Lab #5 Report: Localization

Team #18

Cindy Wang
Eric Gonzalez
Herbert Turner
Jose Lavariega
Reinaldo Figueroa

6.141 RSS

April 5, 2022

1 Introduction (Eric)

This lab focused on the problem of localization, or the problem of determining where an object is within a larger context. This problem has wide variety of applications in robotics, since a robot may not immediately be aware of it's location within an environment. A robot may not even know the map of it's environment. The environment can even change over time, resulting in robotic systems that are robust to changing environments. One variation to this problem is explored in this lab, where a robot is dropped into a *known, static* map. This means the map will not change over time, and so the robot knows the map for all time. Designing a system that can locate itself at all is an important first step. Therefore, this lab implements the Monte Carlo Localization algorithm, or the Particle Filter algorithm, to determine a robot's location within the given environment.

The Monte Carlo algorithm relies on generating a distribution of possible particles for the robot with respect to the known environment it is placed in. Each particle serves as a hypothesis, or a guess, for the robot's location. The algorithm itself focuses on selecting the best particle as the hypothesized position. From the list of particles, the MCL algorithm averages out the particles and publishes the calculated pose as the guessed position for the robot. In order to generate the particles, update their motion, and prune the least likely particles, we also implement two models.

Due to the robot's movements, we must update the movement of our particles. This is achieved using a Motion Model. The robot takes it's measured odometry information, and applies this movement to all of it's particles. Due to imperfections in the vehicle's measurements, friction, and more, noise is introduced to the calculations. This Motion Model allows for the particles to update their movement as the vehicle would, while also preventing accidental convergence to just one guessed location due to the added noise.

As the robot moves, it also senses it's surroundings. For this lab, a LIDAR scanner is used. From the perspective of each hypothesized location, the Sensor Model calculates how likely each particle could have generated the given LIDAR scan. A particle with high probability would reflect a location that could have created the scan, and so it might be a correct guess for the robot's location. Using this information, the Sensor Model resamples particles weighted on their probabilities. Therefore, the particles should converge towards the robot's location, indicating that the robot has a sense for where it is in the map.

With these two models and the MCL algorithm, the vehicle, whether in simulation or reality, is able to calculate probable guesses for it's location. The rest of the lab report will dive into the more technical details behind the algorithm and models. It will also discuss our choices when it came to parameters involved in the models. Finally, we will evaluate the data and discuss the overall results in the lab.

2 Technical Approach

2.1 Motion Model (Herbert)

Let us consider a particle represented by a point that has a three-dimensional pose(x, y, θ). The motion model takes in odometry data that is given in the local coordinate frame of the particle and applies a transformation to give the new world pose of the particle. In its non-deterministic mode, it also adds noise to the odometry data based on particle-specific parameters. This is to give more realistic odometry measurements since noise is present in physical robotic systems that prevent actuators from perfectly matching their control signals. Below we will derive the process for both the transformation and noise addition:

Given our odometry data, because the previous position is at the origin in the relative frame, to find our new world pose at time step k, x_k, we simply transform our odometry dat into the world frame. To do this, we'll start with our equation for converting a relative position to the world frame and then solve for the relative position. Our transformation equation, derived in lecture, is:

$$\tilde{p}_{x_k}^{W} = \begin{bmatrix} R_{x_{k_1}}^{W} & P_{x_{k-1}}^{W} \\ 0 & 1 \end{bmatrix} \tilde{p}_{x_k}^{x_{k-1}}$$

where R is the trigonometric 2D rotation matrix. However, $\tilde{p}_{x_k}^{W}$ does not include the heading value,θ_k.We can calculate our new heading using the formula $\theta_k = \Delta\theta + \theta_{k-1}$.

To add noise to our odometry updates, we followed the derivation used in the probabilistic robotics textbook. After performing our deterministic frame transformation, let $\bar{x}, \bar{y}, \bar{\theta}$ be the the pose at the current time step and $\bar{x}', \bar{y}', \bar{\theta}'$ be the pose after adding the odometry data. We then separate the change in pose into three components according to the following formulas:

$$\begin{aligned} \delta_{rot1} &= \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \theta \\ \delta_{trans} &= \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2} \\ \delta_{rot2} &= \theta' - \theta - \delta_{rot1} \end{aligned}$$

Next, we add independent noise to each components by sampling from a guassian distribution with a mean of 0 and variance based on a weighted average of components. Later, discussion will select how the parameters for our weighted averages are chosen.

$$\begin{aligned} \hat{\delta}_{rot1} &= \delta_{rot1} - \textbf{sample}(\alpha_1\delta_{rot1} + \alpha_2\delta_{trans}) \\ \hat{\delta}_{trans} &= \delta_{trans} - \textbf{sample}(\alpha_3\delta_{trans} + \alpha_4(\delta_{rot1} + \delta_{rot2})) \\ \hat{\delta}_{rot2} &= \delta_{rot2} - \textbf{sample}(\alpha_1\delta_{rot2} + \alpha_2\delta_{trans2}) \end{aligned}$$

Finally, we can compute the updated particle position using the noisy odometry components.

$$\begin{aligned} x' &= \bar{x} + \hat{\delta}_{trans}\cos(\theta + \hat{\delta}rot1) \\ y' &= \bar{y} + \hat{\delta}_{trans}\sin(\theta + \hat{\delta}_{rot1}) \\ \theta' &= \bar{\theta} + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \end{aligned}$$

2.2 Sensor Model (Cindy)

The other component for the particle filter is the implementation of the sensor model detailed in the handout. Given a particle pose x_k in a known map m, we want to determine the probability $p(z_k \mid x_k, m)$ that the robot's current lidar scan reading could be recorded. This probability will then be used in the particle filter, where we only care about the particles above a given probability threshold.

This probability is calculated as follows where n is equal to the total number of beams in a single scan, and we want to take the product of all the probabilities of the individual beams.

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

To determine $p\left(z_k^{(i)} \mid x_k, m\right)$, there are 4 cases to model of which we then take a weighted average. The cases to model include the probability of detecting a known map obstacle, the probability of being a short measurement, probability of a large measurement, and the probability that it is a completely random measurement. It is necessary to include a weighted average of models instead of one probability distribution model because each case does have some chance of happening, and each case has its own likelihood behavior with respect to the scan and pose. We can then tune the weights as necessary when running the car in the real world map. The model for each case is as follows:

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

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

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

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

The weighted average is as follows:

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

$$\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1$$

For the implementation in code, we used a lookup table to speed up computation. Because the sensor model is called for every particle in the particle filter, it would be too slow to compute the likelihoods for every iteration. Rather, it would be better to store the likelihoods in a 2d numpy table indexed by the scan beam's range z_k and the ground truth distance d computed using ray tracing from the particle pose. We used a table that was of dimensions $201x201$. The initial coefficient parameters that we went with was $\alpha_{hit} = 0.74, \alpha_{short} = 0.07, \alpha_{max} = 0.07, \alpha_{rand} = 0.12$
In the implementation, we would pre-populate the sensor model look-up table one time when the SensorModel object was initialized, and then continue to use this table for following computations. We implemented a function that passes in a list of N particle poses, and scan data for that observation. We would then compute the likelihoods for the scan data with respect to each of the N particle poses to return N likelihoods.

2.3 Particle Filter (Jose)

Our team makes consistent use of the Motion Model and the Sensor Model formulations above, to implement a Particle Filter. The specific implementation of this particle filter is called Monte Carlo Localization. As an overview, the algorithm works as follows:

Given a known map of the environment, we use a particle filter to represent a distribution of likely states, in this case the robot state is the pose of the robot (x,y,yaw). Each particle is a hypothesis of where the robot may be. Starting from an initial distribution, simulated as noise from some initial pose, we then aim to converge to an average belief that is correct. Whenever the car moves, the particles are shifted to predict a new state after the movement. The particles are resampled based on a Sensor model estimation, based on how well the sensed data correlates with the predicted state. The particles are resampled based on recursive Bayesian Estimation. The particles should converge (with decreasing variance) to the actual position of the robot.

In the following subsections, we will be providing a better visualization of how this works in our code, as well as some of the insights we have gathered from using this filter.

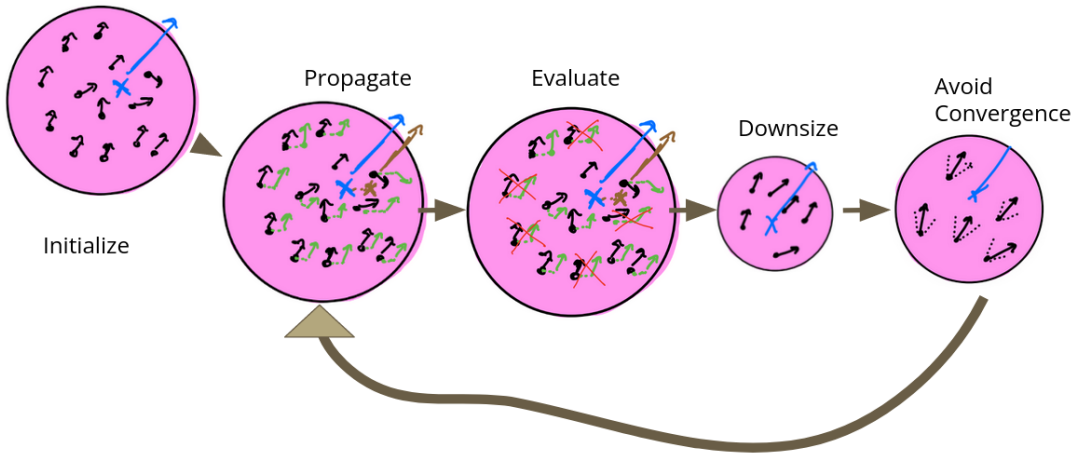

Figure 1: Visualization of the Monte Carlo Localization Algorithm

2.3.1 Algorithm Architecture (Jose)

In this section, we show an overview on how the algorithm works and what caveats are to be considered when implementing on a non thread-safe language. Notably, by introducing threads.

In figure 1 , each step is indicated by a pink circle, and the size of each pink circle represents the variance across hypothesis states at each step.

In our **Initialize** Step, we determine a base position and create a scattering of candidate particles, each uniformly likely to be correct. The randomization to represent differences among these particles is a normal distribution with a variance according to the case of using simulator or real robot.

In our **Propagate** step, we use the motion model to move all of the particles forward by one discretized timestep, according to motion model dynamics. We add some noise to the overall motion model to show even more differences among the particles. This step increases the overall variance across all the particles.

In our **Evaluate** step, we use the sensor model on all existing particles to determine a likelihood of each particle being correct, given the particles pose and the known observation about the known map. The observation in this case is the LIDAR scan that the real or simulated robot is seeing, so there is only one LIDAR scan to be sent to all of the particles. Based on the motion model, we obtain a distribution (no longer uniform) of how likely each particle is of being correct. This is done through a Bayes Filter.

In our **Downsize** step, we resample particles based on weights assigned by the previous step (i.e. the likelihoods), which leaves us with a subset of particles with lesser variance. Enforcing some convergence.

In our **Avoid Convergence** step, we really want to avoid converging to a single particle. We do this in order to avoid having a faulty particle that is guaranteed to drift, rather than offering resample options. So during our avoid convergence step we add noise to the beliefs to generate new particles. This is done independently of if the odometry data from the robot is treated as deterministic or non deterministic.

Our implementation on the robot, has to take care of threading, and is best explained by the following diagram: Architecturally in figure 2 , note that we implement two main threads, running almost in parallel (as information from either the odometry or the lidar is received by the robot computer) these are represented as a main Lidar Callback and a main Odometry Callback, keeping with the standard conventions for ROS.

In our *Lidar Callback*, the raw LIDAR information is fed into the sensor model, which returns probabilities. These are then used to resample particles. With resampled particles and by generating new particles, we take an Average pose, which is the mode for the x and y states, but is a circular mean for the yaw state. The average pose becomes

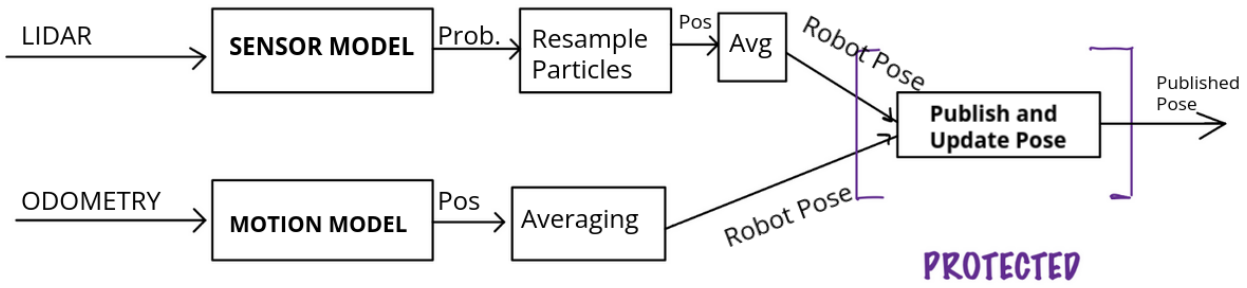

Figure 2: Code Implementation Diagram

the published pose. This is then published to the robot pose, which is thread-protected, to avoid both the Lidar callback and the odometry callback to access the same critical region. The published pose is then broadcast as the belief.

In our *Odometry Callback* we feed the raw odometry into the motion model, from which we get updated propagated in time particle positions. These are then averaged by a simple mean for x and y and by a circular mean for the yaw sate, and published to the robot pose. During the odometry motion model evaluation, we add more noise to our particles in the yaw value, to avoid convergence. In the next subsection we will discuss why this is important.

2.3.2 Main Insights- Discussion (Jose)

So far we have formulated a base implementation of the Monte Carlo Localization algorithm. It may be possible to note that as formulated so far, the probabilities and resampling from the sensor model make us have a downwards trend in the number of particles, by constantly pruning them. This eventually leads us to converge on a single particle, which will almost always be wrong. In order to avoid this, we resample by generating new particles by adding artificial noise to our motion model. This allows us to propagate all of our particles forward and generate new particles whenever a particle is pruned.

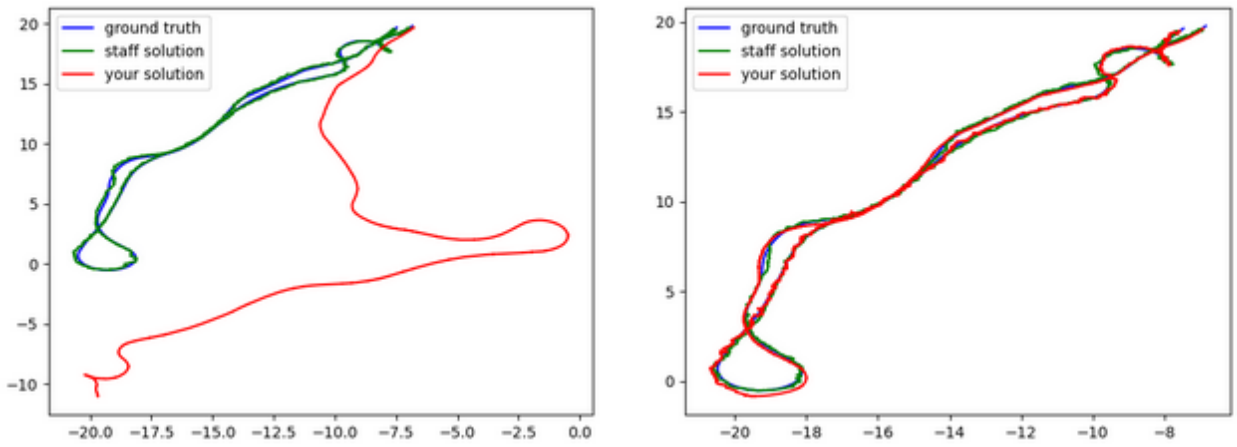

Figure 3: Axes are x,y positions

In the figure 3 above, we show how striking the difference is between not resampling and resampling. Both of the trials were run through an autograder with the highest level of noise possible. Only pruning is done on the left and pruning and resampling is done on the right. This is why it is so important to resample particles by adding artificial noise to the motion model.

2.3.3 A Derivation for the Bayes Filter (Jose - Herbert)

In this lab, we use a Bayes Filter freely to perform an evaluation on our particles via the sensor model. This section focuses on a derivation of the Bayes Filter.
Show:

$$P(x_k|u_{1:k}, z_{1:k-1}) = \int P(x_k|x_{k-1}, u_k)P(x_{k-1}|u_{1:k-1}, z_{1:k-1})dx_{k-1}$$

$$P(x_k|u_{1:k}, z_{1:k}) = \alpha P(z_k|_k)P(x_k|u_{1:k}, z_{1:k-1})$$

Given:

1. x_k
2. u_k
3. z_k
4. $P(x_k|u_k, x_{k-1})$
5. $P(z_k|x_k)$
6. Bayes Rule: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$
7. Law of Total Probability: $P(A) = \sum_n P(A|B_n)P(B_n)$
8. Markov Property: $P(x_t|x_{t-1}, u_{1:t}, z_{p:t-1}) = P(x_t, x_{t-1}, u_t)$

Proof:
We start of with establishing an important property of the Bayes Rule (6) , when we have more than two variables:

$$P(A|B, C) = \frac{P(B|A, C)P(A|C)}{P(B|C)}$$

Which means:

$$P(x_k|u_{1:k}, z_{1:k}) = \frac{P(z_k|x_k, z_{1:k-1}, u_{1:k})P(x_k|z_{1:k-1}, u_{1:k})}{P(z_k|u_{1:k}, z_{1:k-1})}$$

We can treat the denominator like a normalizing constant, $\frac{1}{\alpha}$, which means that we have the form:

$$P(x_k|u_{1:k}, z_{1:k}) = \alpha P(z_k|x_k, z_{1:k-1}, u_{1:k})P(x_k|z_{1:k-1}, u_{1:k})$$

And by the Markov property (8), above, we can say that:

$$P(x_k|u_{1:k}, z_{1:k}) = \alpha P(z_k|x_k)P(x_k|z_{1:k-1}, u_{1:k})$$

Which shows the second equation we are trying to prove.

From the equation above, we focus on the term:

$$P(x_k|z_{1:k-1}, u_{1:k})$$

By the law of total probability (7):

$$P(x_k|z_{1:k-1}, u_{1:k}) = \sum P(x_k|x_{k-1}, z_{1:k-1}, u_{1:k})P(x_{k-1}|u_{1:k}, z_{1:k-1})$$

And if we convert this to Continous Time:

$$P(x_k|z_{1:k-1}, u_{1:k}) = \int P(x_k|x_{k-1}, z_{1:k-1}, u_{1:k})P(x_{k-1}|u_{1:k-1}, z_{1:k-1})dx_{k-1}$$

And again, by the Markov Property

$$P(x_k|z_{1:k-1}, u_{1:k}) = \int P(x_k|x_{k-1}, u_k)P(x_{k-1}|u_{1:k-1}, z_{1:k-1})dx_{k-1}$$

Which proves our first equation! Hence Both Equation 1 and Equation 2 have been Shown, which is what we aimed to do during this section.

3 Experimental Evaluation - Jose

In this lab, our team took many better experimental practices, most notably a standardized way to display error and offset, in the form of a ros topic that is easily subscribed to.

3.1 Standardized RQT Graph Evaluation Architecture (Jose)

We generate a new rostopic, called "localization error" that is used to track the error in the state of our averaged belief compared to the simulator ground truth. It also tracks the variance across all states (x,y,yaw) for our particles. The latter of these two outputs is useful both on the car and on the simulator, while the error in belief is used mostly for the simulator, as we have no consistent ground truth to compare when using rosbags. Implementing this has been incredibly benefitial to our team when generating figures and quantitatively analyzing parameters, and will be important for us to implement as a priority in further labs.

3.2 Tunable Parameters and Results in Simulator (Jose)

Assuming a correct implementation of the particle filter, really the only tunable parameters in the algorithm are the noise with which we initialize the robot position to determine the initial distribution of particles, the noise with which we add to every odometry step, and the noise that the motion model carries over as twist noise for each of the particles (for resampling).

What worked best in the simulator case for the initial particle distribution was a normal distribution with variances of 0.5,0.5 and $\frac{\pi}{8}$ for the x, y and yaw states respectively, with a mean around the initial point.

What worked best for the odometry noise was a normal distribution noise about the origin with a variance of 0.125. This is added to the base odometry data to make a normal distribution about the odometry data with a variance of 0.125.
The results from these parameters in simulation are shown in figures 4 and 10 below:

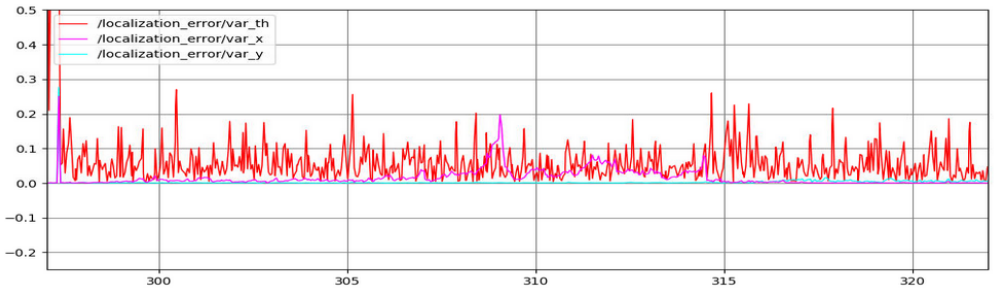

Figure 4: Variance in states across simulator run. Horizontal axis is timestamp, vertical is variance

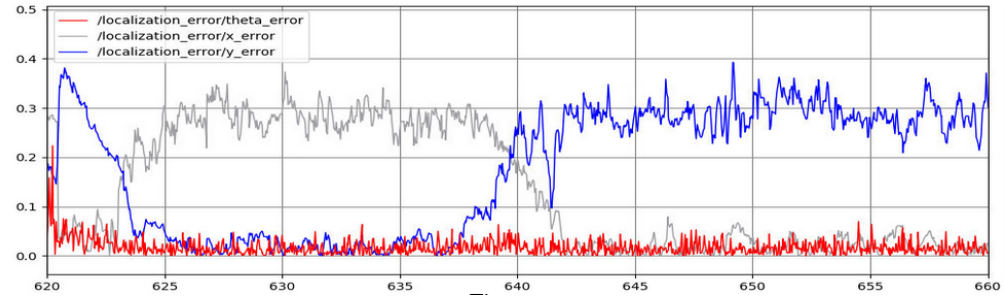

Figure 5: Error in state across a simulator run. Horizontal axis is timestamp, vertical is absolute error (meters or radians)

The variance as observed in 4 is highest for the yaw coordinate, although still relatively low. This is best explained by the random noise in the yaw by way of twist that we add to the sensor model, to avoid convergence to a single

particle. Despite this, the variances in x and y coordinates are neglegible.

The error as observed in 10 is very small for the yaw angle(in radians), although inversely proportional for x and y coordinates. Despite this, the error seems to stabilize around 30 cm in the sim. Averaged out over x and y coordinates both, the error is only 15 cm. The run from this data was captured in comes from a run on an open field in the simulator environment (sparsely cluttered) and running the wall follower controller. This is excellent error to have as there are no easily identifiable features in this environment.

3.3 Tunable Parameters in Car(Herbert)

In the Technical Approach section, we mentioned techniques for adding noise to our models to mimic effects found in physical robotic systems. While we could ignore or randomly choose parameters for the simulator, to get our particle filter to work well on the physical race car we had to perform a parameter search to find values that worked well. Through conversations with other teams and TAs, we decided that empirical evidence suggested the sensor model α parameters could be left untouched. However, the motion model noise adding technique we used, meant we had to determine those parameters ourselves. Initially, we simply tried a binary search of series of values on rosbag data. However, that was unsuccessful so instead we built a visualization tool to help us show how the noise parameters affected the position updates of the particles. We tuned the parameters until we ended up with a final position distribution that was similar to those shown in probabilistic robotics textbook. We found this led to much more accurate localization. Figure 6 shows an example from our visualization tool for our current chosen noise parameters.

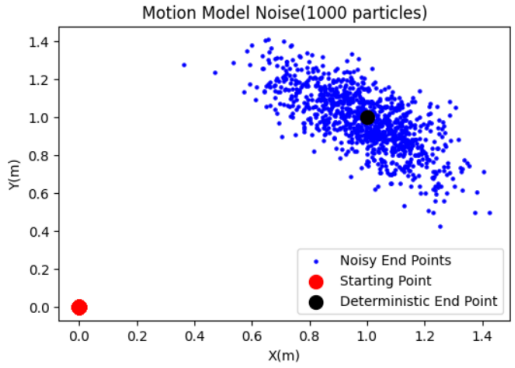

Figure 6: Visualization of 1000 particles with the same starting point given odometry with sampled noise =

3.4 Results on the Car with Rosbag (Herbert-Reinaldo)

To test our algorithm on the car, we ran the Racecar through the Stata basement and recorded a rosbag with main topics that we would need to perform testing. The topics were the /joystick (from teleop,) the /scan, and the /odom. These topics allowed us to play back the rosbag on our simulation to see the real performance of our racecar while running our localization algorithm. In the simulation we plotted the particle points representing our sampling of possible positions, the estimated pose at each callback, and the ground truth pose of the robot model based on the joystick command data.

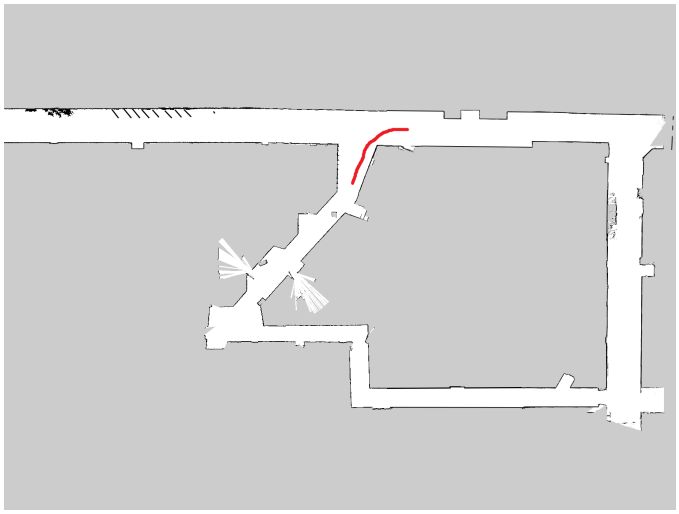

Figure 7: Rough path taken by car in recorded rosbag

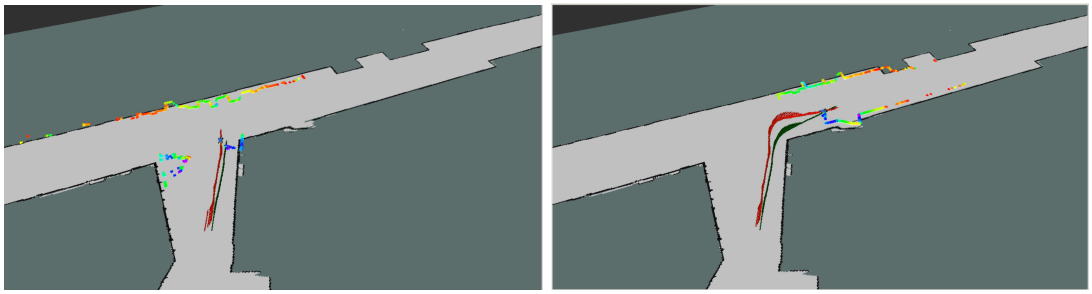

Figure 8: Laserscan, estimated pose(red), and ground truth pose(green)

As you can see on figure 8, the estimated pose (red) and the ground truth(green) are close to each other with some small error. This error is most likely due to an incorrect initialized pose inside the simulation map. In future testing it will be run directly in the car for more accurate results. Overall, testing showed that our algorithm performs well with the racecar moving very close to the ground truth pose.

4 SLAM on the Cartographer (Herbert)

Google Cartographer is a standalone C++ library that supports real-time simultaneous localization and mapping(SLAM) in both 2D and 3D. After setting up our environment, we ran Cartographer in the course simulator to produce a 2D rendering of building 31. Cartographer used the simulated laserscan and odometry data to generate estimated poses of the racecar as well as generate occupancy grids of the simulate environments. After running Cartographer in simulation for about 15 minutes, we use the ROS map server package to process the final occupancy grid and generate our 2D map. In figure 9, you can find our map of building 31 alongside the ground truth map. We also generated a pixel-by-pixel comparison of the two maps with the differences colored red in figure 10.

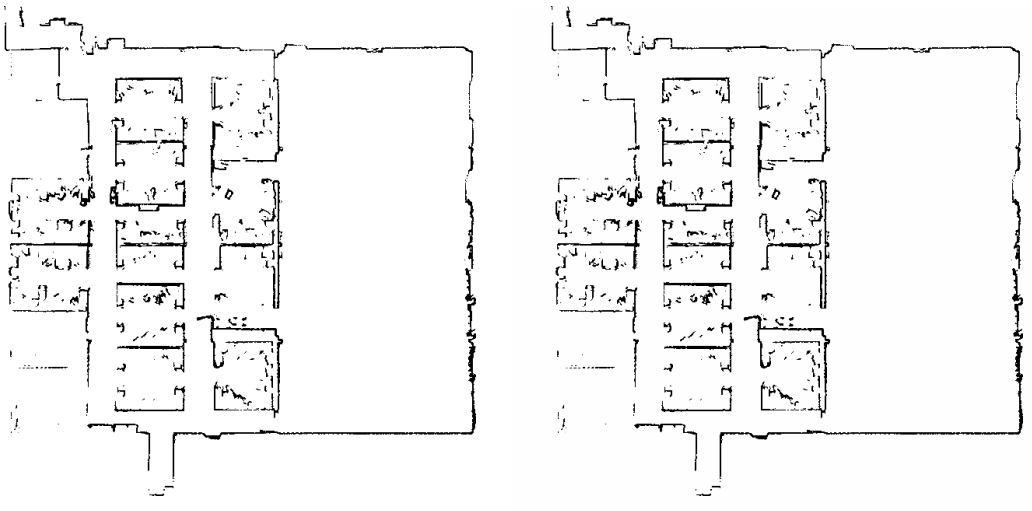

Figure 9: Left: Ground Truth Map; Right: Cartographer map

Figure 10: Pixel comparison of the two maps with differences highlighted in red

5 Conclusion (Eric)

Overall, the Monte Carlo Localization algorithm performed well. As demonstrated in the data, running trials in simulation was highly successful, with the robot closely following the ground truth measurement in Figure 3. Given how successful the initial implementation was, the research team moved forward with tuning parameters involved in the algorithm and models.

The team also demonstrates success when measuring error and variance after having decided upon a set of parameters and noise distributions to use. The robot is able to estimate its location and orientation in the known map reference frame relatively accurately.

The robot's success in localizing itself within a known, static map provides promising reason to move forward with the problem at hand. In the future, we hope to be as successful for the SLAM problem, or Simultaneous Localization And Mapping problem, where the map is not known initially.

6 Lessons Learned

6.1 Cindy

In this lab, I learned about how a particle filter implementation all fits together. From the first writing assignment, to coding the Sensor Model along with Rey, to observing how noise can improve a filter, I learned much from discussions with my teammates. More specifically, I worked on understanding the sensor model, and it wasn't until I was testing it did I fully understand how it worked. I was able to complete the writing component without a full grasp on what all the math meant, but implementing and debugging allowed me to realize how cool a particle filter is. I learned about the efficiency of using numpy arrays and how to better utilize them. I faced a great deal of debugging on the sensor model, largely due to precision and rounding issues. When running the particle filter, I realized that the localization problem can be difficult due to random obstacles and inaccurate maps. So while our implementation can be correct, sometimes it is also important to step away from the code and think about how to better choose a path to record our rosbag.

6.2 Eric

I learned a great deal about the localization problem thanks to this lab! I have learned a framework for which I could tackle this problem in the future for similar systems, as well as similar problems. Making sense of the math and the motivations for it in the Sensor Model made the problem approachable and solvable. It is incredible that the vehicle can deduce so much using measurements of its surroundings, its own movements, and clever math to solve for an estimated pose. Beyond the content of the lab itself, I've gained more practice generating rosbags and selecting topics for it. These are excellent for playing back vehicle movement to quickly test changes, or to help generate data on the fly.

6.3 Herbie

During this lab I learned a lot about how particle filters work and how using accurate probability distribution for noise affects their accuracy. I also learned a lot about the different mathematical models for robotic motion and odometry. Building the visualizer tool also taught me useful skills for plotting and making visualization tools in general using python and matplotlib. ROS-wise I gained new knowledge about how to use the rosbag tools as well as analyze them more thoroughly with both the commandline and RViz. Lastly, I learned a lot about Google Cartographer and SLAM in general and how these tools can be used in the real world.

6.4 Rey

In this lab we all learned a lot about how to solve the localization problem of a robot in a known environment by using Monte Carlo Localization. The biggest takeaway for me while working in this lab is that sometimes it is really important to understand how all the pieces interact and get together. Without this, it becomes really hard to comprehend the purpose of my task in the lab. Specifically, I worked with Cindy on the implementation of the Sensor Model. While working on it, we were able to appreciate the convenience that numpy provides us when doing a lot of repetitive and exhaustive calculations. Debugging the Sensor Model allowed us to comprehend even further its role in the lab and in the algorithm in general. While testing on the Racecar we realized there are a lot of inconveniences that could be intervening in our results. For example, maps seem to be pretty old and do not accurately represent the Stata basement. Overall, once again I realized how important it is to meet and effectively communicate with your peers in order to have an excellent teamwork experience.

6.5 Jose

Besides the technical implementation of the particle filter and some of the insights that are gained by adding additional noise to our odometry, to ensure that we never converge to a single particle and so, we avoid drift, most of my learning was on the testing side. It was very nice that we had the structures to test and quantitatively visualize our error from the parking controller lab in rqt. I implemented this architecture for our current lab, and it has made our life so much easier as a result. We can generate figures on the fly for different trials, and allows us to compare parameters better. This is something I will be taking to other labs and among the first things I will be implementing at the start of each lab.