the /pf/pose/odom and /odom topics to compare the truth to predicted. Figure 7 and 8 display the x and y positions respectively after moving the car. We can see that our estimated pose was almost the exact same of the ground truth. Of course, outside of simulation, we don't expect the graphs to look this accurate.

For predicted and true theta, we plotted the ground truth versus the predicted while the car was following a wall; hence we can see the angle change directions as it's moving. While slightly difficult to see, Figure 9 shows that our predicted angle was extremely accurate. From our three absolute error tests, we can see that our algorithm is accurate while stationary and moving, in position and angle.

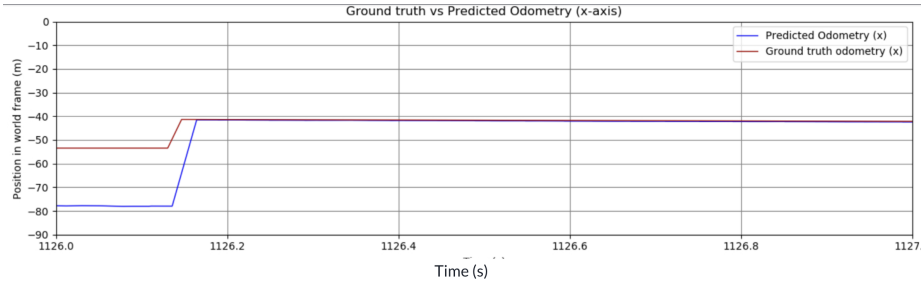

Figure 7: This graph displays the true position in x(m) in red, and the predicted position in x(m) in blue over time(s) for a stationary car. The initial positions before the sharp increase are after initialization in Rviz, hence are inaccurate. We can see that blue line is in the same place as the red line after 1126 seconds, meaning our model is accurate.

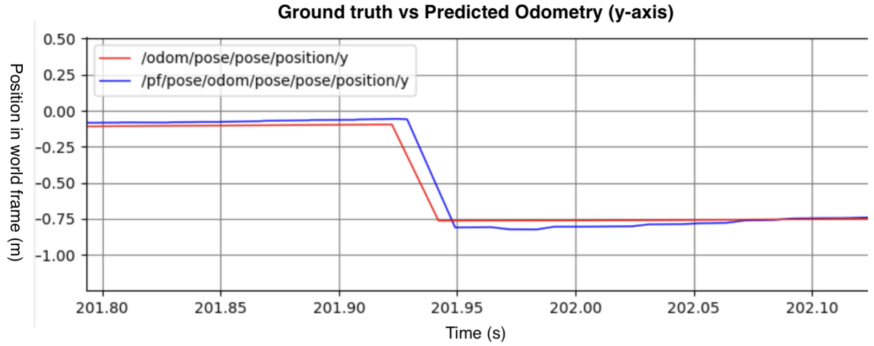

Figure 8: This graph displays the true position in y(m) in red, and the predicted position in y(m) in blue over time(s) for a stationary car. We can see that blue line is extremely close to the red line after 201 seconds, meaning our model is accurate.

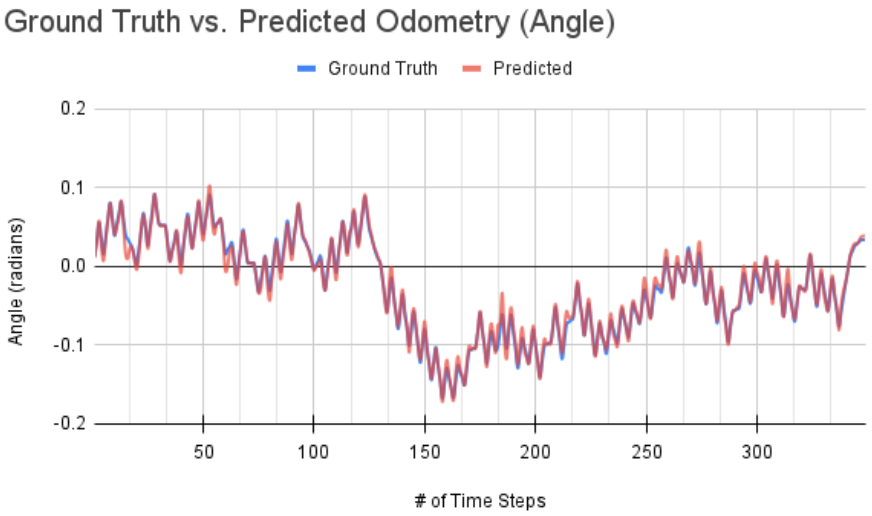

Figure 9: This graph compares the ground truth angle(radians) in blue to the predicted in red over the number of time steps, where each time step is a few milliseconds, for a moving car. The blue and red line have significant overlap, with only a few small differences, which means our angle prediction performs well.

4.2.2 Real Life

Due to time constraints we were unable to implement these tests in real life. However, if we were to run similar tests, we would have to alter the comparison to the ground truth. On the actual racecar, we don't have ground truth odometry readily available to us like in simulation. To compensate for the lack of ground truth information, our test cases would have to rely on easily measurable scenarios for comparison to predicted measurements. For instance, since we don't have accurate pose information we could instead perform tests from a fixed position to a wall and compare that distance to the one our algorithm computed. Other similar measures could be performed for evaluating x,y,theta. For convergence, we could have the same experimental setup since that doesn't require odometry information, just information about our particles.

4.3 Assessment

Coupling the two tests, we can see that our localization algorithm has a fast convergence rate, as well as accurate tracking. Our convergence rates in simulation all approach 0 over a short period of time, and our position predictions are almost completely the same as the ground truth. With both of these components we can assume that our algorithm is both accurate and efficient. Our preliminary trials in real life show that our model can accurately follow the direction of the car. However, we would need more rigorous testing in real life to determine whether it works as well as it does in simulation.

5 Conclusion (Michael Kraus)

Despite having difficulties getting our implementation working on the physical car, our team succeeded in creating an effective particle filter that works in simulation. In addition, our code has been written in an efficient manner that enables it to run quickly and smoothly, thus ensuring the reliable operation of the particle filter on the car.

Currently, the code has been uploaded to the car but still needs some minor debugging to get it working smoothly and producing usable data. Ideally, these minor tweaks should be all we need to get the car working, however, it is likely that we will need to tune the parameters used in the sensor model (which were untouched for this lab) and motion model to ensure **optimal** operation of the particle filter on the physical car.

In the next lab, we will be combining our localization implementation with a path-planning algorithm. As such it is extremely important that our localization algorithm can accurately represent the true location of the car so that our path

planning algorithms will generate meaningful, physically feasible paths (what's the point of generating a path if the car doesn't know where it is?) Although we did not accomplish everything we hoped to in this lab, we have managed to put our team in a manageable position and we feel confident moving forward.

6 Lessons Learned

Presents individually authored self-reflections on technical, communication, and collaboration lessons you have learned in the course of this lab.

6.1 Yohan

During this lab, we faced many challenges regarding scheduling and communications because I was sick during a good part of it. It is much harder to progress on the lab while stuck at home, however we managed to make things work by employing methods like peer-programming over Zoom, delegating real-world tasks to those that could make it, etc.

On a more technical note, we learned the absurd complexity of debugging a multi-stage system. When nothing worked, it proved difficult to isolate the problem and determine which model had issues. However, this is something we were able to do effectively by parallelizing which group member inspects and debugs which section of our codebase.

6.2 Michael

For the sensor model, we learned the "peakiness" of the look-up table can have some odd effects on the particle filter. Moreover, near the edges of the lookup table, there are some really high peaks that can cause the particle filter to favor particles that are far from the car. We had to smooth out the look-up table by raising all probabilities to the 0.4th power.

We also learned some lessons in relation to communication and planning. Mainly, we had to deal with some members getting sick (don't trust Maseeh dining) and had to make sure that we effectively shared knowledge - some people were more familiar with working on the robot and had to make sure that others had the resources and knowledge that they needed while they were sick or unavailable.

6.3 Jaclyn

We learned that the method for introducing random noise to our motion model based on odometry is important, as choosing an improper method for generating random noise caused our robot to perform poorly in simulation. Additionally, we had to learn how to communicate our availabilities and conflicts to find times to work on the robot together, since the semester was much busier after coming back from spring break.

6.4 Karen

Throughout the lab, we learned a lot both technically and communication wise. For the technical elements, we had some challenges getting the particle update to work correctly. Our team spent a lot of time debugging, so we all have a better understanding of how the different parts work. We were able to fix the problem by squashing peaks in the sensor model as well as changing how the noise in the motion model was implemented.

Communication wise, we had more trouble with scheduling meetings this week. Due to midterms, work, and sickness, a lot of us weren't free at the same time. Hence, we also had a meeting on Zoom so that everyone would be able to work together and screen share code. We also held meetings even though not everyone was free, and updated the members that weren't there on the work that was done. This was effective in mitigating scheduling challenges.

6.5 Carlos

This lab posed the most challenges for our team compared to other labs by far. While frustrating, these challenges allowed the group to think on our feet and adapt together. Our main challenge came from implementing the MCL algorithm in simulation. Our division of tasks made sure all the constituent parts were ready to come together relatively early on, a week before the briefing was to be held and this report was due. Our motion model and sensor model had been tested using provided unit tests, but we would come to find our sensor model was faulty and our motion model could be improved. From this, we learned the importance of robust test cases and that we should not rule out bugs in code just because we assume correctness. Aside from the technical issues, we also had to deal with sick group members and schedule changes. We used Zoom to facilitate whole group meetings even if people had to stay home, and if a member was too sick for that we made the effort to keep them updated through other avenues. Our issues would have been much worse had we not quickly learned how to adapt to challenges and continue working. After getting our simulation to work, our biggest challenge was time. With other little difficulties popping up along the way like trying to get graphs formatted right or deciding to take measurements in a different way after one avenue proved not useful, we just didn't have enough time to get the code on the racecar. This brought us to one of our final lessons: prioritize what you can do in the moment. We had been learning to do this along the way with each of our team members who were indisposed and thus made some work options harder or infeasible, but at the end of this lab we finally had to accept that we could only do so much. We finished what we could with the remaining time left, and resolved to fix any remaining issues on our own time after the lab to make sure our robot was still in good shape and that we had learned everything we could.