Team 5
Lab 5: Localization

04/08/2019

Hector Castillo
Mitchell Guillaume
Claire Traweek
Daniel Wiest
Franklin Zhang

1. Overview and Motivations

(Authored by Mitchell Guillaume)

1.1. Lab Overview

Determining a robot's location in a given map may seem like a simple task at first glance, but it is in fact quite challenging and is a very active field of research within robotics. Our team was tasked with implementing a Monte Carlo Localization--also known as MCL or a particle filter localization--algorithm for our autonomous racecar. MCL is one of many ways for a robot to determine its orientation and position in a known map.

Monte Carlo Localization is a stochastic algorithm. This means that it calculates where the robot most likely is, but cannot know the location with absolute certainty. Before we can successfully implement MCL, we must first create probabilistic models of our robot. For extra credit, our team attempted to implement simultaneous localization and mapping (SLAM) using Google Cartographer.

1.2. Goals

■ Sensor Model

For each lidar beam, there are four cases for what the measured distance can be. The goal of the sensor model is to, given a ground truth distance, calculate the probability that the sensor reports a measurement distance.

■ Motion Model

The motion model is the second model required to implement our particle filter. The motion model extrapolates the robot's position and adds noise to the odometry data to account for uncertainty relating to tire slip and other factors.

■ MCL Particle Filter

The particle filter utilizes both the motion model and the sensor model to execute the MCL algorithm. By using both odometry and lidar data, the MCL algorithm calculates the most likely position of the robot in a known environment.

■ Simultaneous Localization and Mapping

A particularly active area of research in robotics is simultaneous localization and mapping (SLAM). Once we successfully implemented the MCL Particle Filter, our final goal was to use Google Cartographer to generate maps of previously unknown environments.

1.3. Motivations

The ability for a robot to localize itself is critical in any situation in which the machine will autonomously move through an environment. This is because an understanding of a robot's position is necessary in order to connect higher order autonomy operations like motion planning with lower order functions like control.

Additionally, when coupled with the ability to map an unknown environment, localization becomes even more powerful. If a robot can simultaneously create a map of an unknown environment, and localize itself within that map, then the robot is capable of exploring previously unknown places. The applications for this are practically endless, including disaster relief, search and rescue operations, and exploration of potentially hazardous environments. More generally, SLAM allows robots to move around autonomously without the need of a pre-existing map.

2. Proposed Approach

2.1. Sensor Model

(Authored by Daniel Wiest)

- **Approach**

Almost all physical processes introduce variability. Dimensional variability in manufacturing applications, genetic mutations in biological DNA replication, and differences in students' test scores are all the results of a process with inherent variability. The process in which sensors measure and report data is no exception–it is also a process with variability.

Instead of merely accepting this variability as a fact of reality, this variation can be modeled with statistics. In manufacturing, the probabilistic distribution of part dimensions can be modeled by measuring a subset of all parts produced. This allows the entire production line to be characterized by a probabilistic (typically Gaussian) distribution. In the case of our robot's lidar making distance measurements, the resulting distribution is *not* purely Gaussian, as there are four discrete cases for the resulting measurement of each laser beam. In each situation, z_t^* will refer to the actual (ground truth) distance and z_t will refer to the measured distance.

In the first of these four cases (Equation 1), the laser beam successfully reflects off of the obstacle and reports back to the lidar. This case *does* have an approximately Gaussian distribution, with its mean centered on the ground truth distance to the obstacle.

$$p_{hit}(z_t|x_t, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_t - z_t^*)^2}{2\sigma^2}\right) & \text{if} \quad 0 \le z_t \le z_{max} \\ 0 & \text{otherwise} \end{cases}$$

Equation 1

The second case (Equation 2) is when the laser bounces back off of something else and the lidar reports a distance that is shorter than the distance to the real obstacle. This can occur due to scratches on the lidar's case, interference with objects on the robot itself, or even temporary obstacles like people walking between the robot and the obstacle. This case has a distribution that is inversely proportional to the ground truth distance squared.

$$p_{short}(z_t|x_t, m) = \frac{2}{z_t^*}\begin{cases} 1 - \frac{z_t}{z_t^*} & \text{if} \quad 0 \le z_t \le z_t^* \\ 0 & \text{otherwise} \end{cases}$$

Equation 2

In the third case (Equation 3), the laser beam does not successfully bounce back to the lidar, and the lidar reports its maximum range. (The Velodyne Puck reports inf, but our code changes all infs to the maximum range of the lidar). This distribution is simply a step function: 0, or 1 if the ground truth distance is greater than the maximum lidar range.

$$p_{max}(z_t|x_t, m) = \begin{cases} 1 & \text{if} \quad z_t = z_{max} \\ 0 & \text{otherwise} \end{cases}$$

Equation 3

The fourth and final case (Equation 4) is the "everything else" case. This distribution is a constant value equal to the inverse of the maximum range. This case accounts for any and all random measurements that aren't any of the three previous cases.

$$p_{rand}(z_t|x_t, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \le z_t < z_{max} \\ 0 & \text{otherwise} \end{cases}$$

Equation 4

These four cases are each multiplied by a scaling constant, and then superimposed to create a single probability distribution for the sensor (Equation 5).

$$p(z_t|x_t, m) = \alpha_{hit} \cdot p_{hit}(z_t|x_t, m) + \alpha_{short} \cdot p_{short}(z_t|x_t, m) + \alpha_{max} \cdot p_{max}(z_t|x_t, m) + \alpha_{rand} \cdot p_{rand}(z_t|x_t, m)$$
$$\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1$$

Equation 5

In our implementation, we used the six constants calculated in part one of the lab assignment. For their values, see Table 1. In addition to these six constants, it's worth noting that the maximum range of the lidar is 200 meters, so $z_{max} = 200$.

Constant	Value
σ	0.5
α_{hit}	0.74
α_{short}	0.07
α_{max}	0.07
α_{rand}	0.12
η	1.0946

Table 1: Sensor Model Probability Constants

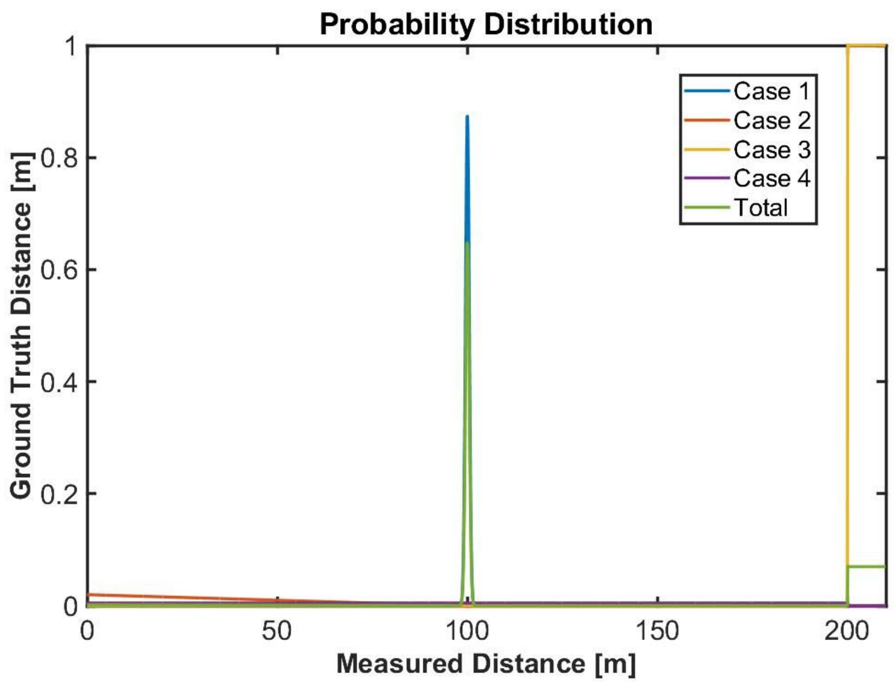

Figure 1: Sensor model is pictured with a ground truth distance of 100 meters. All four components of the distribution are included in the graph. The probability spike associated with the ground truth distance appears very thin because of the 200 meter maximum range of the Velodyne Puck lidar system.

Because of the large number of components in the probability distribution seen in Equation 5, it is computationally expensive to compute its value on every single evaluation of the sensor model, therefore, the probability space was descritized with a resolution of 7 pixels per meter as a function of measured distance and ground truth distance and stored into a python dictionary object during initialization in order to allow for faster evaluation at runtime. The resolution was limited by the runtime associated with building the probability space dictionary, and we found that 7 pixels per meter was a good tradeoff between resolution and the time necessary initialize the model. This discretized space is pictured in Figure 2.

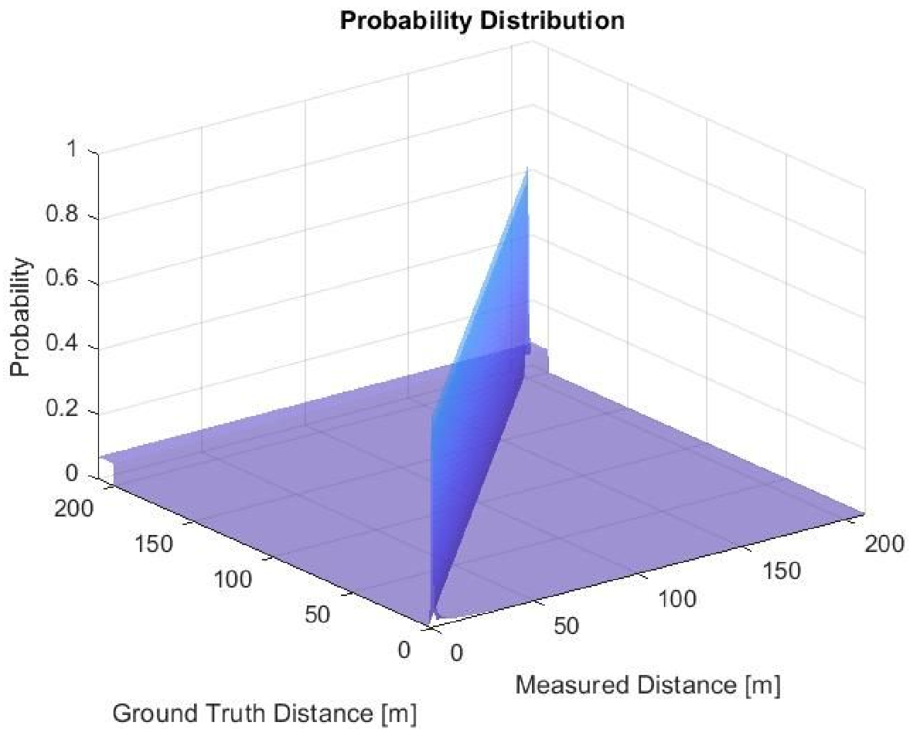

Figure 2: Two dimensional probability space as a function of ground truth and measured distances for our Velodyne lidar with a maximum range of 200 meters.

This two dimensional space is discretized and stored into a dictionary during initialization for efficiency. Once again, the peak associated with the correspondence of the ground truth and measured distances appears very thin because of the 200 meter range of the Velodyne Puck lidar sensor.

During sensor model evaluation, the ground truth distance z_t* is provided by a simulated scanner (included in the lab source code) that uses ray tracing to find the ranges that would be recorded by the car if it was positioned at a given particle's pose. In order to compute the probability of each particle given the uncertainty introduced by the motion model, for each particle, the probabilities associated with all of the beams given the observed (z_t) and ground truth (z_t*)

measurements are multiplied in order to produce a scalar quantity representative of how likely it was that the observed scan was taken from that specific particle. This process is illustrated visually in Figure 3.

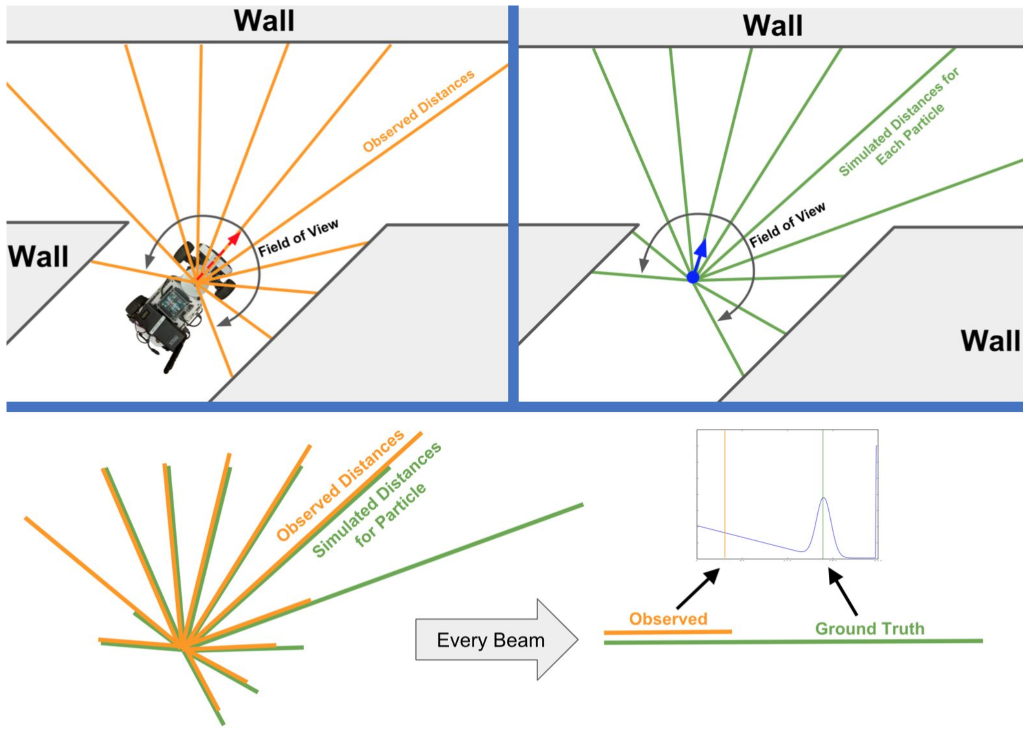

Figure 3: Visual depiction of the evaluation of particle probabilities.

Given the observed distances pictured in the top left corner by the robot, the simulated (ground truth) distances are computed for each of the particles (as seen in the top right) then used as the basis by which each particle's probability is independently assessed. This assessment utilizes the probabilistic sensor model discussed above in order to return the probability of each particle.

One problem that we quickly encountered with this approach was that because there are 25 individual beam probabilities multiplied together, even if each beam had a probability of 0.75, then the resulting multiplied probability would be 0.075^{25} ≅ 0.00075. As is illustrated above, a lot of high probabilities resulted in a very small probability when multiplied and in cases with even smaller beam probabilities, we began to push the boundaries of computer floating point precision. This also hurt the performance of our particle resampling (see paragraphs below for implementation details) because when normalized so that the probabilities of all particles summed to one, most often this would result in a single particle with very high probability and everything else having essentially zero probability, causing the resampling to collapse the particles towards a single point much too quickly to appropriately explore the range of potential locations.

In order to solve the problem of vanishing probabilities, the particles probabilities were raised to the $\frac{1}{65}$ power. This number was deemed appropriate because it allowed for the particles to collapse much more slowly towards the correct power, providing greater opportunity for exploration and allowing convergence to the correct location even when particles were initialized in the incorrect orientations or locations (see relocalization video posted to the website).

After the probabilities of all 250 particles are updated, the process of resampling is initiated. This process involves choosing which particles to keep and which to prune back based on the specified probabilities determined in the previous steps. After a fair bit or research into different resampling techniques, the *systematic resampling* approach was selected because of its computational efficiency and other favorable features cited in this research paper. This approach involves the selection of resampled points from a randomly selected ordered sampling frame. Once correctly implemented, the resampling function works in opposition to the uncertainty-adding effects of the motion model discussed in section 2.2 in order to maintain a quality estimate of the robot's global position.

To estimate its pose, the program must calculate a working average of all the possible particles. The estimated x and y values are simply the arithmetic mean of the particle x and y values, respectively. Estimating the average theta value is a bit trickier because it is a circular quantity. To do this, all of the particles' theta values are broken into x and y components as if theta were an angle on the unit circle. Then, the arithmetic mean of those x values and the arithmetic mean of the y values are calculated. Our average theta is the angle that will point to the (x,y) point of these mean values. So, the average theta is the arctan this point.

$$\theta_{average} = arctan(y_{average}/x_{average})$$

Equation 6

All of the above processes outlined above come together to take incoming scan data, calculate particle probabilities, resample the current particles based on these updated probabilities, then return the new particles for the rest of the particle filter to interact with. This process is outlined visually in Figure 4.

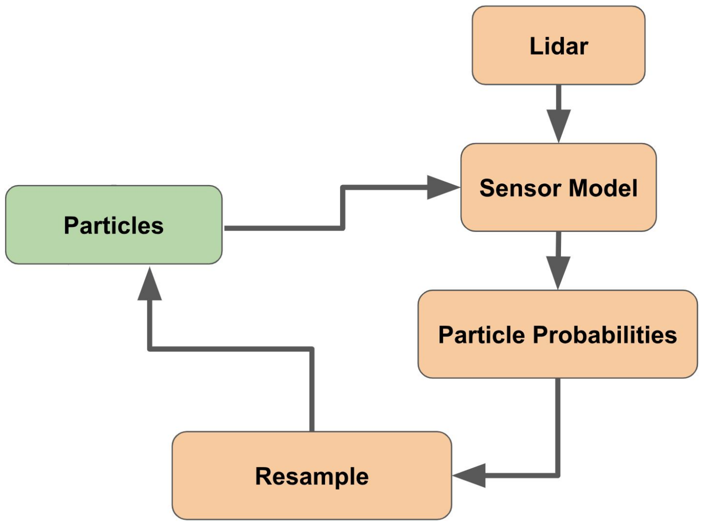

Figure 4: An overarching graphical view of the interaction between different components of the sensor mode.

■ Evaluation

In order to quantitatively evaluate the performance of the sensor model, a *cross track error* metric was utilized to express the distance in meters between the robot position calculated by the particle filter and the ground truth robot position. The error in angle was also quantified using the same method. Figure 5 below shows this metric versus time while traversing the simulated stata basement environment. This method was used to assess the effects of parameter adjustments during tuning.

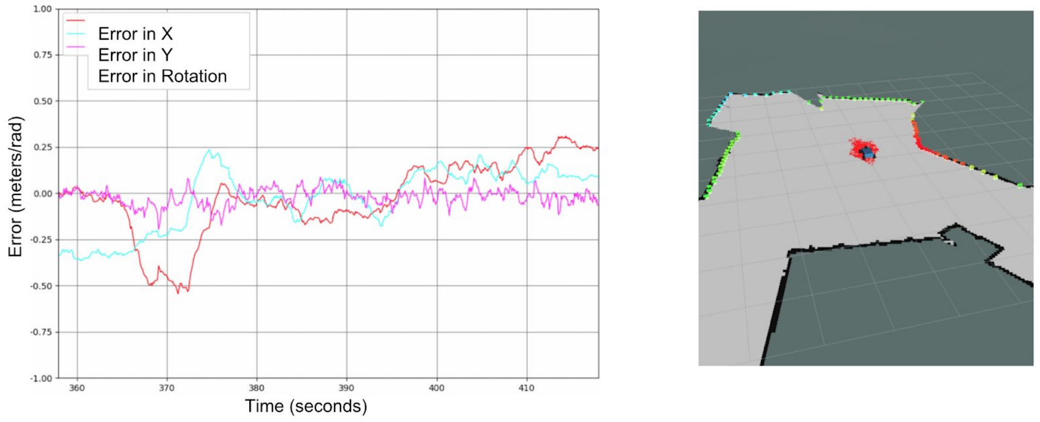

Figure 5: Positional and angular error versus time during evaluation in simulation. A *cross track error* metric was defined as the distance between the ground truth and calculated positions in order to quantify the performance of the sensor model.

As illustrated in Figure 6, the combined system performed quite satisfactory on the actual robotic system, even allowing enough flexibility for our operator to write out MIT with the vehicle. The system also met all runtime speed requirements, with positional updates provided at an average of 35 Hz during particle filter operation.

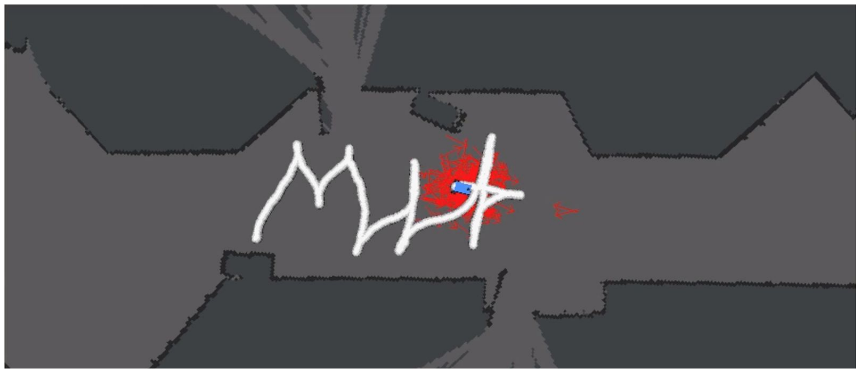

Figure 6: Tracing MIT with the Robot

The robot's pose accuracy was robust and accurate enough to allow for our operator to trace out MIT while driving the robot in the basement. To see the full video see our website.

2.2. Motion Model

(Authored by Hector Castillo and Franklin Zhang)

- **Overview**

  (Authored by Franklin Zhang and Hector Castillo)

  The objective of the motion model is to account for any error in the robot's odometry. This error can come from inaccurate measurements, sensor noise, wheel slippage, etc. The motion models works by taking a list of possible poses called particles where the robot might be, and updates their position based on odometry values. In order to account for error, random noise is added to each of the particles. Figure 7 is shown below. The left picture shows the particles at a certain time and the particles on the right show the particles in a future time with multiple motion model updates in between. The introduction of noise makes the particles at the next time step have a larger spread. Its difficult to accurately predict what types of noise may be introduced into the physical system. However, we can consider what happens when different amounts of noise is added. The essence of the motion model is to consider how different quantities of random noise can affect the system and to explore possible trajectories of the robot in simulation. The model has a explore and exploit aspect to it. Having a large amount of noise enable ts the robot to capture the times when there is an introduction of a lot of noise.

The exploit aspect of the model makes the model more confident and have a smaller spread. With this, it can more accurately predict the pose when there is sparse sensor model feedback. An important part of the design is to balance the exploit and explore spectrum of the model by changing the amount of noise we add into the model.

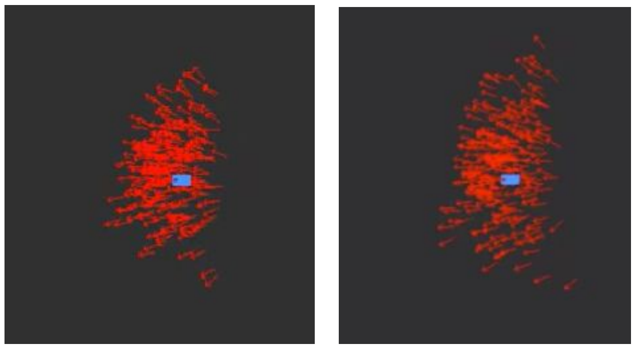

Figure 7: Motion model updates the particle poses, shown in red from a set of initial poses to new sets of poses. The motion model introduces some noise into the particle poses, causing the particles to spread.

- ### Odometry

(Authored by Franklin Zhang)

The motion model gets its odometry information from a topic called /odom which integrated drive commands and outputs the robot's pose relative to the robot's initial reference frame. This information is used to calculate the robot's change in pose by subtracting the previous pose from the current pose.

$$\Delta X = X_t - X_{t-1}$$

Equation 7

- ### Accounting for Error

(Authored by Hector Castillo)

In order to account for error from inaccurate odometry measurements, wheel slippage, etc., the motion model introduces noise to ΔX on each of its components, x, y, and θ. Noise is introduced in two components, a base level of noise and scaling noise.

$$\Delta X' = \Delta X \times e_s + e_b$$

Equation 8

The base level of noise is there to account for drift in directions that are given a zero velocity command, such as error in y and θ when the robot is trying to drive straight. The scaling component keeps the noise proportional to the distance traveled or rotated and accounts for the increased error at higher velocities. The noise is generated randomly with a normal distribution. Evaluation and selection of the standard deviations that generate this noise is discussed in detail in the Evaluation section, 2.2.7.

- **Transformation Matrix**

  (Authored by Franklin Zhang)

  In order to update the particles' poses, the odometry values need to be translated into the robot's reference frame using a transformation matrix. A helper function creates a matrix given the initial pose P_{t-1} = [x_{t-1} , y_{t-1}, θ_{t-1}]. This matrix rotates and translates the particles based on the values from the robot's odometry. This transformation matrix (A) is shown below.

$$A = \begin{bmatrix} cos(\theta_{t-1}) & -sin(\theta_{t-1}) & 0 & x_{t-1} \\ sin(\theta_{t-1}) & cos(\theta_{t-1}) & 0 & y_{t-1} \\ 0 & 0 & 1 & \theta_{t-1} \end{bmatrix}$$

- **Update:**

  (Authored by Franklin Zhang)

  The odometry data (ΔX) are translated and rotated by the transformation matrix (A) to produce an updated list of particles (P_t). This operation is shown in the equation:

$$A(P_{t-1})\,\Delta X = P_t$$

Equation 9

- **Implementation Structure:**

  (Authored by Franklin Zhang)

  The Motion Model is a class that is initialized inside the particle filter node. The node is subscribed to the odometry topic. Whenever the odometry topic updates, the node evaluates the new set of particles given the odometry data and the previous set of particles. The output of the motion model replaces the previous particles in the particle filter node. This flow of operations is shown in Figure 8.

  ARGUMENTS:

  1. Odometry: The pose of the vehicle measured by the vesc speed controller
  2. Particles: A list of particle poses stored in the particle filter

  OUTPUT:

1. Updated Particles: Transforms the particles using a transformation matrix created using the odometry data.

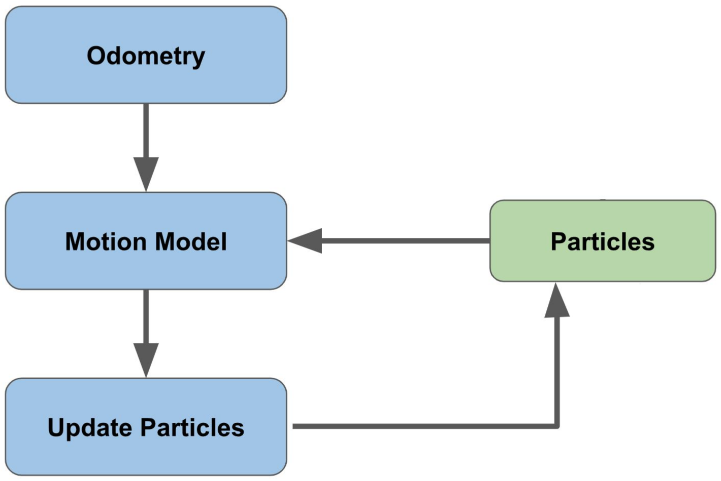

Figure 8: The flow of the particle pose updates using the motion model. The Motion model takes in odometry data and the previous set of particles and returns a new set of particles.

- **Evaluation:**

(Authored by Hector Castillo)

Selecting the standard deviation values for the noise determines how strict of loose the particles will fit the robot's actual pose. A strict model will represent the robot's actual position more accurately, provided minimal error. A more loose model will allow for more exploration and account for a wider range of error. Our proposed method for modeling the error in the robot's odometry was to set up a test in which the robot drives forward 1m, and the robot's actual ΔX is recorded. This, however, would require several trials in order to extrapolate meaningful standard deviations, and the error in measuring the robot's ΔX may even exceed the robot's error in odometry. Thus an error of 5% was determined as the maximum allowable error in the system, before a hardware change would be necessary. This set the standard deviations to $\sigma = 0.025$ to generate noise around 1.0. The result is a noise factor between 0.95 and 1.05 within 95% confidence which is them multiplied to x, y, and θ, each of which gets its own generated noise.

A quantitative evaluation and optimization of the base level noise was attempted using simulation tools. In simulation, the robot drives at 0.5m/s, and /odom is publishes at 50hz. The motion model runs every time /odom is published, so it loops at 50hz as well. Thus, the average Δx for the robot in simulation is 1cm (0.01m) at each time step. In order to add noise within 5% of the robot's Δx, the standard deviation would have to be $\sigma_x = 0.00025m$. This value was tested in simulation and analyzed through video analysis, but no measurable spread was found.

The next step was to test various standard deviation values for noise in x and compare their percentage spreads, a quantitative measure of how much the particles spread in a certain direction, the x direction in this case, over a given distance. This value is calculated by dividing the change in width of the set of particles in the direction of interest over the distance the robot traveled. The results are plotted in Figure 9.

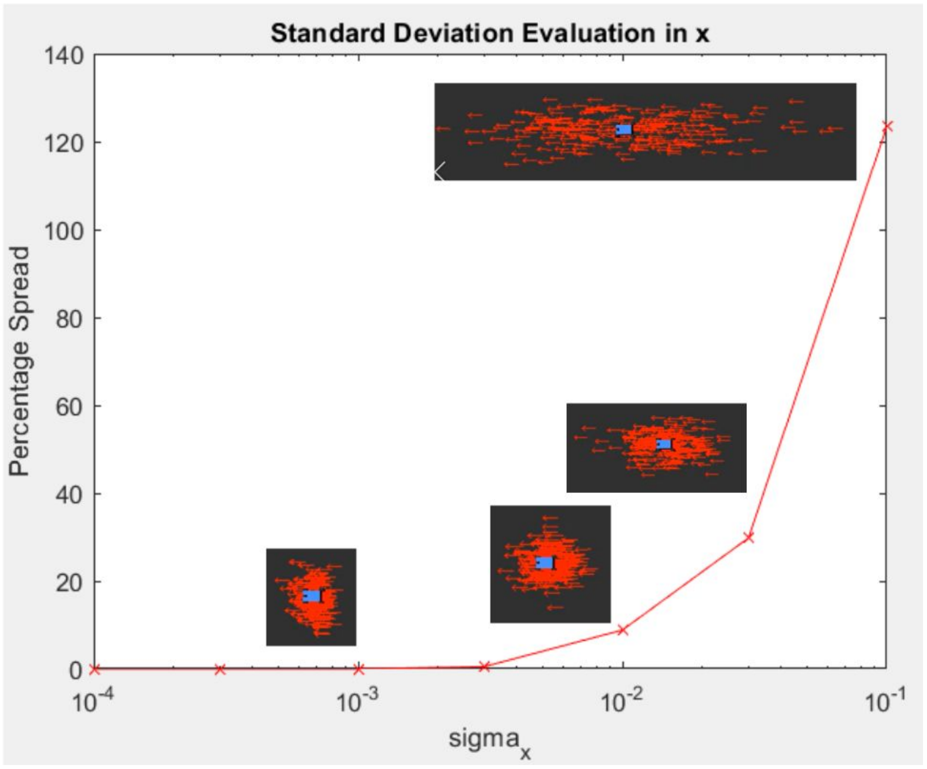

Figure 9: Plot of percentage spread values measured at several standard deviations for adding noise to *x*. Percentage spread is a quantitative measure of how much the particles will spread in a certain direction over a given distance. This value is calculated by dividing the change in width of the set of particles in the direction of interest over the distance the robot traveled.

A standard deviation of 0.01 was found to provide about 5.8% spread. Similar evaluations were performed for y and θ, resulting the in the following values being used.

$$\sigma_x = 0.01m \qquad \sigma_y = 0.005m \qquad \sigma_\theta = 0.01rad$$

The primary motivation for using values that result in this much spread is that the particle mean will still track well with the actual robot pose while allowing for exploration and robustness of the algorithm.

2.3. Integration

(Authored by Mitchell Guillaume)

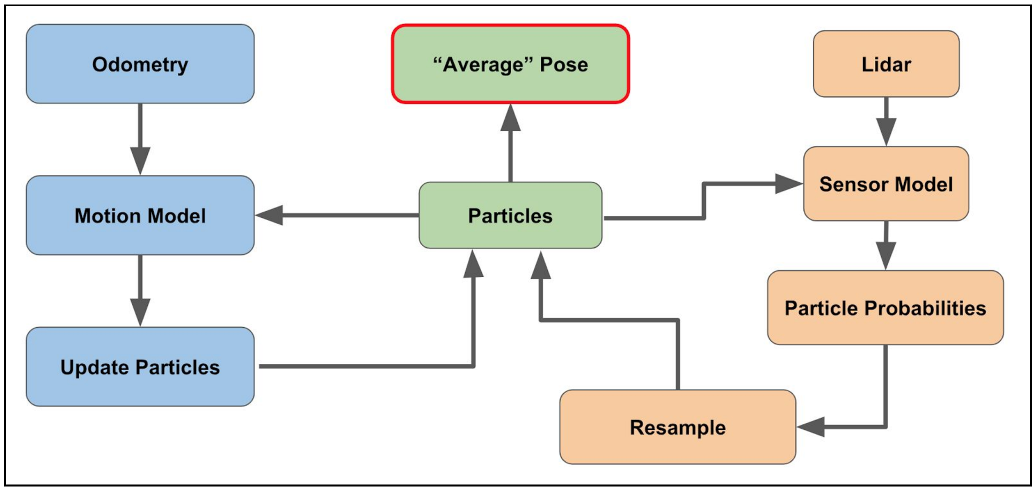

Figure 10: Particle filter architecture. Orange blocks correspond to the sensor mode thread, and blue blocks correspond to the motion model thread. Green blocks are shared resources, or values based on shared resources.

■ Approach

The particle filter is essentially two separate processes running concurrently on separate threads. One process uses the car's odometry and the motion model to update the particle positions as the robot drives. Because of the noise in the motion model, the motion model will cause the particles to diverge and spread out as the car drives.

The other process feeds data from the lidar into the sensor model to calculate which particles are most likely to be correct. Then, the particles are resampled. Less likely particles are eliminated, and particles are added in more probable positions. In short: the motion model pushes particles out, and the sensor model pulls them in.

This might seem relatively straightforward, but there's a catch. Both processes are updating the particle positions, and the two processes are running at different speeds (based on sensor sample rate and computation time). These two processes will conflict if they try to update the particles at the same time. To account for this, each process has the ability to "lock" the particles variable using the Lock() object from the threading module.

The architecture of the integration can be seen in Figure 10. When motion model receives new odometry data, the model is applied to each particle to update their positions. Whenever the sensor model gets new lidar data, it is applied to each particle to find how likely each particle is. Using those particle probabilities, the particle resampling process updates the particles such that they are in more probable locations. The state estimate is the average pose of all of the particles, discussed in greater detail in section 2.1.1 on page 7.

- **Evaluation**

Our sensor model tracks 250 particles, with 25 lidar beams per particle. These values gave us a good balance between computational speed and accuracy and robustness of our particle filter. When running our implementation on the robot, we observed a stable 35 Hz particle update rate. A cross track error plot can be seen in Figure 5. In addition to this quantitative evaluation, our team performed lots of qualitative evaluation by driving the car in different areas of the stata basement, and by driving very aggressively and sporadically. In all cases, the particle filter was able to converge on our car and remain so for the duration of our driving. An example of this qualitative evaluation can be seen in Figure 6.

2.4. SLAM with Google Cartographer

(Authored by Claire Traweek)

As a final challenge for this lab assignment, we decided to implement a working SLAM scheme for our robot. We installed Google Cartographer using catkin build and the provided cartographer_config files. Following installation, we found that cartographer was not working as expected; the laser scan was consistently offset from what was expected (see Figure 11).

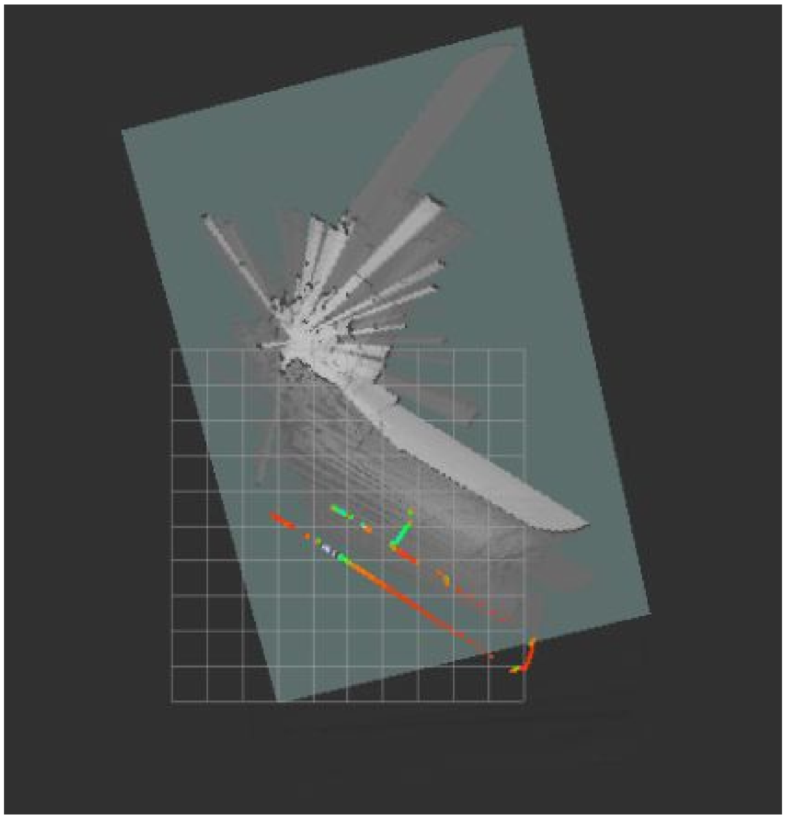

Figure 11: The laser scan data drifted consistently downward as the robot drifts forward. We see this above, as the robot drives straight down a hall. Watching the scan data points, it is clear the robot is progressing down the hall, but because it appears to Cartographer that the robot is driving about 60 degrees from straight, a messy map is produced.

We addressed this problem by modifying the Velodyne files; instead of performing scan transformation in Python and publishing to a new node, the transformation is done as the data is received and published directly to the /scan topic. To do this, the velodyne/velodyne_laserscan/src/velodyne_laserscan.cpp file was directly edited and saved on the racecar.

The resulting maps still had a lot of drift. After observation, we determined that this was due to a “fake wall” being observed in front of the car due to the low angle of the lidar. This can be seen in Figure 12. We were able to reduce the impact of this problem by adjusting the spring tension in the car’s suspension so that the lidar was closer to being level. Now, instead of beams sometimes hitting the floor at long ranges, they successfully point towards the horizon.

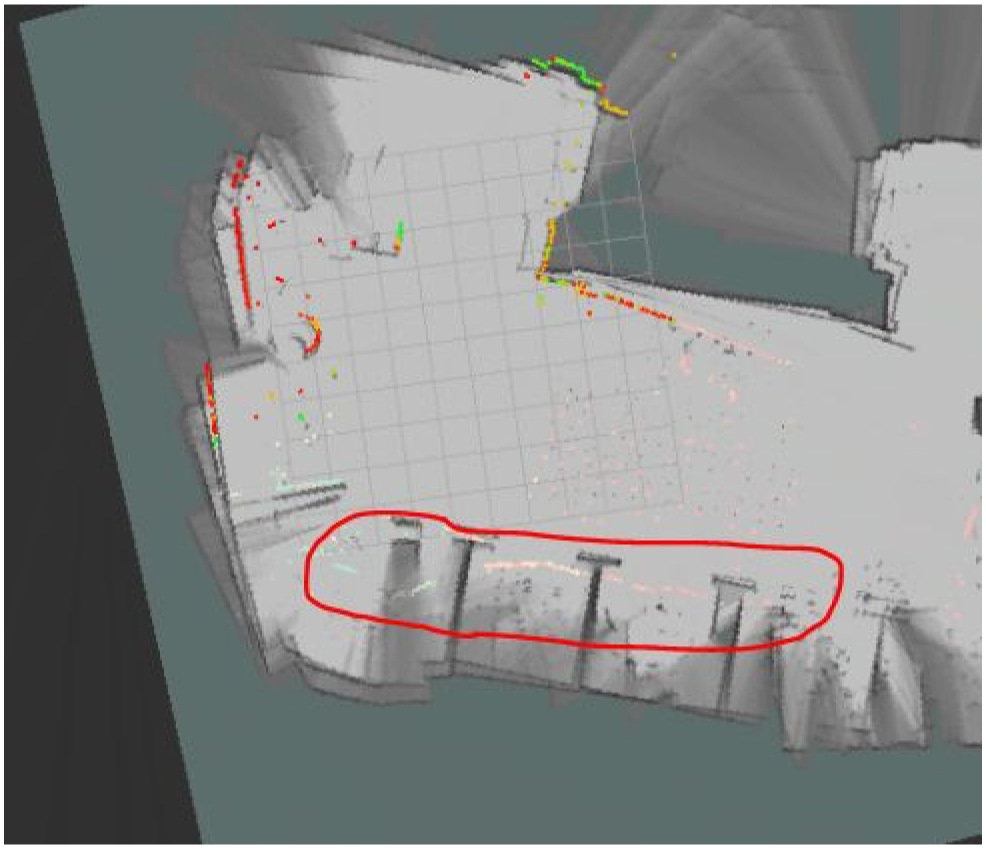

Figure 12: An example of a false wall generated by the lidar scan hitting the floor in front of the robot. This effect was corrected by adjusting the height of the robot such that the lidar was level.

Eventually, with a great deal of work, we were able to produce fairly accurate maps (see Figure 13), but were still experiencing some drift. We hypothesize that this is due to a combination of the car's tendency to drift to the left when driving straight and imperfect tuning.

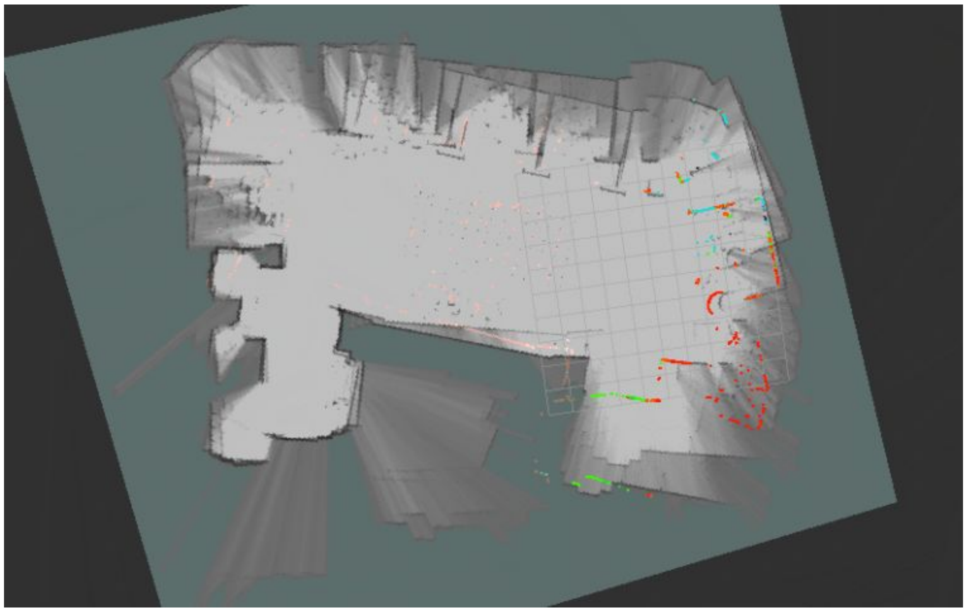

Figure 13: Example of a map drawn using Google Cartographer. The map is not perfect, but we hypothesize that it can be improved by tuning the robot

3. Lessons Learned

3.1. Technical

- We learned how to implement a localization algorithm such as the particle filter.
- Learned how to use Flame graphs to analyze and reduce run times
- Learned about a typical distribution of Lidar reading
- Learned about Google Cartographer

3.2. Team and Communication

- Learned to how to communicate when some of us are behind or confused.
- Learned how to address issues between team members professionally and openly

3.3. Individual Lessons Learned

“I learned how to create a motion model in a matrix in a matrix implementation. It also introduced to me a new approach to localization. I’ve seen Kalman Filters before, but didn’t realize that a particle filter can also be used for localization tasks” -*Franklin*

“I learned a lot about the ROS debugging process, what the errors actually mean, and what a lot of the random files that are always there do” -*Claire*

“This lab helped me to better understand the fundamental connection between exteroceptive sensing and its role in constructing a probabilistic model of a robot’s state” *-Danny*

“This lab gave me a look into the world of localization and mapping. I wasn’t aware that there were so many different techniques, and I enjoyed learning about the trade-offs of the different approaches.” *-Mitch*

“I learned that there are ways to simplify a complicated problem in order to get meaningful results which not wasting time on more tedious methods. Specifically, I figured out how to use video analysis to evaluate particle spreading and choosing a standard deviation that way rather than trying to experimentally calculate a more accurate standard deviation through long and tedious testing.” *-Hector*

4. Future Work

(Authored by Claire Traweek)

4.1. Car Tuning

During this lab, we realized that the performance of the car improved when the front of the car was slightly elevated such that the lidar scan doesn’t hit the floor at longer distances. We also noted (but were unable to correct) that the car drifts hard left when commanded to drive straight. In the future, for labs that rely on odometry particularly, we are going to work on both finding a hardware solution as well as potentially modifying our software correct for this inaccuracy.

Additionally, the cartographer results in general can be improved by more thoroughly walking through the “Algorithm Walkthrough for Tuning” section of the documentation.

4.2. Code Optimization

Although we achieved a stable 35 Hz update rate for the robot’s position during particle filter operation, which was entirely sufficient for the requirements of this lab, the performance could most likely be significantly improved through further optimization. This would probably require a restructuring of the way that our code operates and would rely heavily on the use of profiling tools like cProfile. Additionally, large improvements in speed could be realized by writing all of the programs in C++ rather than Python.

This document was revised and edited by Daniel Wiest and Mitchell Guillaume.