Team 15 + Irene Terpstra, Miles Silva, Matthew Leonard, Lili Sun

6.141

March 31, 2022

LAB 5 REPORT: Localization

1 Introduction (Irene)

For Lab 5, our team solved the problem of robotic localization. We created an accurate localization model by implementing Monte Carlo Localization on our robot so that the racecar can determine its orientation and position in a known environment. The goals of this lab were to produce a localization algorithm that can use LIDAR and odometry data to create an accurate estimation of the robot's position as it drives around a space both in simulation and in the real world.

To create an accurate localization model, our team needed to be able to synthesize odometry data and LIDAR data and transform that into a model that represents the robot's pose while also taking into account the fact that our sensor readings contain a lot of uncertainty. We began this process by calculating a motion model for the odometry data. This model allows us to update the position of the robot in space based on odometry information. Then we calculated a probability function for the LIDAR that defines how likely it is to record a given sensor reading from an associated hypothesis position. We used these calculations to update the estimate of the robot's position. We then combined the two algorithms to program Monte Carlo Localization on our robot, fusing the motion information coming from the odometry with the information coming from the lidar while including the uncertainty of the measurements in the model.

After programming the localization model, we performed a variety of tests to assess the accuracy of our model. We performed unit tests on the different parts of the particle filter to verify that they were successfully implemented. Then we ran our robot in the simulation environment to compare the accuracy of our model against the simulation's ground truth. We finally tested our code on the robot and used visual inspection to compare our published localization model against the path of the robot.

The ability to perform localization is critical to accomplishing the task of creating a fully autonomous vehicle. In order to implement path planning or obstacle avoidance, the robot needs to be able to track its position and orientation in space. This lab has allowed us to make significant progress toward making our robot fully autonomous. Aside from teaching us how to implement localization, this lab also showed us how to do sensor fusion as well as model the uncertainty in our measurements.

2 Technical Approach (Irene)

To create an estimate of our robot's location, we fused two measurements together. The robot can track its movement through the use of odometry, which is an estimation of the robot's change in pose based on sensors that directly measure the robot's velocity and turning angle at a given time. Dead reckoning, or measuring position estimations based solely on odometry, introduces a large amount of error in the measurement because in order to calculate your position you integrate your velocity measurement. Integration has a compounding effect on the error, so the pose estimation gets distorted over time. In order to more accurately measure the pose, we combined our odometry measurements with a secondary sensor, our LIDAR scanner. We can determine the robot's position in its environment indirectly using LIDAR scans. The robot can estimate its position by using the LIDAR to create a geometric model of its surroundings and comparing this model to a known map of the environment. However, this means that if two areas of the map have similar geometry, the robot will have difficulty distinguishing between them. While both of these measurements are not perfectly accurate, when used in conjunction, each can counteract the error inherent in the other method and create a highly robust localization model.

To create the localization model we used Monte Carlo Localization. Monte Carlo Localization models the pose of the robot as a set of points that represent a possible location for the robot to be in along with the probability that the robot is in that location. Every time we receive an update of new odometry data, we apply that data to each particle in the filter to find its new position. We then use the LIDAR scanner to update the model. The robot uses a probability model to predict the likelihood that the lidar scans we made came from the different positions the robot might be in. Finally, the robot prunes unlikely particles and returns the pose of the robot from an average of the particles that represent pose estimates.

Below is a detailed description of the three parts of the particle filter: the motion model, the sensor model, and finally, the particle filter which puts it all together.

2.1 Motion Model (Miles)

The motion model updates the localization estimate by updating the particle filter with incoming odometry data, taking into account possible error in the measurements.

Whenever the robot moves, it generates odometry data, which is then fed into the motion model. Each particle in the particle filter is then moved by the amount specified by the data. Each particle moves not only in the 2D position dimensions, but also in the rotational dimension, and so particles that are oriented in one direction might move to a different position than particles oriented in another.

However, the odometry data is often unreliable, with the measurements being offset by an unknown amount. To account for this, the particles are then injected with a small amount of noise that can plausibly account for random error in the odometry measurements. Each dimension of the robot's location (namely x, y, and theta) is injected with a different amount of noise to account for the different scales of each dimension. For example, theta is bounded by 2π and -2π, while x and y are unbounded. The amount of noise added to each dimension is given by the formula below:

$$y = x + u(-2x, 2x) \tag{1}$$

Where y is the updated particle dimension, x is the measured odometry data and $u(a, b)$ is a uniform random distribution from a to b. We found that this formula properly scaled the injected noise to more closely follow realistic errors in the odometry data, while also being easy to compute to allow it to be run many times per second.

2.2 Sensor Model (Miles)

The particles, after being updated by the motion model, are then given likelihood weights by the sensor model whenever new LIDAR data is received. The sensor model uses a probability density function to determine the likelihood that each particle is correct given the measured

LIDAR data and ground-truth measurements. The ground-truth is determined by raycasting from each particle in a predefined map.

There are four parts to the probability function. The probability of the LIDAR returning a hit at the ground-truth distance is represented as a gaussian distribution centered at the true distance (2). This is adjusted by the probability that the LIDAR returns a shorter distance than ground-truth, which is represented as a downward-sloping line starting at zero distance because the LIDAR is more likely to hit closer objects than farther objects if they are uniformly distributed in the environment (3). Then the probability that the LIDAR returns maximum distance is added, which is represented as a spike of size 1 at the farthest possible measurement point (4). Finally, a uniform probability distribution is added, to represent unavoidable random error in the LIDAR scan (5). Each part is weighted differently and added together to form the final, composite function. The final function and the corresponding weights are described in (6).

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

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

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

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

$$\begin{gathered} p(z_k^{(i)}|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m) + \\ alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m) \end{gathered} \tag{6}$$

Where $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, $\alpha_{rand} = 0.12$, and $\sigma = 8.0$.

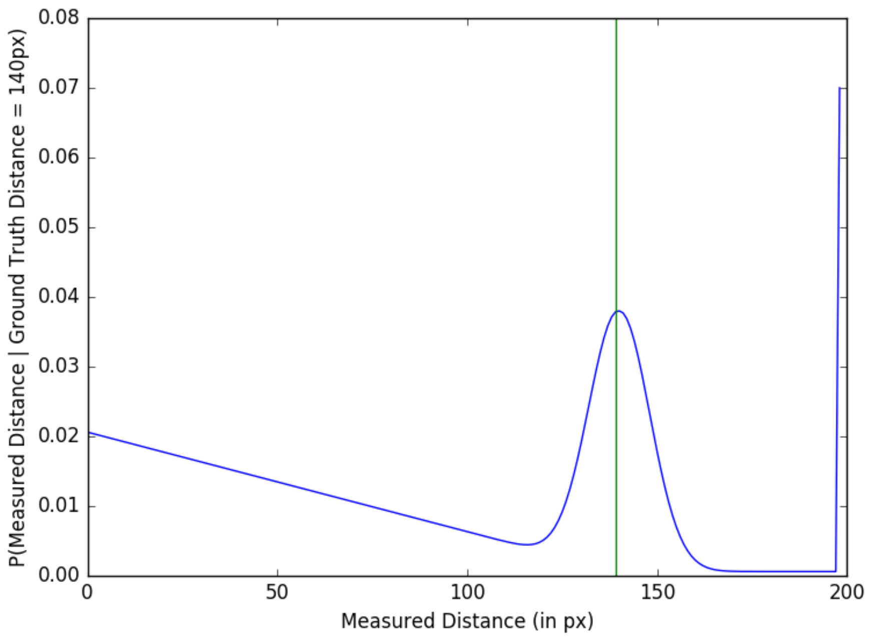

Figure 2.21. The composite probability distribution of the likelihood of measuring a distance from the LIDAR, for a true distance of 140px.

In order to improve the performance of the algorithm to be able to run in real-time, the probability distribution is pre-computed in the final implementation. First, each distance point is discretized so that the function is no longer continuous. Then, for each ground-truth point and each measured distance point, the value of the probability function (6) is computed and added to a probability table (Figure 2.22). This allows the code to simply look up the probability value in the table whenever it receives LIDAR data, speeding up computation significantly.

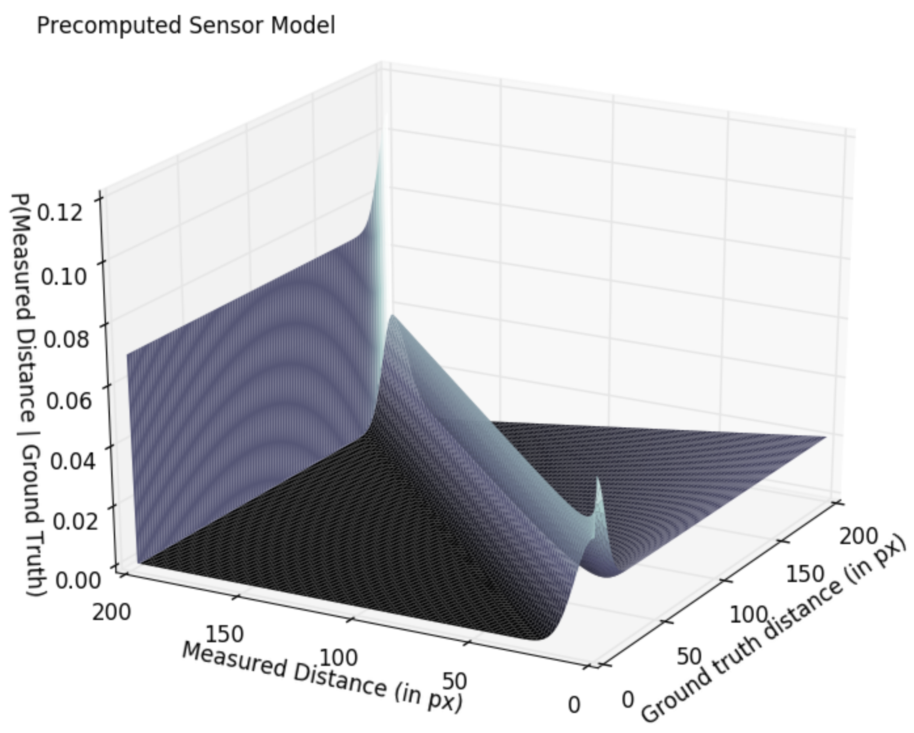

Figure 2.22. The precomputed sensor model, denoting the probability of receiving a measured distance from the LIDAR for a given ground-truth distance.

2.3 Particle Filter (Miles/Matthew)

The particle filter puts the motion model and the sensor model together by passing in the measured odometry and LIDAR data and storing the resulting updates to the particles.

The particle filter is initialized by receiving an estimated pose for the initial state of the car, along with a covariance of possible error. We did not use global localization and dead reckoning, as this proved difficult given our data. The particles are thus spread around the robot's initial ground truth position in a Gaussian distribution. The robot continuously receives an update from the motion model, which means that every particle's current position is updated. In parallel with the motion model updates, the sensor model continuously updates the probability of each particle existing in its current position, as described above. The particle filter can then ‘prune’ particles that have a low probability of existing in that location. This means that the filter will remove particles on the edge that drift far away from the robot’s actual location and remove those particles, and thus the center of the particles will shift closer to the robot’s actual location, and so the center of the particle cloud is always a good estimate. We used a specific circular mean algorithm, as averaging angles through traditional methods does not give the expected results. We were advised not to use averaging to determine our center, but we did not encounter any issues with this during our implementation process and felt no need to revise our function to choose our center.

We made more performance improvements by taking into account threading in our algorithm, as ROS is not thread-safe by default. In order to avoid race conditions when odometry and LIDAR data are received at the same time or before previous updates are finished, we implemented thread locking in our code. We locked the stored particle array each time it is about to be updated and released it once it is updated, so no race conditions could occur. Once these changes were implemented, as well as other performance improvements such as array vectorization our code was able to perform at an acceptable frequency to work in real-time.

3 Experimental Evaluation (Lili)

After implementing the motion model, sensor model, and integrated particle filter, we needed to ensure that they were functional, robust, and reliable. Testing was also instrumental in giving us feedback and ensuring we knew what to iterate and improve on technically. In addition to this, the two models and the integrated particle filter were written separately, so we needed a way to test individual components for debugging.

3.2 Test Procedures

We built up our tests incrementally. The first stage included unit testing specific pieces of our code. Unit testing was important as it enabled us to develop more efficiently, by coding different parts of the lab in parallel. The second stage consisted of testing the integrated particle filter in simulation. Testing our code in simulation is always an important step and this lab was no different. Through testing in simulation, we are able to find bugs early on and conduct repetitive tests in a more controlled environment, prior to introducing the complexity and limitations of the real hardware. The final stage consisted of putting the particle filter onto the car.

3.2.1 Unit Tests

The two main components of the code that needed unit testing were the sensor model and the motion model. These components were good candidates for unit testing because both have a singular and very clearly defined functionality: given an input, there is a correct, expected output.

The motion model took as input a list of particles, each described by an x-position, y-position, and heading angle, as well as odometry, consisting of displacement in x, displacement in y, and displacement in the heading. The physical model is fixed and does not change from implementation to implementation, allowing us to test for correctness by feeding in specific inputs and checking the outputs. The ability to check the correctness worked when the motion model was deterministic, i.e. no artificial noise was added to the particles. The motion model passed the unit tests given.

The sensor model particles in the form described above as well as a LIDAR observation. Similar to how the physical model of the car was fixed, so was the probability density function we were using. Therefore, we could check for correctness as well by feeding in known inputs. The sensor model also passed the unit tests provided.

3.2.2 Simulation Tests

Before testing our system on the robot hardware, we needed to test it in simulation first. To do this we ran the particle filter, the racecar simulator, and the parking controller from the visual servoing lab together. The particle filter published its estimate of where the robot was at any given moment. In the parking controller, we were able to set a point for the car to drive to with a smooth path. With the racecar simulator, we were able to set initial positions, publish ground truth odometry, and have LIDAR scan data. Through these three packages, we were able to control and visualize smooth robot paths and compare them with the visualized estimate which used the odometry and LIDAR scan data.

When running these tests, we loaded the Stata basement map as well as the original map to test our particle filter on different LIDAR scans and environments. We also tried multiple different paths including straight lines, going around corners, and change of directions, in order to test for more diversity in the odometry data.

Through our testing in simulation, we were able to tune the noise parameters and coefficients to achieve better performance with the particle filter, before testing it in real life.

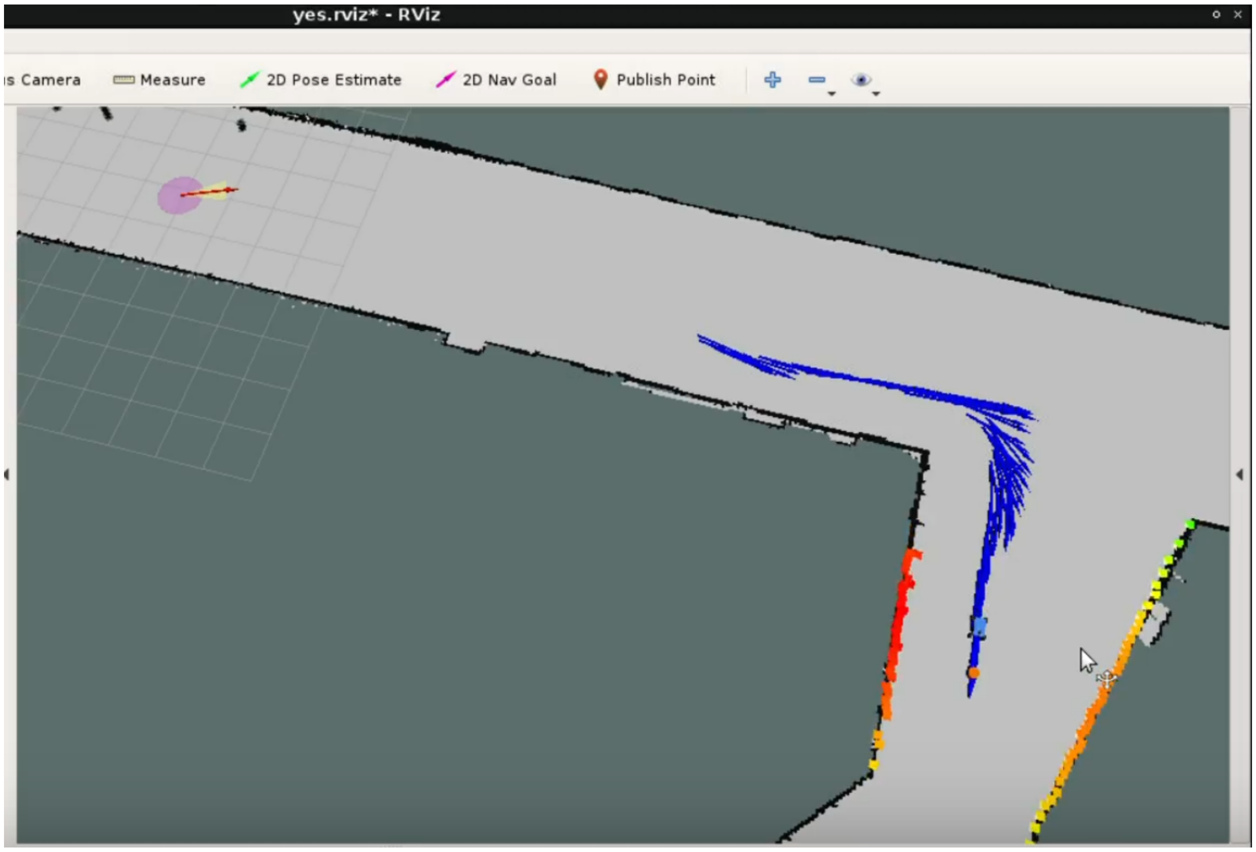

Figure 3.1: The blue arrows are the estimated odometries from the filter. This was its performance turning a corner on the Stata basement map.

3.2.3 Hardware Tests

After we were satisfied with the performance of the particle filter in the simulation we put the filter on the robot. To do this we started at a position where we could approximate on the map shown in RViz. We had a few paths and followed them multiple times, driving in teleoperated mode while recording rosbag files. There were deviations from the Stata basement map and the real basement. For example, one of the long walls had pile after pile of boxes (which wasn't in the map given) so we chose different paths that excluded and included that wall to test. In addition to this, we tried to start on the same spot many times and follow approximately the same path to test repeatability behavior.

After recording the data we played the bag files while running the racecar simulator and the particle filter. In this way, we could test on more noisy data with stochastics from the real hardware. We also used this to test if our Velodyne LIDAR corrections were still functional on the particle filter.

3.3 Test Results

3.3.1 Simulation Tests

We scored our simulation tests with a metric based mainly on deviation from the ground truth path. The formula is shown below.

$$d = \frac{1}{T} \int |x_{est}(t) - x_{true}(t)| dt \tag{3.1}$$

$$max(0, 1 - max(0, d_{submission} - d_{ref})) \tag{3.2}$$

Our particle filter proved to work very well, with final scores in the table below.

Table 3.1: The results from simulation tests.

Test Name	Score out of 1.0
No noise	0.948
Some noise	0.951
More noise	0.946

We were also able to verify our results via a visual inspection. We can clearly see in the figures below corresponding to test cases with no noise, little noise, and more noise, that our particle filter's estimated path did not deviate far from the ground truth.

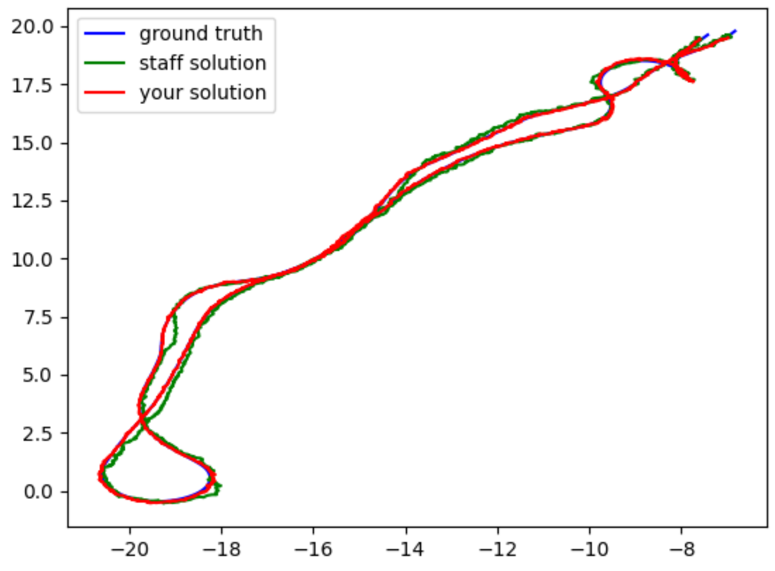

Figure 3.2: Our particle filter's estimated trajectory with no odometry noise.

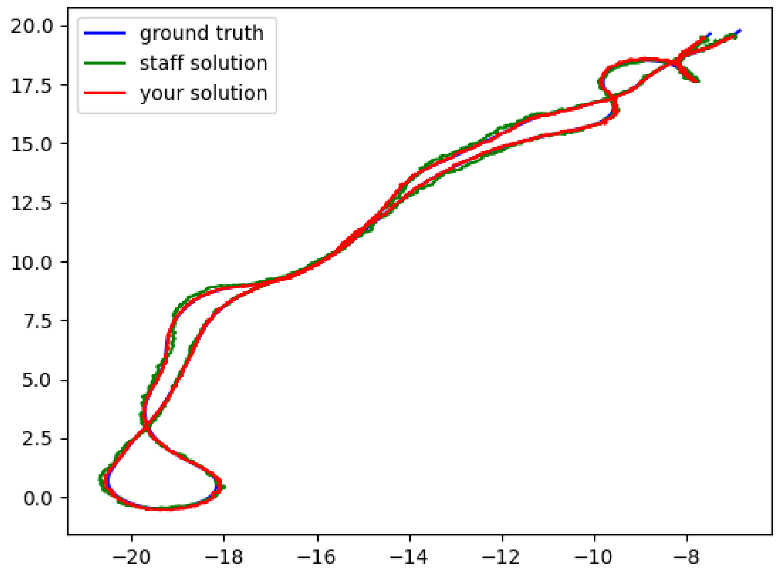

Figure 3.3: Our particle filter's estimated trajectory with some odometry noise.

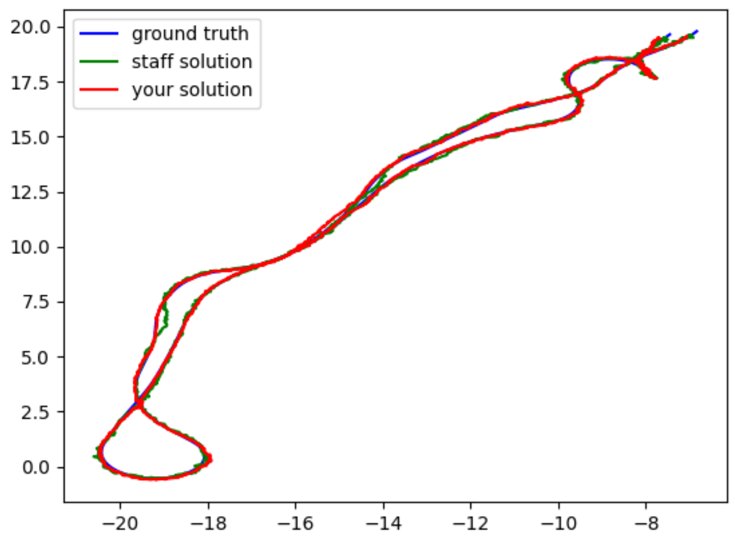

Figure 3.4: Our particle filter's estimated trajectory with more odometry noise.

3.3.2 Hardware Tests

Our hardware tests were qualitative as we had no ground truth to compare our results to. We tested two main paths and driving patterns. One path started in the hallway outside the main classrooms and the second around the bike racks. We also tested patterns such as zig zag driving. In Figure 3.5 we can see that the estimated odometry (in blue) exhibits the zig-zag pattern as well as a U-turn performed.

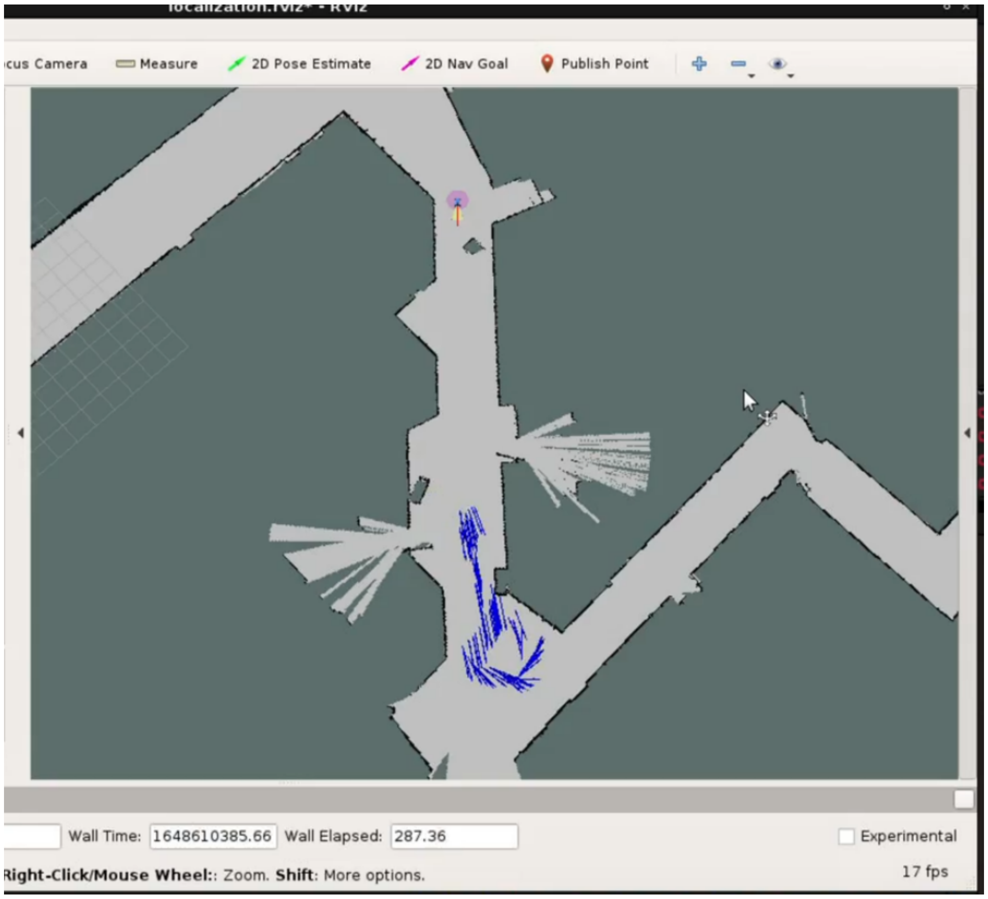

Figure 3.5: The blue arrows show the estimated odometry from the particle filter when running on data collected from driving the robot around.

3.3.3 Test Limitations and Observations

It is important we recognize the limitations of our testing. We did not have ground truth for the hardware tests so it was difficult to gather any tangible metrics. This is something for us to keep in mind for future labs, on how to quantify our performance as best as possible.

Overall, our data and qualitative observations show that our particle filter is effective and reliable.

4 Conclusion(Matthew)

We were able to create a robust, accurate, localization system that could model the robot's position at all times, even if there is a large amount of noise in our odometry data. Our Monte Carlo Filter was comparable to the staff solution, and the robot was not susceptible to being led off course in the simulations that we covered. We conclude that it thus avoids two of the major issues that could make a filter ineffective. Our results matched our predictions when we reduced our Monte Carlo filter into a Dead Reckoning system by removing the spread of particles and keeping track of only one pose by computing the sum of all of our odometry data. We found that the Dead Reckoning system performed well when there was no noise in our data, but failed on noisier tests. This was not surprising, because the integration of perfect odometry data is a problem with a singular deterministic answer that is solved by directly adding up the contribution of each piece of data. When we added a distributed particle cloud, it performed better on the tests. It is also interesting, but not surprising that we ended up having to incorporate noise into the odometry data we received. It is likely that without this noise, the particle cloud would move in unison, and as unlikely particles were eliminated, the cloud would fail to spread and would instead approach a singular point. Over time, this would essentially turn into a Dead-Reckoning system.
In the future, we could consider improving our Monte-Carlo localization filter by adding more particles that provide a better overall estimate. We could possibly accomplish this by improving our threading procedures and allowing various computations to be performed in parallel. In addition to general-purpose optimization, we could tune several parameters of our robot in order to decrease error in our position estimation. We could align the LIDAR against a wall of some specific distance and repeatedly sample measurements, and plot the actual measured probability distribution we see. We could use this to modify the probability values for each of the four types of LIDAR error we discussed. We could also modify the function we use to output the robot's final position, given the cloud of particles. Despite the warning that if we used a simple averaging, we might have difficulty if the robot could be in one of two equally likely locations, it does not appear that our robot had any such issue. The estimate was never split between two specific clouds of probability, and in fact, our robot did not deviate completely from the correct path as often as the staff solution did. Nonetheless, it is possible that there is a more sophisticated algorithm for reducing the cloud to a single position that would eliminate sources of smaller errors.

Lessons Learned:

Lili- On the technical side I definitely underestimated the importance of the covariance of the initial particle distribution. Both this and the noise injected in the motion model ended up being critical to our performance! On the CI side, I learned how to more systematically solve issues when it involved someone else's work, in this case, code.

Miles: I learned the importance of aggressive optimization when it comes to code that runs many times per second. Specifically, I learned the problems multithreading may cause, and how to use python's threading library to fix them. I also learned the value of precomputing calculations that you would reuse often. On the CI side, I learned how to more properly structure technical reports, as well as how to present hybrid briefings more effectively.

Irene: During the lab, I learned how to not only address measurement errors from sensor reading but use the errors themselves to create a more robust and accurate model. On the CI side, I learned a lot about how to give better technical presentations and write lab reports. The talk with the CI instructor was very helpful.

Matthew:

I think the coolest thing that I learned from this lab was how the odometry and LIDAR could be combined to work to the other's strength and create a very accurate localization model. Independently, the odometry was very unreliable, and the LIDAR was also frequently inaccurate when we attempted to use it on the robot. This makes me want to consider how we could combine multiple sensors to reduce other kinds of error in the future. On the CI side, I learned how to present with incomplete results and to make sure that the audience understands that your preliminary results are promising.