Lab 5 Report: Localization

Team 18

Eric Delgado, Arianna Ilvonen, Jesus Diaz, Charles Ge, Dora Hu

1: Introduction [Jesus Diaz]

Localization is the process of determining the position and orientation of a robot within its environment. In the context of autonomous self-driving systems, it is especially important, because such robots rely on an accurate representation of their *pose*–their position and orientation–to successfully navigate in their environment. Robotic localization using Monte Carlo methods, referred to as Monte Carlo Localization (MCL), is a probabilistic approach used to estimate the pose of a robot within its environment. The method is based on sampling from a probability distribution to approximate the robot's belief about its location. Initially, the robot's position is uncertain, represented by a set of particles randomly distributed across the map. As the robot moves and gathers sensor data about its surroundings, such as the LiDAR data collected in our case, these predictions are updated based on motion and sensor models. Through repeated iterations of prediction and correction, the predictions converge towards the robot's true pose, providing an accurate estimate of its localization in the environment.

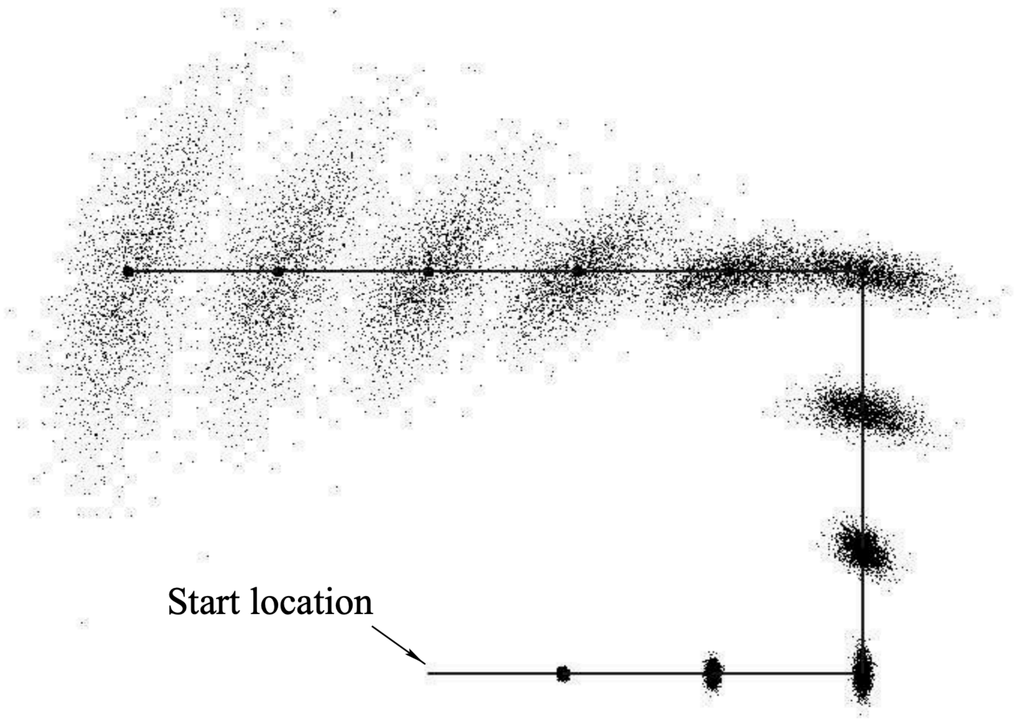
In this lab report, we will discuss our technical approach to the problem of localization, describe our implementation of the MCL algorithm, and evaluate our findings on the performance of this algorithm in accurately predicting the pose of our robot in simulation and in a real-world environment.

2: Technical Approach [Jesus Diaz, Eric Delgado, Charles Ge, Dora Hu, Arianna Ilvonen]

We implemented the MCL algorithm in the form of three modules: a probabilistic kinematic model or *motion model*, a sensor model, and a particle filter. Collectively, these modules relied on (1) a given map of the area where the robot is present, (2) odometry data with our relative position and orientation, and (3) LiDAR data with distances to surrounding obstacles. Of these, a map of the tunnels in the Stata Basement was provided for us, and the odometry and LiDAR data were provided by real-time measurements from the robot itself. We explore the purposes and implementations of each of these modules below:

2.1: The Motion Model

In the MCL algorithm, the state of a robot is represented by a set of particles, each representing a prediction of the robot's pose at a given point in time. As the robot moves through the environment, the motion model will use the odometry data to update these particles accordingly by adding the change in pose from each time step to each particle. Odometry data is assumed to be subject to noise and inaccuracies in measurement, which we can account for by adding a normal distribution of noise to the particles. Thus, by itself, this model accumulates uncertainty the longer the robot is left to run from its original pose:



[1] Figure 1: Example of motion model's effect on particle distribution in isolation. Pose estimation of the robot's location spreads out due to accumulated noise.

In our implementation, we were able to pull odometry data from the "/odom" (in simulation) and the "/vesc/odom" (on hardware) topic to make these predictions and the numpy library to generate a normal distribution for our noise. We added noise along two components — translation and rotation. Each odometry input was broken down into a translational movement in the robot's current frame plus a rotation. These data points were then augmented with the translational and rotational components of the noise, respectively. The noise was gaussian and both the translational and rotational noise was centered around 0 with a standard deviation of 0.05. It is worth noting that other motion models exist, most notably the velocity motion model which uses the velocity data instead of odometry to make predictions about the robot's location. Odometry data is only available retrospectively, after the robot has moved, making it unsuitable for guiding the robot's actions in real-time. Luckily, this is not necessary in the context of the MCL algorithm, and seeing as how the odometry motion model is able to provide us with data that is usually more accurate, it is a better choice for our MCL implementation.

2.2: The Sensor Model

In order to combat the aforementioned noise from our motion model, we implement the sensor model. The sensor model evaluates the probability a ground truth observation z_k is generated from a hypothesized sensor reading x_k within a known map, for all particles in the model. It does this by first discretizing all possible scans into a 2-dimensional matrix of probabilities - with measured LIDAR scans along one index and hypothesized particle scans along the other. This discretization vastly reduces computational overhead for the particle filtering algorithm because it avoids recurring probability computations at each step of the sensor model. This choice of discretization comes with the sacrifice of losing a small amount of accuracy due to modeling the probability distributions as discrete, rather than continuous. Our model uses a 201-by-201 sensor table, which is discretized enough to capture the probabilities well enough for our purposes. It populates this matrix using these formulas:

$$p_{hit} = \frac{1}{\sqrt{2 * \pi * \sigma_{hit}^2}} * e^{\frac{-z_{ki} - d^2}{2 * \sigma_{hit}^2}}$$

$$p_{short} = \alpha_{short} * \frac{2}{d} * (1 - \frac{z_{ki}}{d}) \text{ if } z_{ki} \leq d \ \& \ d \neq 0, \text{ else } 0$$

$$p_{max} = \alpha_{max} \text{ if } z_{ki} = z_{max}, \text{ else } 0$$

$$p_{rand} = \frac{\alpha_{rand}}{z_{max}}$$

$$p_{total} = p_{hit-normalized} + p_{short} + p_{max} + p_{rand}$$

Figure 2: Overview of Formulas Used to Calculate Probabilities for the Lookup Table

With the parameters:

$$\alpha_{hit} = 0.74, \alpha_{hit} = 0.07, \alpha_{hit} = 0.07, \alpha_{hit} = 0.12, \sigma_{hit} = 8.0,$$

Figure 3: Overview of Parameters Utilized in the Construction of the Lookup Table Calculation

When plotted, these probabilities can be visualized as such:

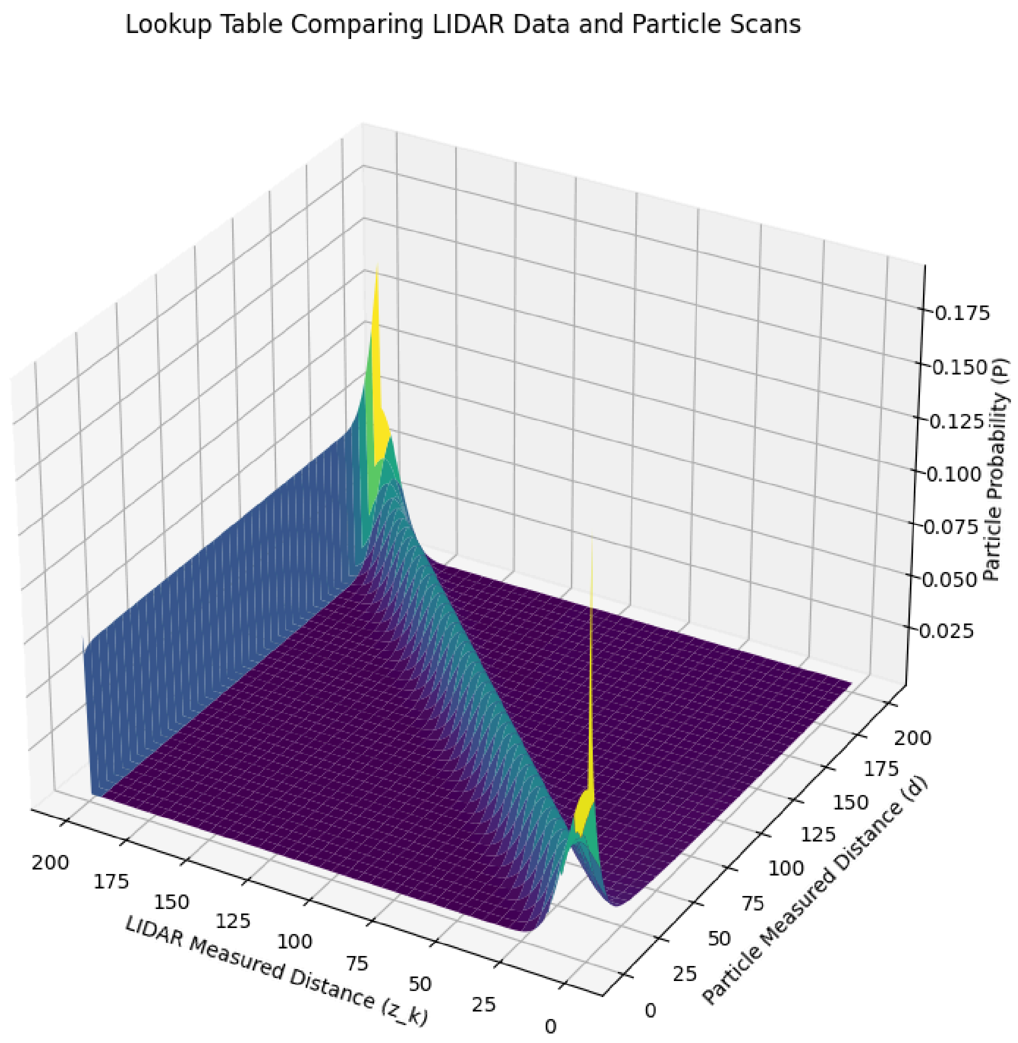


Figure 4: Probability distribution graph comparing ground truth observations with hypothetical observations.

Particle and LIDAR data is then converted from real-world meter measurements into pixel measurements using the map resolution and, using vectorized computations (for program speedup), the probabilities for each particle are quickly found and returned in the matrix.

2.3: The Particle Filter [Charles Ge, Dora Hu]

The particle filter estimates the car's actual pose by concurrently using both the motion model and the sensor model to update particles that estimate the car's location. We implemented the motion model and sensor model as classes in Python and implemented the particle filter as a ROS2 node. The particle filter node subscribes to the car's odometry and LIDAR scans and in its callbacks, calls functions from the motion model and sensor model respectively to update one of its parameters, an array of particles.

Particles are initialized randomly in a normal (standard deviation 1.0) translation distribution with a fully uniform random angular distribution around a pose estimate, upon which the particles are fed into the sensor model to compute probabilities for each particle. Particles are then resampled according to their respective probabilities, while ensuring that the number of particles stays constant. The particles are concurrently fed into the motion model, which copies the car's odometry and applies the transformation to each particle in its respective frame, along with a small, normally distributed noise for thorough coverage. The particles are constantly being re-cycled through both the sensor and motion models to converge upon the true position of the car.

Since the callbacks for the car's odometry and LIDAR scans are continuous, the particle filter concurrently calls the motion model and sensor model. However, since the motion model and sensor model both modify the same particle array belonging to the particle filter, it would be ill-advised for them both to be able to edit the particles simultaneously. Therefore, we implement threading to prevent more than one process from modifying the particle array at any given time. By using a lock around any code that modifies the particle array, we require a thread or process (in other words, either the sensor model or motion model) to access the lock to modify the particle array. If another process is modifying the particle array, the first process cannot access the lock and must wait for the other process to relinquish the lock as it finishes modifying the particle array. Lastly, we downsample the observed LIDAR scan data to ease computational load for the probability calculation since the actual robot LIDAR is more dense than needed for our desired pose estimation accuracy.

3: Experimental Evaluation [Dora Hu, Eric Delgado, Arianna Ilvonen]

After we succeeded in making the particle filter work in simulation, we began trying to replicate its actions on the actual car. This involved numerous differences between simulation and real life, such as simulated input and actual LIDAR scans as well as the RVIZ map and the frame of the real world. We approached the implementation of MCL on the actual car by first implementing the motion model and sensor model separately before combining them in the particle filter.

We began by visualizing the results of our computations in the motion model and sensor model to ensure that our individual modules worked as expected. We visualized our particle array via a PoseArray to tell if our motion model could accurately capture the car's motion by moving the car with a controller and observing the change in position of the RobotModel representing the car as well as the particles.

Our goal for the motion model was to have the particles and car move from their starting positions along the cars measured odometry, so the particles end with the same change in pose as the car. The particles are initialized as random poses with x and y locations at or around the car's pose; when running the particle filter with only the motion model we could see the particle poses change with the same odometry as that of the car. Qualitatively, this can be observed in Figure 5:

the particle poses start near the car but spread out as the car moves more; when the car moves backward the particles backtrack to their original positions.

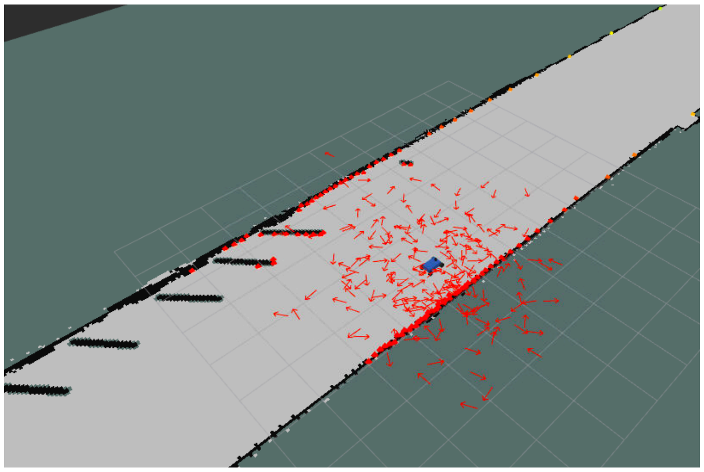
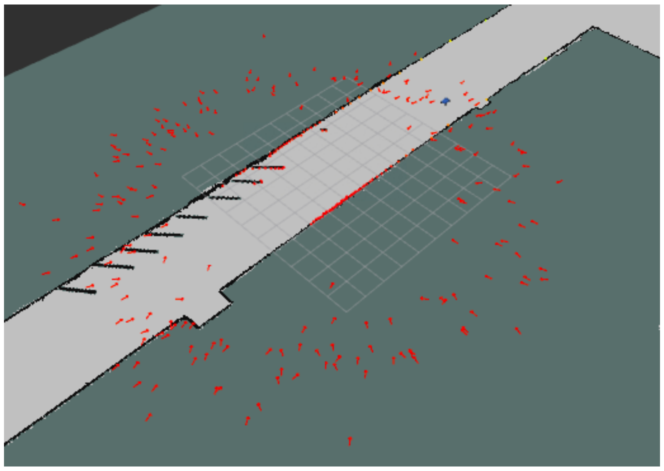
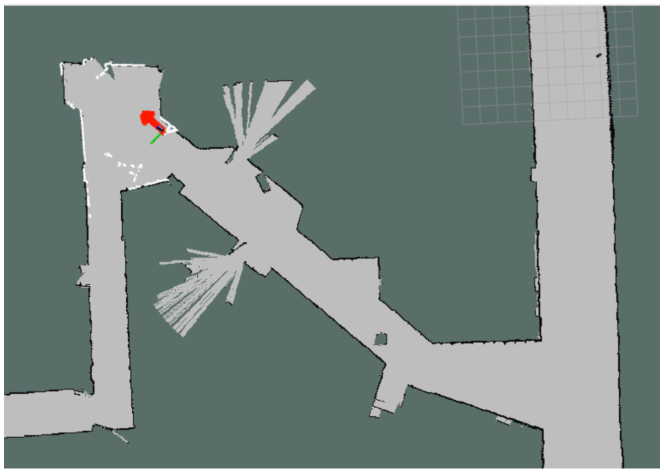



Figure 5: As the robot car moves further from its starting point, its particles move further from their starting points in the directions they initially point. Without the sensor model running, the particles do not converge to the pose of the car, so they are transformed with respect to their starting orientation. This causes them to spread out over time.

Our goal for the sensor model was to use LIDAR scans to evaluate the probability of each particle having the car's actual location as its own. This output would then be used by the particle filter to resample the particles to converge to the estimated car position. We qualitatively observed if this worked by observing the particles while running the particle filter with only the sensor model and confirmed that the particles converged, so the sensor model was working as expected. We also confirmed that our callback was correctly visualizing the LIDAR scans by seeing the LIDAR scans follow the walls as we moved our robot car around the Stata basement.

When combining the motion model and sensor model, we wanted the particles to initially spread out but converge to the location of the car, which can be observed in Figure 6.



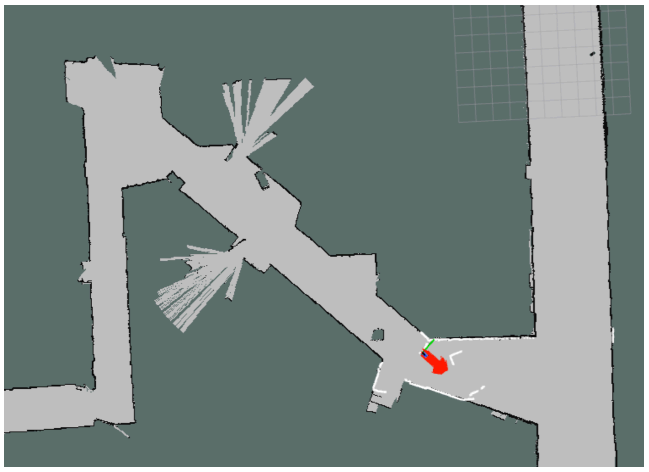


Figure 6: The particles (the amalgamation of red arrows) follow the robot car's position and point in its direction. This displays the working behavior of the sensor and motion model together, in which they correctly determine the global location of the car.

To test the particle filter on our physical racecar, we decided to record data of our localizer's published position and compare this to the integration of the odometry we received from our car. We chose this because it would, when plotted, be an easily digestible way of interpreting the path our localizer takes compared to the true ground position. Shown by both figures 7 and 8, The MCL pose estimate follows the general shape of the true ground position quite well, but is offset by a small quantity - this indicates that our system is quite precise but not completely accurate, something that will require more testing to resolve. Additionally, the pose estimate from odometry integration is not completely correct, since odometry measurements are subject to noise and integration error as we determine position changes from velocity. This comparison does suggest that our pose estimate from the MCL algorithm is close to correct, and does correctly take into account odometry measurements from the car.

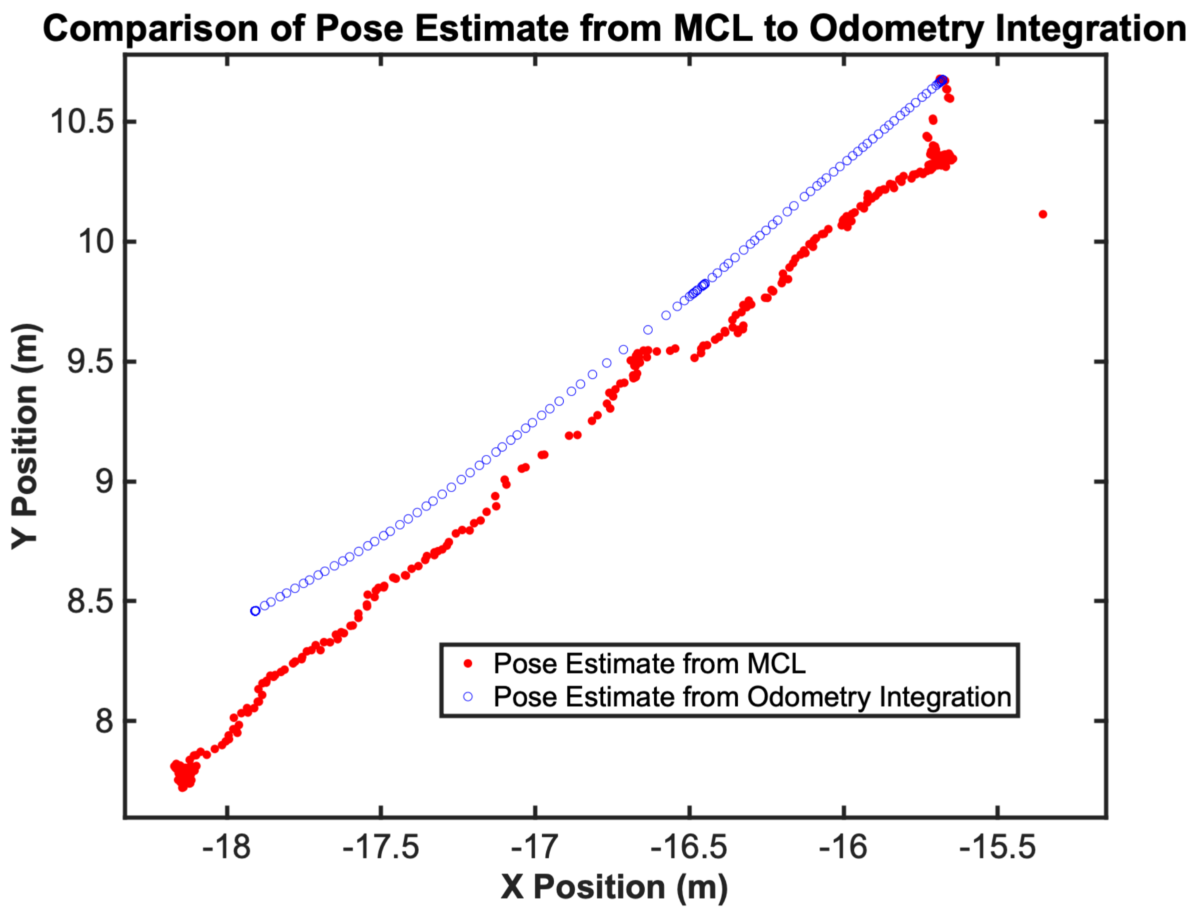


Figure 7: Comparative visualization of the race car's estimated position based on odometry against the averaged pose estimations from our monte-carlo localization. Although both measurements are not considered ground truth, they do suggest that the MCL algorithm is correctly taking into account odometry measurements from the car and accurately estimating global pose.

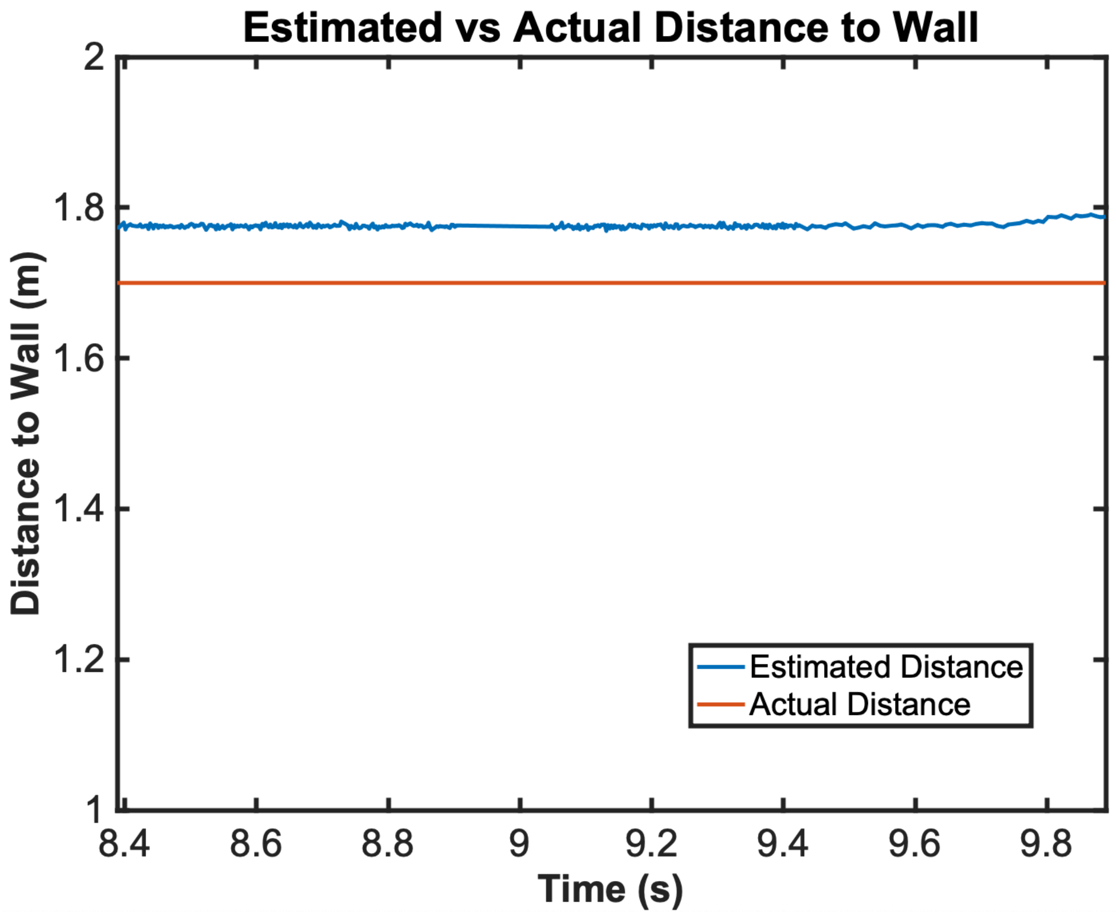


Figure 8: Comparative analysis of actual versus localized distance measurements of the race car to a wall during wall following. As actual distance was a ground truth measurement made in the real world, this set of comparisons suggests that our estimation method has a small degree of error, but is overall mostly accurate.

4: Conclusion and Lessons Learned [Jesus Diaz, Charles Ge, Eric Delgado, Dora Hu, Arianna Ilvonen]

With respect to the technical aspect of this lab, we learned the importance of code modulation in debugging. It was much more efficient to test each module separately and observe the individual behaviors to deduce which modules were failing to operate correctly, rather than debugging the entire system at once. It allowed us to realize unexpected behaviors in our code, and sequentially build up our program with robust modules. This also allowed us to conclude when there were problems present in the integration of the modules, and to slowly iron out our code until we eventually reached a satisfying solution.

This lab proved to be a significant technical challenge for all of us, and it forced us to rethink our team dynamic. Due to the challenging nature of this week's lab, we ultimately did not take our usual approach of dividing tasks among ourselves; rather, we worked closely on every aspect of our MCL algorithm, from the initial design of the motion model up to the painstakingly long hours of debugging that came with integration. This shift in our teamwork dynamic not only enabled us to successfully tackle this lab, but it also highlighted the importance of collective effort and adaptability when faced with a particularly challenging situation. Moving forward, we recognize the value of this experience in reinforcing the necessity of unity and flexibility in achieving our goals as a team.

5: Citations

- [1]S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. Cambridge, Mass.: Mit Press, 2010.