Localization in the Stata Basement using a Particle Filter Algorithm

Ayden Johnson Erin Menezes Gregory Pylypovych Jose Ramirez Gloria Zhu

I. Introduction (Ayden)

The goal for this lab is to create a localization system for the racecar. Localization is the process of identifying a robot's position and orientation within a given map. Localization is essential for a robot to navigate in an environment towards a specific target. Self-driving cars, for instance, need localization with respect to a city map to plan a route to drive along, and warehouse robots must be localized within the building map to move towards tasks to complete. Localization can only be carried out when a pre-existing map is provided to the robot, but the robot will be able to scan its surroundings to identify its pose on the map, without being given its initial pose beforehand. Additionally, the localization process is continuous, so as the robot moves around its understanding of its pose is automatically updated.

For this lab, an algorithm was designed so that the racecar accurately localizes itself within the Stata Center's basement, given a map of this area. It is able to maintain this accuracy as it drives around to different parts of the basement. The sensor data it receives is expected to be prone to errors, so ideally the robot should make an estimate of its pose that is as close as possible to its true pose.

II. Technical Approach (Erin)

This lab focused on a localization algorithm, the particle filter, that uses two system models as well as a given map to localize a vehicle. More specifically, using scans from a LiDAR sensor as well as a known map of the Stata basement, the particle filter algorithm was used to localize the vehicle in the Stata Center's basement. This algorithm utilized a Monte Carlo simulation to distribute particles (expected vehicle poses) around the vehicle, assigned probabilities to each particle, and then resampled these particles so that the wider distribution of particles with lower probabilities converged to a smaller distribution with higher probabilities of being the real vehicle pose.

To update these particles, two system models were used. The motion model uses odometry data to compute the vehicle's pose in the next time step, while the sensor model assigns probabilities and resamples accordingly to determine which particles are more likely to be the real pose of the vehicle. Together, these models combine into the particle filter algorithm that allows the vehicle to be localized in the basement map as it moves around in real space.

A. The Motion Model (Erin)

The motion model is one of the most basic ways to predict future poses of a vehicle. It takes in odometry data to estimate the pose in the next time step. This odometry data (real-time linear and angular velocity) can be calculated from the integral of the acceleration outputs of an IMU (inertial measurement unit), wheel encoders, motor input power, etc.. The motion model used in this localization algorithm takes in the vehicle's odometry from the /odom topic, as well as a list of particles, pose estimates of the vehicle. For each particle, which is inputted as a pose (X, Y, and $/theta$) with respect to the world frame, the next time step's particle is computed, again as a pose with respect to the world (Fig. 1).

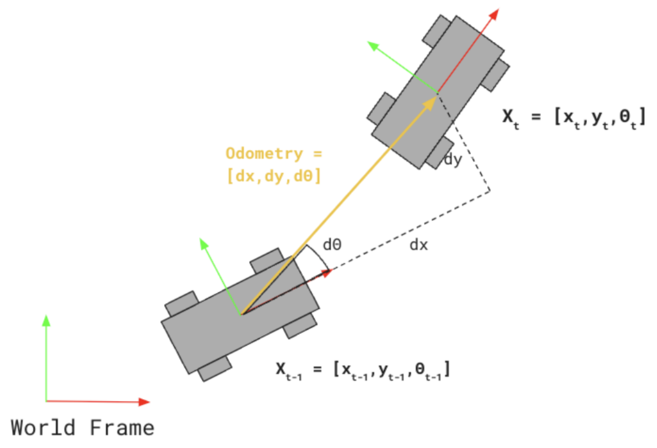
Odometry = [dx,dy,dθ]
$X_t = [x_t, y_t, \theta_t]$
dy
dθ
dx
$X_{t-1} = [x_{t-1}, y_{t-1}, \theta_{t-1}]$
World Frame

Fig. 1. The motion model uses an inputted odometry and current vehicle pose to calculate a new vehicle pose in the next time step. The vehicle poses are determined in the world frame, while the odometry is calculated from the current vehicle's frame.

However, the odometry data is inputted as changes in the pose (dX, dY, and d$/theta$) with respect to the particle's current pose, and therefore has to be transformed to the world frame to be able to be easily used. A rotation matrix was used to rotate the odometry vector, and this was then added to the particle's pose. This produced a new pose with respect to the world frame, which is the particle's pose for the next time step (Fig. 2). This process was implemented on all the particles to determine the pose of the particles in the next time step.

Finally, we added some noise to each particle, both to ensure that they would spread out and cover more ground as well as account for and "write over" any noise inherent to the IMU sensor's odometry measurement.

$$\underbrace{\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix}}_{\text{Next pose, in the world frame}} = \underbrace{\begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix}}_{\text{Current pose, in the world frame}} + \underbrace{\begin{bmatrix} \cos(\theta_{t-1}) & -\sin(\theta_{t-1}) & 0 \\ \sin(\theta_{t-1}) & \cos(\theta_{t-1}) & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Rotation matrix to convert odometry vector to the world frame}} \times \underbrace{\begin{bmatrix} dx \\ dy \\ d\theta \end{bmatrix}}_{\text{Odometry}}$$

Fig. 2. The following equation is used to calculate the vehicle's pose for the next time step, given the odometry and the current pose. Since the odometry is not provided in the world frame, it must be transformed to be a vector in the world frame before being added to the vehicle's current pose.

B. The Sensor Model (Jose)

The sensor model is used to locate where the vehicle currently is on the map by using the motion model, map, and sensor readings (Fig. 3). It starts by initializing particles at possible current locations according to the motion model's predictions. It then collects the data from the map as to what the expected sensor readings at those positions would be and compares that data to the sensor readings that the robot is currently reading. It uses those readings to compute the probability that the vehicle is currently at that particle in that pose and then places the vehicle at the particle with the highest probability (Fig. 4).

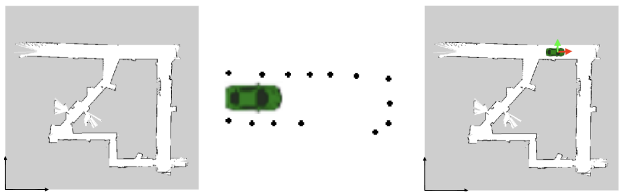

Fig. 3. The robot is given a map, a last known location, and a set of sensor readings. By comparing the known readings to the expected readings, it can figure out its location.

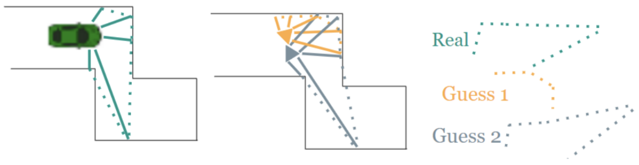

Fig. 4. For Monte Carlo localization, the robot creates a set of particles randomly placed around its last guessed location. It then compares the simulated LiDAR ranges to the measured LiDAR data to calculate the probability that the robot collected its data at each of the particles. It will then decide which particle the robot will most likely be at.

The probability at each pose is a product of the individual probabilities of each range at that certain point as according to (1). The equation states that the probability that the list of distances z_k is measured when the true distances are x_k with a map m, is equivalent to the product of the probabilities of each individual measured distance $z_k^{(i)}$ given a true distance $x_k^{(i)}$.

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

However, the probability of each individual distance has to take into account 4 sections:

- measuring the right distance
- having the measured distance be shorter than the actual distance
- having the measured distance be longer than the actual distance
- having a random measurement

The first section is represented by (2), which is a Gaussian distribution around the right distance. As we get closer to the true value $x_k^{(i)}$, we expect the value of $p(z_k^{(i)})$ to rise. However, due to randomness, we expect there to be some deviation, which leads to the Gaussian distribution.

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

The second section is now represented by (3), which is the likelihood of an unknown obstacle intercepting one of our laser readings, and thus giving back a shorter reading than expected. It is a linearly decreasing line, as the farther the laser gets, the less likely it is to be intercepted (if unknown obstacles are assumed to be randomly distributed around the map).

$$p_{short}\left(z_k^{(i)}|x_k,m\right) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} \\ \text{if} \quad 0 \le z_k^{(i)} \le d \text{ and } d \ne 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Section 4 is calculated through (4), which occurs if a LiDAR beam bounces off an object, and does not return to the robot. To make sure that there is not infinite reading, the probability is capped at a certain distance with a large spike.

$$p_{max}(z_k^{(i)}|x_k,m) = \begin{cases} \frac{1}{\epsilon} & \text{if} \quad z_{max} - \epsilon \le z_k^{(i)} \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Section 5 is just to ensure that nothing is left out, and as such is a small constant value to make sure that the probability is never 0 at any distance.

$$p_{rand}(z_k^{(i)}|x_k,m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \le z_k^{(i)} \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

We then have to weigh and sum each probability to get the overall probability that depicts how likely a range point z_k at position x_k at time k is.

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

$$\alpha_{max}\cdot p_{max}(z_k^{(i)}|x_k,m) + \alpha_{rand}\cdot p_{rand}(z_k^{(i)}|x_k,m)(6)$$

where

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

In order to do that, the model must compute the probabilities for many particles in many positions, so computing that probability each time the sensor model is updated would create serious strain on the program. This has an easy solution, as the probability that the robot records a measured distance at a certain point depends only on the true distance, and the measured distance. As such, we can create a pre-computed table of probabilities depending on measured and true distances (Fig. 5). That way, when the robot is deciding which particle has the highest probability of being the true pose, it only has to do callbacks to the table instead of redoing all the calculations.

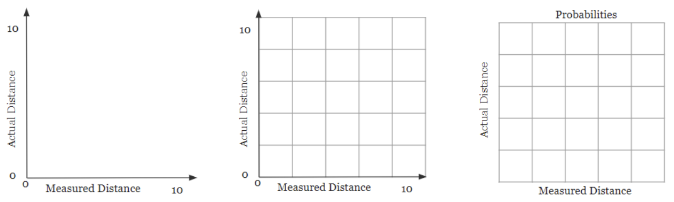

Fig. 5. In order to decrease the complexity of the program, precompute the probabilities according to the measured and true distances. First, break the distances into discrete sections, and then calculate the probability for each of the discrete areas. Those probabilities can then be put in a table to be used later.

C. The Particle Filter (Gloria)

Finally, we put it all together to implement the particle filtering algorithm (Fig. 6). We start by initializing a set of normally distributed particles at a user-broadcasted point (with stddev = 5.0). Then, the particles were updated by both the incoming odometry data (using the motion model) and LiDAR data (using the sensor model). We were able to regulate the speed at which the particles are updated, and we found that 30Hz and 20Hz worked well for the respective models. Our current pose guess was then recalculated as the new mean of all of our particles.

The odometry and LiDAR data work together to update and regulate our particles and pose guess. With each motion update, the particles track to where they would be given that exact odometry twist, with some Gaussian noise added to each tracking. However, this dead reckoning approach is susceptible to accumulating errors quickly, as it does not incorporate any feedback from the robot. Thus, with each sensor update, we subsequently resample our particles to correct them according to what the robot is perceiving about its environment.

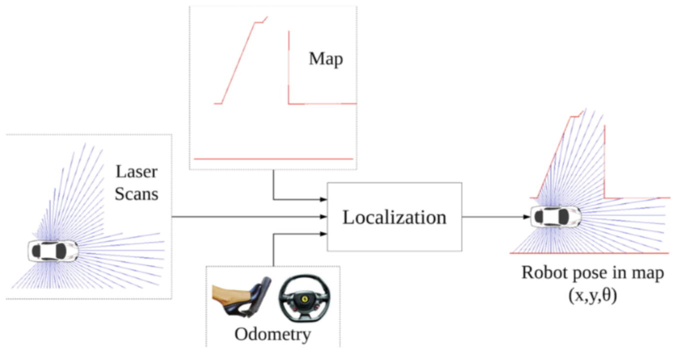

Fig. 6. Block diagram of incoming sensor and map data to our localization algorithm, and output estimated 3D pose (x, y, and yaw).

The motion model operation is faster and less costly, but it is more error-prone, while the sensor model is slower and more complex, but yields more meaningful results about our particles. By implementing the two together, we are able to achieve low-latency localization while maintaining high levels of pose estimation accuracy.

III. Experimental Evaluation (Erin and Gloria)

To determine the success of the particle filter algorithm, we used a series of metrics consisting of both quantitative and qualitative measurements.

A. Vehicle Position Accuracy (Erin and Gloria)

Our particle filter is able to quickly and accurately localize a static robot position, including when the robot is perturbed by small arbitrary amounts. This is important because the particles represent poses that are associated with likeliness probabilities, so in order to adapt to changing robot positions, the new position must be located within our previous particle point cloud.

Additionally, our particle filter algorithm successfully tracked a moving car with high accuracy. We tested this by running a wall-following algorithm in simulation, with the car moving at constant velocity. In Fig. 7, the red line (representing the average position of our particles) correlates precisely and accurately with the blue line (representing the ground truth position of the vehicle).

Quantitatively, we measured our accuracy using the distance error between the particle average position and the ground truth position (Fig. 8) and found that our average error was around 1.34 meters. Although this seems high when compared to Fig. 7 visually, the majority of this error actually occurred in the x-direction in the robot frame; essentially, the particle guess would “bounce” in front of and behind the racecar, but would still be located on the overall racecar trajectory. However, if the vehicle were to stop at an arbitrary time step, the particle guess may actually be located slightly in front of or behind the car.

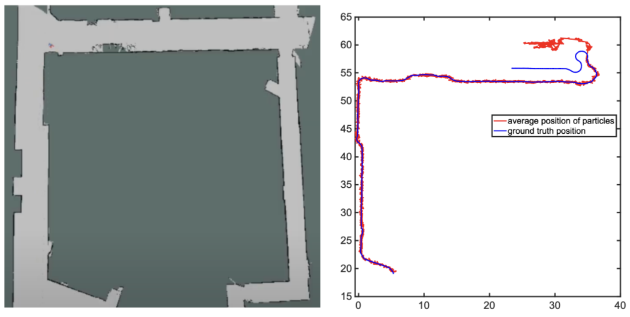

Fig. 7. L) A map of the Stata basement; R) the blue line shows the ground truth position of the vehicle in simulation, while the red line is the average position of the particles. The red line is almost always directly on top of the blue line, meaning that the average position of particles, the estimated position of the vehicle, was accurate.

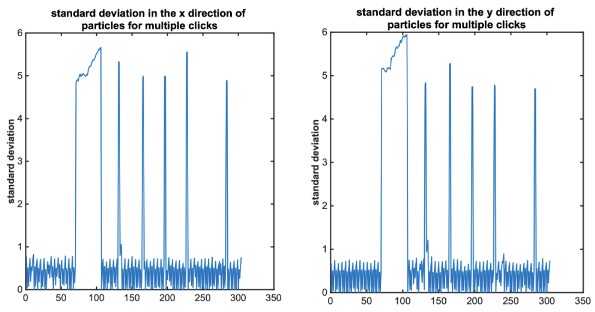

Fig. 9. The convergence rate, evaluated along the X and Y direction, was a maximum of four time steps. Each peak represents a new published point, which reset the particle distribution to a standard deviation of around 5. On average, the particles converged with 2-3 time steps.

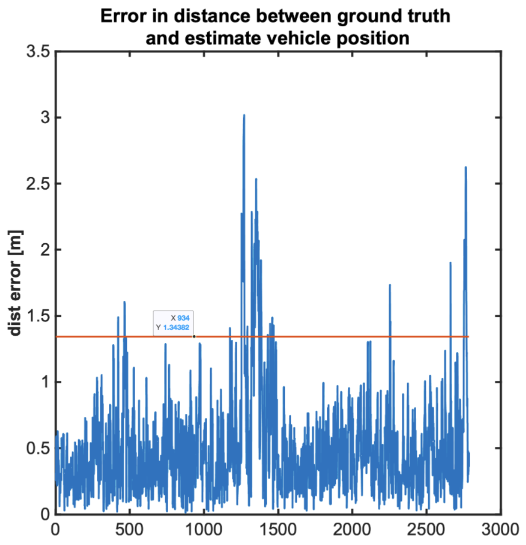

Fig. 8. The distance error between the ground truth of the vehicle and the estimated vehicle position (average position of the particles) is shown in the graph, including the average error of 1.34 meters.

B. *Particle Convergence Rate (Erin and Gloria)*

We also evaluated the convergence rate of the particles to determine the success of our particle filter algorithm. Our goal is to converge as fast as possible (e.g. picking the correct particle pose instantly). In practice, we found that our particles converged within 2-3 time steps, as seen in Fig. 9.

We always initialize our particles with a standard deviation of 5, so the peaks in the figures correspond to each time we clicked to re-initialize. Additionally, we see visually that the particle filter converges very rapidly, even with slight perturbations to the car, which is essential for tracking rapid movements and driving speeds.

C. *Noise in the Motion Model (Erin and Gloria)*

Finally, we evaluated our motion model with and without the addition of simulated Gaussian noise. When we ran the algorithm without noise, all of our particles converged into one pose extremely rapidly. However, the accuracy of this pose was very variable, and once the particles converged, they stayed the same forever. In Fig. 10, the red arrows depict the particles' converged points, which varied greatly between two simulation runs in which the vehicle's position was the same and the particle initialization point was very similar. Note that the single red arrows shown are actually all the particles in the same pose, stacked on top of each other. As they had "collapsed", the particles were unable to fix their position in future time steps.

However, once noise is added, as in Fig. 10, the particles cover an area around the estimated point, rather than a single point. This variance allows the sensor model to adjust the particles every time step depending on their probabilities, as long as the vehicle is within the particle range.

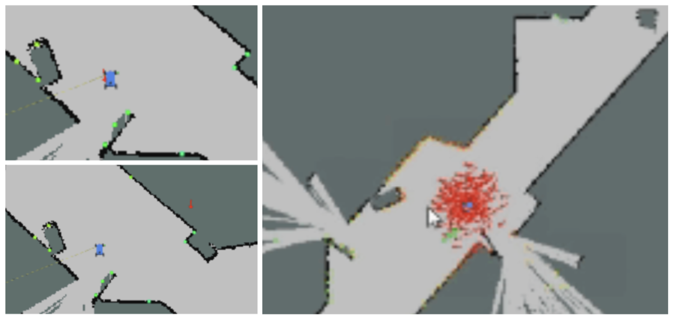

Fig. 10. The motion model without noise resulted in a single point of convergence for the particles, as in the left two images. With no motion of the vehicle, two instances of publishing a point resulted in very different convergence points. However, with the addition of noise (right image), the points are able to accurately converge around the position of the vehicle.

While the convergence rates between the noise and no noise motion models are approximately the same, the noisy model allows for more samples and variation, which helps "fine tune" the estimated position of the vehicle.

IV. Conclusion (Greg)

In this work, we verify the effectiveness of particle filter localization from LiDAR data in simulated indoor environments. We additionally verify that the localization system can work on the racecar, albeit while highlighting that our current localization algorithm struggles with global localization, latency, and significant sensor noise.

A. *Future Work (Greg)*

Expanding on our localization system will entail specifically addressing the global localization challenge and adapting to the types of sensor noise our racecar experiences. We can make progress on our system by specifically adapting our sensor model priors to reflect our racecar's LiDAR noise, and taking a disciplined approach to global localization through a search algorithm. Cost-efficient solutions to both of these issues are still open problems in real-world robotics, bottlenecking safety and precision in autonomous vehicles, locomotion, and manipulation. While our racecar's form factor is substantially different from many of these systems, developing a simple solution to our case may lead to promising applicaitons in the analogous problems in other areas.

B. *Lessons Learned (Ayden)*

In this lab I gained a better understanding of how the Monte Carlo localization algorithm works, and what it looks like in practice. I also am more familiar with using ros2 to run simulations of the racecar and displaying live data of the real racecar in action.

C. *Lessons Learned (Jose)*

Learned more about how localization worked with sensors. I've done some work with motion models, but it's good to lower the uncertainty of those by collecting more data. Also learned about how Monte Carlo can be applied to localization, as I had not heard before of combining the two. I learned how annoying it can be to make graphics on Google Slides, and I wish I knew of a better image editing program. Also how complicated it can be to use threading.

D. *Lessons Learned (Erin)*

This lab helped me understand what localization means, in terms of using sensors on a real world robot to determine its location on a map. However, I found it odd that the map had to be given, considering in more fluid contexts, like on a road, the map is constantly changing (new roads, new signs, new buildings, people, etc.). I wish we had more time in the lab to actually implement SLAM, since it is a more relevant localization and mapping algorithm. I also learned the effects of a little bit of noise in a system, learned how to implement a particle filter for more constant maps, and how to determine, gather data for, and analyze metrics for an algorithm. Finally, this lab gave me practice in creating slides and a report for a technical audience.

E. *Lessons Learned (Gloria)*

Through this lab, I learned more effective debugging strategies, including how to break down a large program into various smaller parts to unit test. Additionally, I learned about localization and SLAM filters, and the difficulties of going between sim2real. It was also cool being able to use prior labs (wall follower) to test the localization simulation.

F. *Lessons Learned (Greg)*

This lab brought me to further improve how fast my iteration cycle is from modifying code to testing on the racecar. Through trying to deploy our algorithm to the racecar, I got better picture of what kinds of real-world issues our algorithms will face when deployed in the final competition.