Lab #5 Report: Localization

Team #12

Artem Laptiev
Cruz Soto
Evan Bell
Yasin Hamed

Robotics Science and Systems

April 15, 2023

1 Introduction
by Evan Bell

The goal of this localization lab was to use motion and sensor data to model the uncertainty of the robot's position within an environment. Determining a robot's orientation and position in a known environment, called localization, is a critical problem in the field of robotics. Given a known map, the goal is to estimate the robot's position as accurately as possible while moving through the environment. This is important for environment navigation because the intended action of the robot is not always accurate to the physical behavior, and error between the two can accumulate through time unless corrected.

This position correction is mainly done through Monte Carlo Localization (aka MCL or particle filter), which is a method in which sensor data and motion odometry data are both used to continuously generate probable points of the robot's location to refine the estimated position and pose of the robot. The motion odometry includes the measured translational velocities in each direction and the measured rotational velocities around the three axes in space. The sensor data is a 2D lidar scan taken from the 3D Velodyne sensor on the robot.

Both motion and sensor rospy Subscribers must update the robot's position, which may lead to both subscribers trying to update the position at the same time in some cases. The complexity of these models individually makes combining them more difficult, specifically in getting the data types and data representations to align between the many functions performed on the data.

This lab lays a strong foundation for path planning and path following in the future, because the success of path planning and path following relies on having an accurate estimation of the robot's position regardless of small error in measured data.

2 Technical Approach

2.1 Motion Model
by Evan Bell

The motion model is meant to update the particle positions according to motion odometry data. This is critical because the robot's best estimation as to its new position is to combine the old position some time dt ago and the relative change in position since then.

The actual motion odometry data is expressed through *Odometry.msg* types, mainly in the $geometry\_msgs/Twist$ message which holds two $geometry\_msgs/Vector3$ float64 vectors, "linear" and "angular". The linear vector holds robot velocities (m/s) in the x,y,z directions. This is relative to the robot, so the positive x di-

rection is the forward direction of the robot, the y direction points to the side of the robot. The robot is only expected to have a x velocity because the robot can only drive forward/backward and will never be moving laterally or vertically. The angular vector holds robot angular velocities (rad/s) around the x,y,z axes. Again this is relative to the robot. The robot is only expected to turn along the z axis, as turning along either of the other axes would represent the robot turning up or on its side. Thus, the odometry is described by a forwards linear velocity (m/s) and a z angular velocity (rad/s), relative to the robot's frame.

Between each new odometry message, a time difference dt is kept. To obtain the total change in position of the robot p_{t+dt} in this time frame, we multiply the velocity information of the message by the time quantity to obtain total x_{odom} distance traveled and total θ_{odom} rotation around the z axis made since the old position.

$$x_{odom}\ (\text{m}) = \text{x\_velocity} * dt \qquad (\frac{m}{s} * s)$$

$$\theta_{odom}\ (\text{rad}) = \text{z\_angular\_velocity} * dt \qquad (\frac{rad}{s} * s)$$

This is now translation in the frame of the robot. Given a particle p_t in the world frame with position (x_t, y_t) and pose θ_t, we create a rotation matrix from θ_t, which we multiply by the odometry distance values to transform them to the world frame the particle was in. Then we add p_t to get the new particle values.

$$R = \begin{bmatrix} \cos(\theta_t) & -\sin(\theta_t) & 0 \\ \sin(\theta_t) & \cos(\theta_t) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$p_{t+dt} = R \begin{pmatrix} x_{odom} \\ 0 \\ \theta_{odom} \end{pmatrix} + p_t$$

$$p_{t+dt} = R \begin{pmatrix} x_{odom} \\ 0 \\ \theta_{odom} \end{pmatrix} + \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix}$$

This is done for every particle to update each of their positions.

A further part of the motion model is to implement noise in the odometry data to account for error in the robot's position which may come from error in odometry. In the simulation, the robot's actual position can be exactly known because it is deterministic. This is because the reported odometry data is always correct. However, in the physical implementation this is not the case. The reported odometry may be incorrect, and often we expect it to have some error. This requires that we express noise in the reported odometry.

This is done by offsetting the odometry used for each particle. The goal is to offset the odometry in a way that might correct errors in the given odometry, resulting in a particle position that is more accurate to the robot position than it would be using unchanged odometry. If we vary many particles, ideally a few will be more correct in this way, and the sensor model can use them instead of the particle derived from unchanged odometry.

The goal is to generate two values $(x_\Delta, \theta_\Delta)$ for each particle which are added to x_{odom} and θ_{odom} distance values to produce noise in the odom values used. We also choose to generate a value y_Δ to account for uncertainty in the lateral position of the robot, although the $(x_\Delta, \theta_\Delta)$ values of the previous time-step may account for this. This new odometry used in the calculation is derived as

$$= \begin{pmatrix} x_{odom} \\ 0 \\ \theta_{odom} \end{pmatrix} + \begin{pmatrix} x_\Delta \\ y_\Delta \\ \theta_\Delta \end{pmatrix}$$

We explored two main distributions for choosing these values. The first distribution is a uniform distribution. In this, x_Δ and y_Δ are randomly chosen from the range [-0.5, 0.5] with equal probability (this value is in meters). θ_Δ is randomly chosen from a range $[-0.44, 0.44]$ (unit is radians, converts to roughly -25/25 degrees). A uniform distribution is pictured below.

Figure 1: Uniform Distribution

The properties of this are that the values are symmetric around 0, meaning we are just as likely to get a value above 0 as below it. We expect that error may be just as likely to occur in either direction (actual x velocity is equally likely to be faster than measured as it is to be slower than measured, for exam-

ple). This distribution has ends to the range of values produced, meaning that error greater than the range values will not be represented.
The second choose is to choose from a Gaussian distribution. We choose x_Δ and y_Δ randomly from a Gaussian distribution centered at 0 with standard deviation 0.1, and θ_Δ randomly from a Gaussian distribution centered at 0 with standard deviation 0.0873 (unit is radians, converts to roughly 5 degrees). An example Gaussian distribution is shown below.

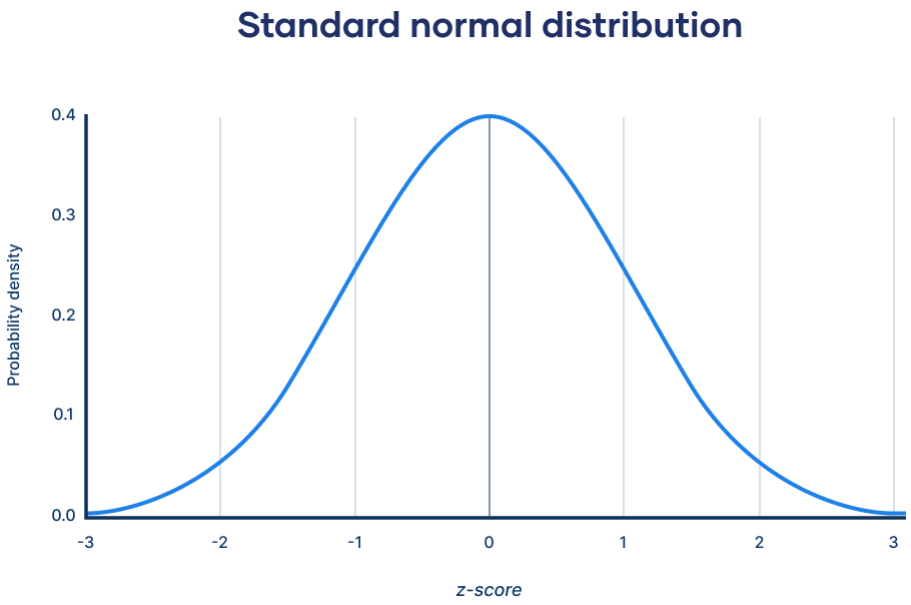


Figure 2: Gaussian Distribution

The key property here that there is a much higher chance of values closer to the mean (0 in this case) being chosen than values farther being chosen. This indicates that we expect small error in odometry to be more likely than large errors. Given that we expect massive error in estimated position to only accumulate after a while with no correction and we expect to be constantly performing this MCL correction, this assumption seems valid and makes this distribution appear more useful than a uniform one. Also, in this Gaussian distribution there is an equal probability that a value $-z$ is chosen than z being chosen. This is also consistent with our expectation that error may be just as likely to occur in either direction as mentioned for the uniform distribution. This distribution is most commonly used for sampling random occurrences, and does very well at

modeling noise from error in other applications, which we find to be the case here as well.

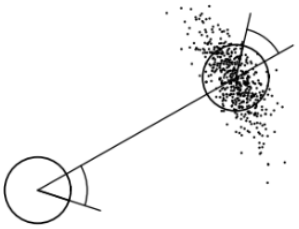


Figure 3: Odometry noise (*Probabilistic Robotics textbook*)

The above is a visualization of how noise in odometry can generate noisy potential points. If the odometry information provided above had error, one of the 'noisy' positions obtained may be more accurate to the robot's actual position, which the sensor model can help determine.

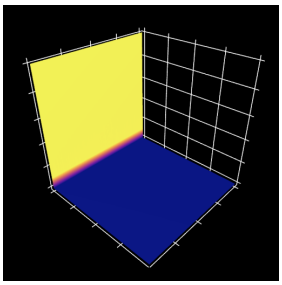
2.2 Sensor Model by Artem Laptiev

After receiving a set of potential location points for the robot generated by the motion model, the next step is to use sensory perception and knowledge of the environment to determine the most likely position of the robot. The sensory model serves as a check on the motion model's predictions, ensuring that the robot's true location is not lost over time due to prediction errors. While different sensory systems could be used for this task, LIDAR scanning of the environment is the most practical solution for our purposes.

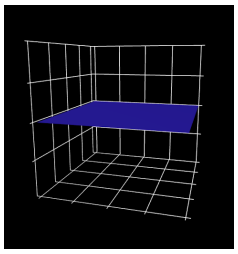
Given a LIDAR scan distance to an obstacle, denoted as z, and the ground truth distance from a position in question, denoted as d, we can estimate the probability that the position in question is the robot's true location. This is accomplished by designing a probability function $P(z, d)$ that accounts for a range of possible scenarios for LIDAR scan readings. Specifically, we model these scenarios with four functions: $hit(z, d)$, $short(z, d)$, $max(z, d)$, and $rand(z, d)$ as the probability of detecting a known obstacle in the map, the probability of a short measurement, the probability of a very large (aka missed) measurement, and the probability of a completely random measurement respectively (Figure 4).

The probability of detecting a known obstacle in the map is high, as it is assumed that the map is a true representation of the environment. However, there

(a) $hit(z, d)$ (b) $short(z, d)$



(c) $max(z, d)$



(d) $rand(z, d)$

Figure 4: Probability functions that constitute the sensory model. Measured distance on the x axis and the ground truth distance on the y axis

is also a probability of a short measurement due to internal LIDAR reflections, hitting parts of the vehicle itself, or other unknown obstacles such as people or animals. On the other hand, a very large measurement is also possible, usually caused by LIDAR beams that hit an object with strange reflective properties and did not bounce back to the sensor. Finally, there is a probability of a completely random measurement, which is very unlikely but still accounted for in the probability function.

Designing the sensory model involves selecting the appropriate coefficients for each of the four probability functions: α_{hit}, α_{short}, α_{max}, and α_{rand}. The default values for these coefficients, provided by the lab instructions, are $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, and $\alpha_{rand} = 0.12$. We planned to use $dynamic\_reconfigure$ module to tune the coefficients for better performance

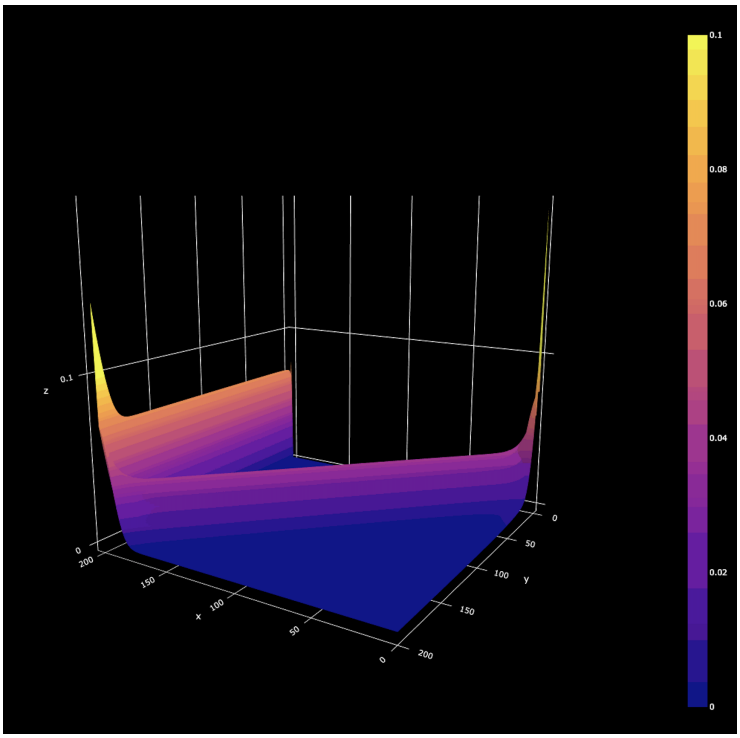


Figure 5: The final sensory model representation. Measured distance on the x axis and the ground truth distance on the y axis

however that turned out to be unnecessary.

The four probability functions and corresponding coefficients are combined to form the sensory model, which calculates the probability that a given location is the robot's true location (Figure 5). Specifically, the sensory model is defined as:

$$P(z,d) = \alpha_{hit} * hit(z,d) + \alpha_{short} * short(z,d) + \\ \alpha_{max} * max(z,d) + \alpha_{rand} * rand(z,d)$$

It is important to note that running the sensory model function on real-time sensory data is computationally expensive. Therefore, we precompute a discrete sensory model table with values up to the maximum LIDAR scan distance and look up values from the table in real time instead.

2.3 Particle Filter
by Yasin Hamed

At the highest level, the particle filter serves to combine both the motion model and the sensor model in order to accurately update the robot's position recorded position in the known map. This filter subscribes to topics which stream information about the robot's odometry readings as well as LIDAR sensor readings. Whenever new odometry is received by the particle filter, it updates the possible poses the car could be in as detailed in section 2.1 regarding the motion model. When the particle filter receives a LIDAR scan reading, it resamples from the possible poses of the car it is considering as possibilities as detailed in the section 2.2 regarding the sensor model. The updates from the motion model can be thought of as simply representing a movement of the car through space while the updates from the sensor model can be thought of as corrections for drift in the trajectory that results from uncertainties and noise present in the odometry and motion model. This can be seen in figure 6, where the distribution of possible poses can be seen as starting relatively tightly clustered. However, as the motion model is applied, these begin to disperse further and further from the true position of the car. Finally, the points are corrected by the application of the sensor model which takes in external information about the environment to adjust the pose estimate of the robot.

The cluster of points shown in figure 6 surrounding the true pose of the robot is always being considered and processed by the program. However, the final estimate of the robot's pose must be a single pose which is effectively an average of all of the points in the cluster. This is a trivial problem when averaging the x and y components of the poses – simply taking an arithmetic mean across all the x and y components of all the pose possibilities results in an x and y value for the estimated pose, respectively. The rotation, θ, of the final pose estimate is somewhat more complicated to average across the poses since θ takes on a range of values which is circular. That is to say $0°$, and $1080°$ are both the same angle, but taking the arithmetic average results in $540°$ which is not the same angle (it is geometrically the same to $180°$). Instead of averaging the values of theta directly, we decomposed the angles into horizontal $\theta_{x,i}$ and vertical $\theta_{y,i}$ components, averaged all of the horizontal components and all of the vertical components separately, resulting in the the averaged values θ_x and θ_y. Finally, we combine these two results into a final angle measurement which serves as the θ component of the pose estimate for the car. This averaging process is summarized by EQUATIONS XXXXXXXX.

For possible poses $p_1, p_2, p_3, ..., p_n$:

$$x_{avg} = \frac{\sum_{i=1}^{n} p_i.x}{n} \tag{1}$$

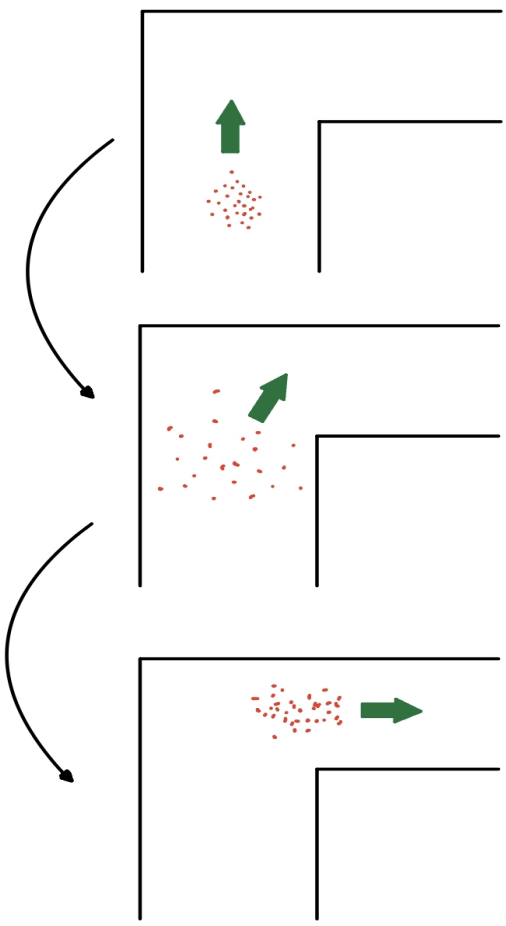


Figure 6: The above maps visuallize the progression of the pose estimates through time as they are at first updated by the motion model, spreading out, and then by the sensor model, recentering on the true position of the robot.

$$y_{avg} = \frac{\sum_{i=1}^{n} p_i.y}{n} \tag{2}$$

$$\theta_{avg} = \arctan \frac{\sum_{i=1}^{n} \sin p_i.\theta}{\sum_{i=1}^{n} \cos p_i.\theta} \tag{3}$$

By averaging along the Cartesian components of the angles, an effective average is produced from θ which follows logically from the geometry.

The motion and sensor models only run when receiving information from the topics they subscribed to (for odometry and LIDAR scans), meaning they run independently of each other. In the case where odometry and LIDAR data is received by the particle filter at time very close to one another, it is possible that one process interrupts the other in the middle of updating the poses. This would lead to unwanted behavior in the particle filter and inaccuracies in pose estimations. To avoid this, we implemented threading into the particle filter with a thread from the motion model execution and a separate thread for the sensor model execution. These threads allow for the models to execute to completion once called and avoid interruptions between the models.

3 Experimental Evaluation

3.1 Visualization by Cruz Soto

With a working sensor model and particle filter, the team moved to integrating the model in real time by visualizing the predicted location of the trajectory using a spread of points. This was first done in simulation and visualized in Rviz by publishing a marker object that wouldn't fade over time. The sensor model itself works by referencing a table of discrete probabilities to cut down on processing time and using these probabilities to update the eventual drift of the odometry, and the best way to test.
The car itself had some performance choke points in localization, losing coordination during sharp turns or going into corners. Split hallways were also relatively hard for the vehicle, as the timeout of the lasers meant that looking straight down a long hallway in multiple directions (such as the bottom right corner of the map where three long halls meet) gives very little sensor data consistent with the map (as probability mapping is not possible at the maximum range of the sensor).

(a) Predictions at x_k (b) Predictions at x_{k+1}

Figure 7: The car's predicted location shoots up in uncertainty when executing turns or in featureless hallways

3.2 Errors and Analysis by Cruz Soto

However, the greatest difficulties of the project came in the analysis. With so much time spent working on the development of a particle filter, the best analysis of the performance came from the autograder, which returned a 95% fit to the class solution under an immense amount of generated noise. However, coming up with performance parameters to evaluate the solution became immensely difficult due to the amount of processing required to collect ROSbags and the visualization under a short time. However, after visualizing both the simulated and real path in RViz, the team developed an experimental, albeit archaic procedure to test the model.

First, tests were run in two different hallways in real life and in three in the simulation. The average particle position was taken down, recording the markers every other time the sensor model updated (essentially as frequently as possible without spending immense time conducting analysis).

Using these points and the wall follower algorithm in simulation (and driving in real life), a disturbance in localization was introduced by resetting the location (re-initializing the marker) and by turning out of a straight path and back into one. Both of these would cause the localization distribution the spread out massively, and it is by assessing the performance of the car after these events that we objectively quantified the convergence rate of the car. The error propagation in these different spaces was then averaged to give the overall performance of the car in simulation and in real life. as expected, the convergence rate and steady state error in real life were longer and larger than those in simulation.

Looking at the best fit data, "convergence time" was defined as the point at which all the data was within 10% of the steady state error or within 0.2 meters of the car (close enough to avoid collisions). These times for the simulation were found to be 1.9 seconds, while the real-time implementation saw convergence

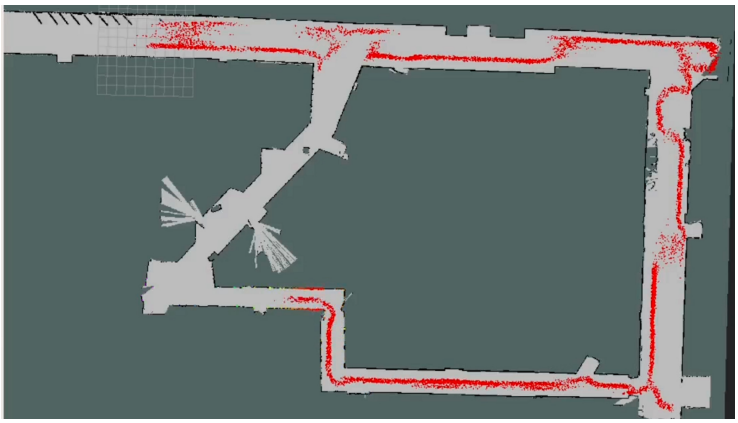


Figure 8: The red trail indicates the previous belief distribution as the car has moved through the map.

times of 2.5 seconds or longer.

4 Conclusion
by Artem Laptiev

In conclusion, the localization lab has successfully demonstrated the importance of accurately determining a robot's location within an environment. The use of Monte Carlo Localization, motion odometry data, and sensor data has allowed for the continuous refinement of the robot's estimated position, even in the presence of small errors in the measured data.

The motion model has been implemented to update particle positions according to motion odometry data, taking into account the time difference between odometry messages and the robot's linear velocity. Additionally, noise has been added to the odometry data to account for errors that may be present in the robot's position estimation.

The sensor model has been implemented to generate particle weights based on the comparison of 2D lidar scans taken from the 3D Velodyne sensor on the robot to the known map of the environment. This allows for the more accurate selection of particles that are closer to the true position of the robot.

Overall, this lab provides a strong foundation for path planning and path following in future robotics applications. By having an accurate estimation of the robot's position, errors can be minimized and successful navigation can be achieved.

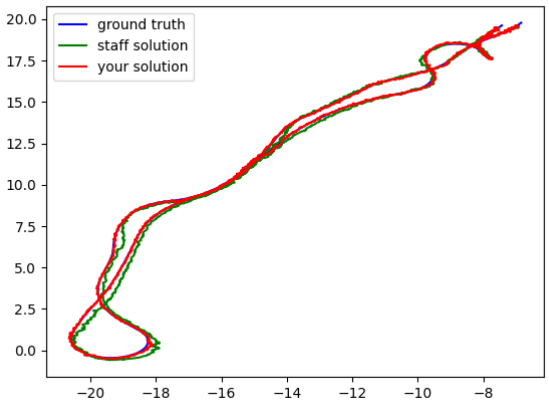


Figure 9: The results of the localization, with an average error of 0.24 meters from the true path and a 95% fit.

5 Lessons Learned

5.1 Artem Laptiev Lessons Learned

During this lab, we went through a challenging process of debugging our code for an extended period of time. It was a valuable lesson that taught me the importance of keeping code clean and organized. To avoid such struggles in the future, we began implementing clean code protocols, such as using predetermined rospy structures instead of building custom data representations, as well as extensively commenting our code. I found building the framework of motion model vs sensor model tag-and-pull system to be an interesting and insightful experience. It provided me with valuable knowledge on how to use internal-external systems for better control, which I believe can be applied to various other tasks as well. In addition, we learned how to visualize graphs with Pyplot, which proved to be a useful skill that I intend to utilize for future labs.

5.2 Cruz Lessons Learned

This lab in particular required a lot of time spent working on and troubleshooting code that was difficult to visualize and consumed a significant amount of time in simulation. Generating our own noise as well as properly referencing the motion model took a lot of time and called for a lot of interteam collaboration. Working on the motion and sensor model over spring break took a surprising amount of time, but was manageable due to the tests that could be run to check them. However, helping Yasin troubleshoot the particle filter took up most of the week upon return, as there was nothing other than the autograder to check it. Instead, most time was dedicated to visualizing particle spread by generating

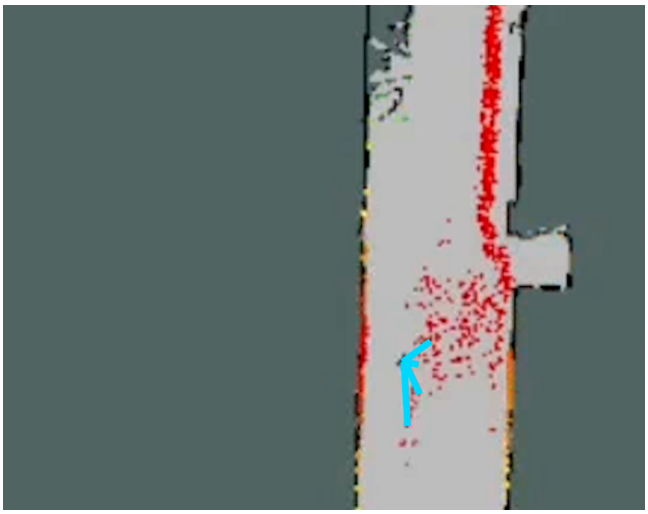
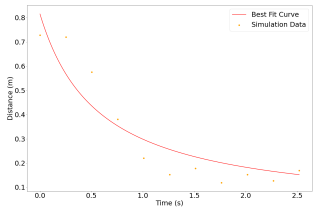


Figure 10: The blue lines indicate measurements to arbitrary points that were most recently published. The four images are from the most thorough run of the basement done in simulation using a combination of driving commands and wall following.

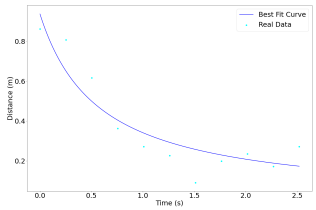
markers, which really helped me in getting to know the different kinds of RViz markers and how to utilize them in future labs. Communication with different members who were stretched very thin with lots of assignments, but were able to budget multiple days for the project over the weekend. In future, we will need to work on communicating times during the week that would work for project focus outside of lab hours.

5.3 Evan Bell Lessons Learned

The topic of localization was really interesting. I had mostly considered decision making based concepts (path-planning, sensing, object identification) when thinking about robotic controls, but localization is a critical part of robotic controls that I hadn't given much thought to. This lab was certainly difficult. Working with large quantities of data meant we used as many numpy based calculations as possible for time efficiency, and there was a learning curve for me especially on working with linear algebra concepts instead of the for-loop type programming I was used to. It really helped me to understand how important planning is for these kind of programming tasks because we needed to have trust that the base operations we were performing on these massive matrices were correct, and there isn't the ability to debug as easily with a task of this complexity. We spent long hours debugging just to find very small bugs that were causing the entire program to fail (I almost didn't even sleep on my birthday) which was frustrating, but it only highlights the need for more rigorous



(a) Convergence of a simulated run, which goes to 0.21 meters steady-state error



(b) Data and convergence of a real run, estimated at 0.28 meters of steady state error.

planning of code. We split up the modules as well, so part of those issues came from having to combine each of our different coded portions which didn't all use the same data types or have the same data properties. The hardware issues were mostly figured out for this lab though, which made the aspects of physical testing much easier than before. I am very happy with the end product, and I am excited to implement it in the final challenge later on.

5.4 Yasin Lessons Learned

I found this particular lab to be a very long and challenging lab compared to previous labs. This was due to the sheer number of different modules and parts that needed to be integrated together to make the monte carlo localization run effectively. The relative increased complexity of this lab led to several problems when debugging our code, and has made me realize that I need to budget even more time in future labs to accommodate for stark increases in complexity. Additionally, in future labs, I will have to work on my assigned portions of the lab with the mentality that the lab staff will not be able to help me since, at least during this lab, staff and class resources were quite limited given that office hours were always very full. Overall, this was a humbling experience in project planning and resource management which I will use to improve my future experiences in the coming labs.