Lab #6 Report: Robot Path Finding

Team # 6

Sera Hamilton
Carlos Sanchez
Yuewei Liu
Miguel Padilla
Willow Pickernell

RSS

April 24, 2025

1 Introduction

An essential aspect of autonomous vehicles is localization and navigation. Localization consists of using sensors to determine the position and orientation of the robot within a known map. Localization is crucial within a robotic system as once the robot knows where it is within a map, it can then use this map for navigation purposes. To solve the problem of determining the robot's position we implement Monte Carlo Localization. MCL works by creating an initial set of particles to estimate the robot's location within the map. These particles are then updated using Bayesian updating techniques to get an improved estimate of the robot position.

We broke this down into three parts:

1. A motion model to update the positions of the particles based on the robot's velocity and change in time
2. A sensor model to calculate the likelihood of each particle using input from the LiDAR
3. A particle filter which relies on the two models to update the set of particles and taking the average of them as the robot's location

Navigation consists of using our estimated position and the known map to plan a path to a goal and subsequently following it. We use A* for path planning, which involves taking an occupancy grid representing the occupied and free space in the environment, and then applying the algorithm to determine a trajectory.

We use Pure Pursuit for path following, which then follows the trajectory.

With this, we have the tools necessary to localize the robot within the map, plan a path, and then move the robot from point A to point B. We first simulated the robot using ROS2 in RViz and then implemented the algorithm on the actual car, which is detailed below.

2 Technical Approach

Our technical approach consists of the development of a motion model, sensor model, and the integration of the modules into a full particle filter for Monte Carlo Localization allowing us to know where the robot is within a map. Expanding on this, we then develop a path planner using A* and path follower using Pure Pursuit to then move the robot to a goal location within the map.

2.1 Sensor Model

To update the particles based on the lidar data, we needed to develop a sensor model. We first computed a sensor model lookup table that allowed us to lookup the likelihood $p(z_k^i|x_k, m)$, which yeilds the probability of sensor reading z_k^i given the hypothesis position x_k and map m at time k.

The likelihood can be broken down into the sum of four different probabilities:

1. p_{hit} : The probability of detecting a known obstacle on the map

$$p_{hit}(z_k^{(i)}|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(z_k^{(i)}-d)^2/(2\sigma^2))} & 0 \leq z_k \leq z_{max} \\ 0 & otherwise \end{cases}$$

2. p_{short} : The probability of a short measurement due to hardware issues or unexpected obstacles

$$p_{short}(z_k^{(i)}|x_k, m) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} & 0 \leq z_k^{(i)} \leq d \,\&d \neq 0 \\ 0 & otherwise \end{cases}$$

3. p_{max} : The probability of a missed measurement due to reflective materials

$$p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{\epsilon} & -\epsilon \leq z_k^{(i)} \leq z_{max} \\ 0 & otherwise \end{cases}$$

4. p_{rand} : The probability of an arbitrary, erroneous measurement

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

The likelihood is then the weighted average of the four distributions, where $\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1$:

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

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

From these equations, we created a precomputed table using parameters $\alpha_{hit} = 0.74, \alpha_{short} = 0.07, \alpha_{max} = 0.07, \alpha_{rand} = 0.12, \sigma = 0.8m, \eta = 1, \epsilon = 1$. The table, pictured below in Figure 1, had a size of 201x201, representing a range of 0 to 200 pixels (and as such $z_{max} = 200$. Further, a row index i represents the distance of the observation, z_k^i and a column index j represents the actual distances d.

Additionally, the table must be normalized so that each column represents a probability. First, since we are working with discrete values, we normalize p_{hit} by dividing each by the sum of all the p_{hit}'s in the column. Then the whole table is normalized by dividing each column by the sum of the entries in the column. From this, we have completed precomputed table as pictured below in Figure 1.

From here, we can develop an evaluate function to generate the likelihood of each particle given the current observation z_k.
To do this, we first simulate what the expected ranges, d, would be for each particle. We divide both z_k and d by the map resolution and lidar scale to map scale to convert the values to pixels for ease of use with the lookup table, and clip the values between 0 and z_{max}. Then, the likelihood of a scan for each particle is the product of the likelihoods of each measurement in the scan, which we can find in the lookup table.

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

2.2 Particle Filter

Using the sensor and motion model, we can construct a particle filter to localize our robot within a map. First we initialize a set of 200 particles around our guess of where the robot is in the map, as pictured in Figure 2. Let the initial guess be x_0, y_0, θ_0. We then initialize our particle positions by drawing from a Normal ($N(\mu, \sigma)$) distribution , $X \sim N(x_0, 0.3), Y \sim N(y_0, 0.3)$. We tried a few different values for standard deviation and landed on $\sigma = 0.3$m as values like $0.5, 1.0$ were too high and $0.1, 0.2$ too low. Our set of theta values were chosen from a uniform distribution, $\theta \sim Uni(-15 + \theta_0, 15 + \theta_0)$. We initially tried a normal distribution, but because we wanted our theta values to be more evenly distributed around our guess without being too large we decided on a uniform

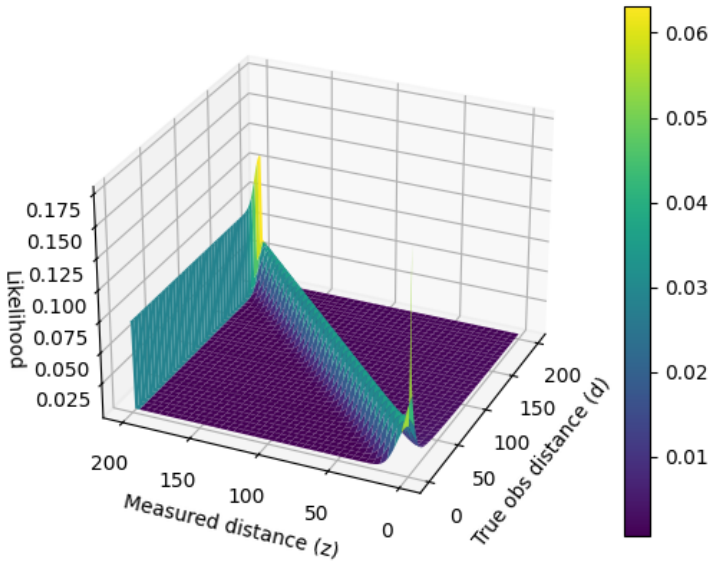

Figure 1: Plot of the precomputed sensor model table. We can see that along the diagonal as the measured distances get closer to the true distances, the probabilities of the scan increase.

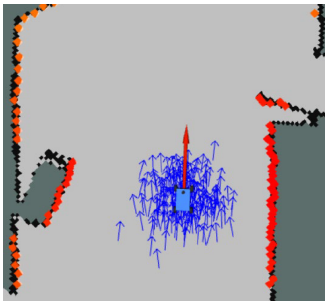

Figure 2: Particles (blue) initialized around an initial guess of the car location.

one.

From here, we add in the motion and sensor model. Each time we get new odometry data, the motion model as described in 2.1 is used to update the particle positions. Each time we get new sensor data, the sensor model is used to evaluate the likelihood of each particle given the sensor data. From here, we

resample the particles based on their likelihoods. Additionally, every time the particles are updated we compute and publish the new average position of the particles as our estimate of the car's location, along with a transformation from "map" to "base_link" to relate the world frame to the body frame. A video of our implementation can be viewed here. In the video, we can see that as the car gets closer to more irregular geometries it is better able to orient itself in the map.

2.3 A* Path Planner

Given a map represented as an occupancy grid, with 100 being occupied, 0 free, and -1 unknown, we apply A* to find a path from the robot's current position to the goal. The goal of A* is to minimize the cost of the path from the start to the goal.

We first dilate the occupancy grid to ensure the robot doesn't get too close to the wall. Dilation is achieved with the dilate function of the cv2 library, which convolves a filter of ones with the grid. We found that a filter of 17 by 17, resulting dilation pictured to the right in Figure 3 below, works best as it keeps the car away from the wall but doesn't create a discontinuity around the smallest corridor.

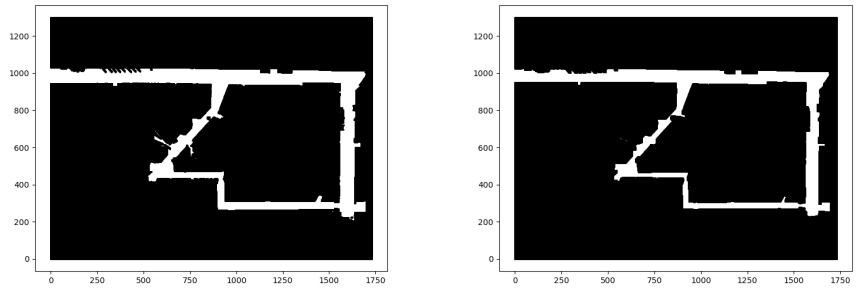

Figure 3: Original map (left) next to map dilated with a 17x17 filter (right)

Then, we create a graph representation of the map. The graph has nodes at each unoccupied point in the map and edges between neighboring nodes. The full graph is pictured in Figure 4 below.

We additionally set a cell size to further reduce the map to a smaller representation, increasing planning speed. For instance, a size of 3 would mean we add every third node. This is pictured in Figure 5 below. Because we no longer use every point, the nodes and edges are much more distinct than in Figure 4:
We can then run A* to find a path between the robot's current position and the

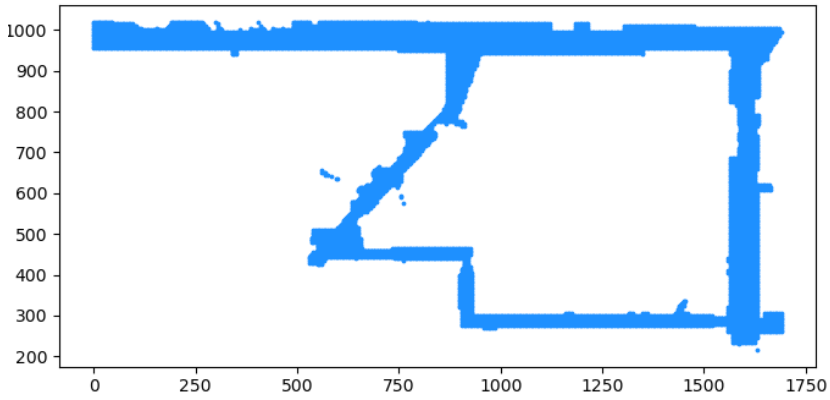

Figure 4: Graph representation of the map.

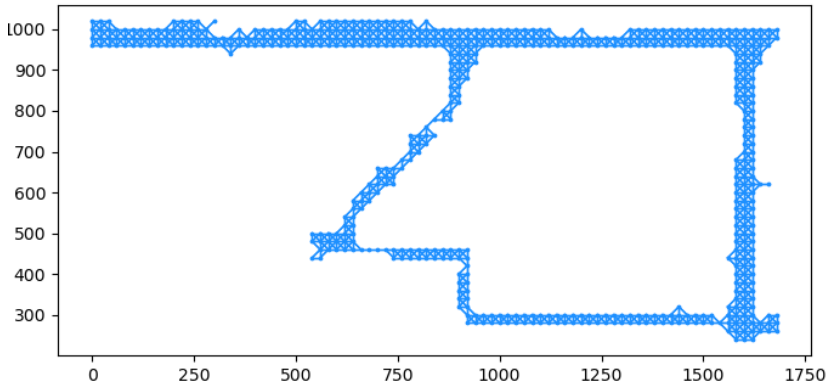

Figure 5: Graph representation of the map with cell size = 20.

goal. We want to minimize the following equation at each node, n:

$$f(n) = g(n) + h(n)$$

$g(n)$ is the accumulated cost from the start pose to node n, defined as follows, where c is the edge weight between n_i and n_{i+1}:

$$g(n_k) = \sum_{i=0}^{k-1} c(n_i, n_{i+1})$$

$h(n)$ is the heuristic estimate of the cost from n to the goal pose. For our implementation, $h(n)$ is the Euclidean distance from n to the goal pose defined below, where (x_n, y_n) and (x_g, y_g) represent the node and goal positions respectively:

$$h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2}$$

Starting at the robot's initial position, with each iteration of A* we expand along the node with the lowest f value, additionally updating costs to neighboring nodes if a shorter path is found. Further, we record the parents of each node along the way. Once the goal is reached the path is constructed using the parent of each node starting from the goal, as pictured in Figure 6:

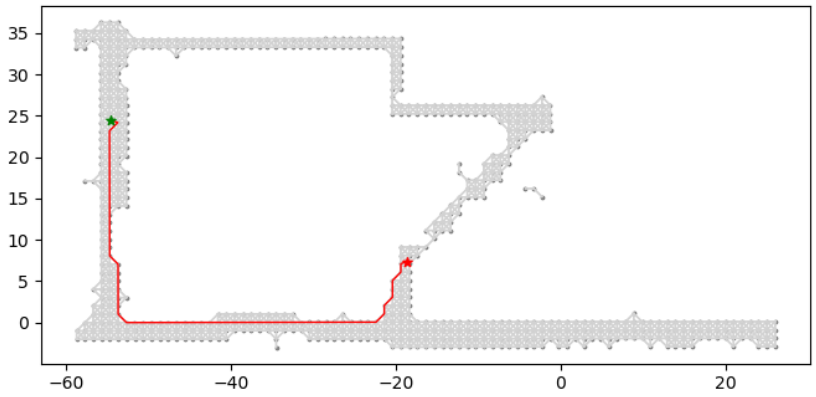

Figure 6: Optimal path returned by A* from start (red star) to goal (green star).

Implementing this in Rviz, we can now plan a path for the car to follow within the map, as pictured in Figure 7:

Figure 7: Pictured left is a case where the dilation is too large causing the car to take a much longer path to the goal point (red). Pictured right is a case with the 17x17 dilation, in which the car takes a shorter path to the goal.

2.4 Pure Pursuit Path Follower

Once we have a path planned, our next step is to follow said path. We accomplish this by using a more stable pure pursuit controller.
We model the the closest point $\hat{P}_3$ on the line to the robot with respect to two points on the path:

$$\hat{P} = \hat{P}_1 + u(\hat{P}_2 - \hat{P}_1)$$

where $\hat{P}_1$ and $\hat{P}_2$ are two points on the path, $\hat{P}$ is the closest point to on the path to the robot, and the scalar value $u \in [0, 1]$ specifies the proportional of the line segment from $\hat{P}_1$ to $\hat{P}_2$ where $\hat{P}$ lies.

$$u = \frac{(\hat{P}_3 - \hat{P}_1) \cdot (\hat{P}_2 - \hat{P}_1)}{||\hat{P}_2 - \hat{P}_1||^2}$$

We find points $\hat{P}_1$ and $\hat{P}_2$ by calculating the distance from the robot to each point on the path, and the two closest points are then $\hat{P}_1$ and $\hat{P}_2$.
Once we have the minimum distance between the robot position and the path, we also have the set of points on our path where our robot may meet the path: any point between $\hat{P}$ and the goal pose (i.e. any point in front of the robot).

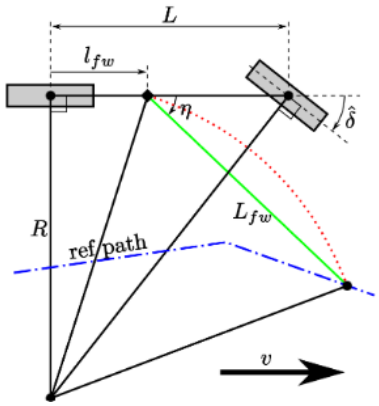

Figure 8: Kuwata, Yoshiaki & Teo, Justin & Karaman, Sertac & Fiore, Gaston & Frazzoli, Emilio & How, Jonathan. (2008). Motion Planning in Complex Environments Using Closed-loop Prediction. 10.2514/6.2008-7166.

We then choose a dynamic lookahead distance L_{fw} using the turning radius $R = \frac{L}{\tan \delta}$. We decided that it makes sense to set the lookahead based on the speed of the vehicle (increasing as our speed increases) and the turning radius (our turning radius decreases as our steering angle increases, meaning that we have a shorter lookahead when making turns and a longer lookahead when we are following a straight path).

We also recognize the need to place upper and lower bounds on our lookahead distance. For example, our turning radius $R \to \infty$ as our steering angle $\delta \to 0°$

and having our lookahead $L_{fw} = \infty$ would not be useful. We also want to include the physical constraint of having a maximum steering angle and thus a minimum steering radius R_{min} into our model. We converged on the following calculation:

$$L_{fw} = \epsilon \cdot \begin{cases} R_{min} & R \leq R_{min} \\ R & R \leq 3 \cdot v \\ 3 \cdot v & otherwise \end{cases}$$

where $\epsilon \in (0, 2]$ is used to scale the lookahead (in most cases) between $2R_{min}$ and $2R$. Different ϵ values were tested as part of our evaluation of the model's performance on the robot.

Our final point of consideration was choosing an offset l_fw from the rear axle of the robot. Initially, our implementation was of standard pure pursuit, using $l_{fw} = 0$. However, we thought that it was leading to erratic swings in the path of the robot, so we decided to set $l_{fw} = \frac{L}{2}$ as an estimate for the distance from the rear axle to the robot's center of gravity. We marked a noticeable decrease in the erratic steering of our robot after this change was made.
Our combined A* and Pure Pursuit can then be seen here.

3 Experimental Evaluation

We tested the performance of our localization implementation in both simulation and the real world.

3.1 Simulation

In simulation, we individually tested the primary components of our localization algorithm: the motion model, sensor model, and combined model. Additionally, we tested our A* and Pure Pursuit implementations.

3.1.1 Particle Filter

When testing the motion model, we published the particles on the Stata basement map in Rviz. We used the pose estimate tool to reposition the robot and observed the resulting particle distribution and estimated pose of the robot given our motion model. We tested both a deterministic model and one with noise from a normal distribution centered at 0, and varying standard deviations. Noise was applied to x, y, and theta.

We performed the same process with the addition of the sensor model. We overlaid our localization estimate with the laser scan to qualitatively evaluate how well our localization aligned with the ground truth. Due to latency in the simulation, we decreased the number of particles from 500 to 200. The combination of 200 particles (100 post-downsampling) greatly reduced latency.

We rigorized our evaluation by quantifying the translational error between our estimate of the robot's pose and the ground truth pose.

$$\epsilon_{pose} = \sqrt{(\Delta x^2 + \Delta y^2)}$$

for $\Delta x = x_{real} - x_{obs}$, $\Delta y = y_{real} - y_{obs}$

We obtained ground truth odometry pose values through a provided ros2 topic. To track error over time, we turned on our wall-follower and graphed the translational pose error. We varied noise levels by adjusting the standard deviation of our sampling distribution, as depicted in Figure 3 below.

	σ (m)				
	0	**0.01**	**0.03**	**0.5**	**3.1**
Trial 1 Error	0.2	0.02	0.2	4	7
Trial 2 Error	0.1	0.04	0.3	5	8
Average Error	0.15	0.03	0.25	4.5	7.5
Average Total Error	0.143			6	
	2.486				

Figure 9: Simulated error for varying noise levels on a straight line drive.

Simulated error was also plotted for one trial and it was confirmed that error remained stably low across the run, for the noise parameter we ultimately selected (0.01).

3.1.2 A* and Pure Pursuit

To evaluate our pure pursuit controller, we simulated multiple paths and noted the robot's performance based on the average distance to the path throughout its traversal. We tested multiple ϵ values including 1.0, 1.5, and 2.0. As expected, we found that the robot's performance when traversing the path using a longer lookahead distance was much less erratic than with a shorter lookahead.
It was also in simulation testing of the pure pursuit controller that we encountered large oscillations as the robot converged on the path. This was remedied by using a longer lookahead distance and by increasing the l_{fw} value to the approximate value of the robot's center of gravity.
Once we settled these issues in simulation, we felt ready to test it on the robot.

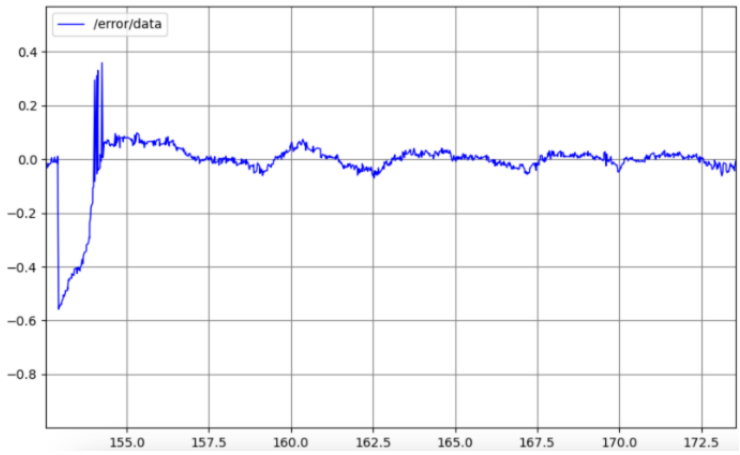

Figure 10: Error for a simulated drive.

3.2 Real World

3.2.1 Particle Filter

Once we were satisfied with our algorithm's performance in simulation, we tested it on the robot. We performed a similar process as in simulation but had to measure error slightly differently since we no longer had a ground truth topic to grab the real pose. Instead, we performed a qualitative and quantitative assessment of our robot's performance as follows.

We initialized the full localization algorithm and started the provided Rviz map in order to visualize our particles and pose estimates. We were also able to visualize how well our estimated laser scan aligned with walls in the map.

In our first test, we aligned the robot at a distance of one meter from a wall and initialized the pose of the robot to that same position and location on the map in RViz. Essentially, we are starting the robot in the same spot both in the real world and on the RViz map. Then, we drove the robot through varying locations of the Stata tunnels and observed how well the laser scans aligned with the map walls.
We then performed a quantitative assessment of our localization by starting the robot at a known location matching in Rviz and real life. We then drove the robot until some stopping point and compared the robot's ending position with the Rviz estimate using known landmarks both in the real world and the map. We then measured the orthogonal distance from the wall (real) and the estimated distance from the wall (RViz), and computed the error or difference between these measurements. We tested this on a straight hallway, a short curved hall, and a longer loop. Error estimates are provided in Figure 7 below:

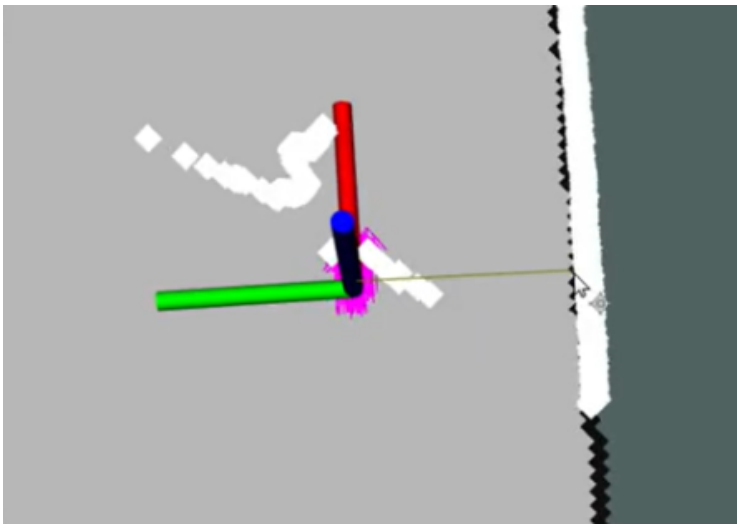

Figure 11: Distance from the wall as measured based on the robot's calculated position in the map, (thin olive-colored horizontal line) pulled from RViz.

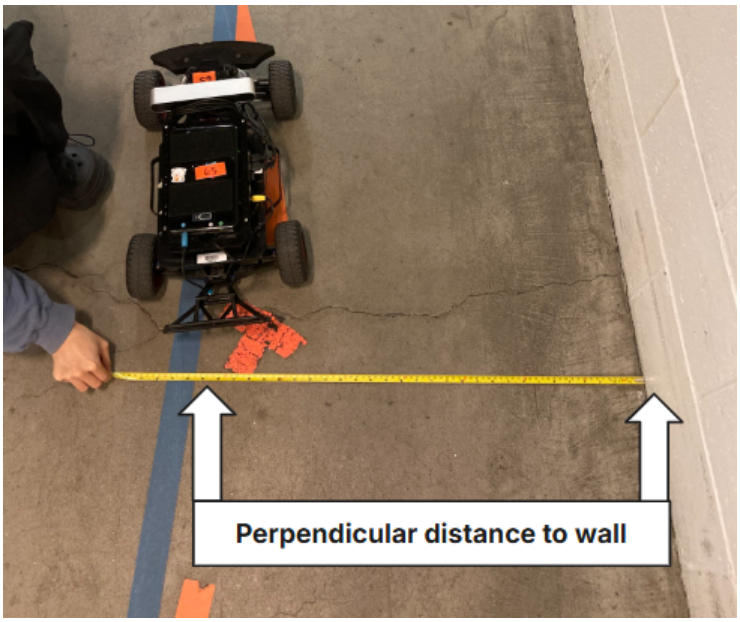

Figure 12: Real life measured distance from the right wall for corresponding trial with above screen capture from Rviz (prior figure.)

	Trial 1 (Straight)	Trial 2 (Short Curve)		Trial 3 (Longer Loop)	
RViz Meas. (cm)	144	175	74.9	275	110
Actual (cm)	139	182	75	269	114
Error (cm)	5	7	0.1	6	4
Average Error (cm)	4.42				

Figure 13: Error from real position measurements vs calculated location.

We see that the localization performed accurately, with an average error of just 4.42 centimeters. In the stata environment, this level of accuracy would be sufficient to safely navigate obstacles at moderate speeds.

3.2.2 A* and Pure Pursuit

We qualitatively evaluated the performance of our A* algorithm by aligning our car's position with the initial pose estimate in Rviz and then choosing a variety of goal positions. From there, we observed the projected path and ensured that it was did not cross through impassable structures and did not have any noticeable unnecessary deviations from the straight line connecting the initial and goal positions. We checked the performance of our A* algorithm in simulation so we proceeded to qualitatively observe the pure pursuit controller's performance. After planning a path, we ran the car to ensure, roughly, that the car stayed on the path both in Rviz and the real world. We also made sure that the car reached the goal position set on the Rviz map, by comparing landmarks on the map with the real world.

To more rigorously test the performance of A*, we measured the time with which the algorithm was able to plan a path. We define path planning time as the time between setting a goal position to the path appearing in Rviz. The time was measured using the Rviz clock.

We also modulated some of the system parameters to improve performance: boundary padding level and cell size. Boundary padding refers to the size of the kernel used to "inflate" the map and the cell size determined how often we sampled the occupancy grid representation of the map. Ultimately we found a consistent real world performance with the boundary padding level set to 17 and the cell size set to 7 (i.e. use only every 7 pixels from the occupancy grid).

	Time to plan path (s)
Test 1 (short turn)	11
Test 2 (long stretch)	12
Test 3 (med. turn)	11
Test 4 (med. turn)	11
Test 5 (long stretch)	10
Average Time	11

Figure 14: Real world path planning times for A* algorithm.

4 Conclusion

Overall our localization implementation on the robot with particle filtering was highly successful and the robot was able to accurately deduce its location to within a 4.4 centimeters of the true location. The algorithm performed well, especially along complex geometries as the sensor model had more unique information to pinpoint the location in the space. However, an area for improvement could be in situations with less distinct features, such as when traveling down a straight, plain hallway, as in this case the robot has less information to determine where exactly it is in the map.

Further work could also be done to ensure the particle filter works accurately in real time, even at higher speeds. This would require optimizing our code to be more efficient and experimenting with different angular noise ranges to reduce the observed lag in angle prediction.

Additionally, our A* and Pure Pursuit implementations were also successful, with an average planning time for A* of 10 seconds.
One next step we could take with A* would be to try lower levels of map discritization, to increase the speed of the algorithm. For Pure Pursuit, an additional step would be tuning the lookahead distance to perform better around corners.
Additionally, one next step we could take would be to implement a full SLAM system in which the robot is actively mapping its environment while localizing itself within the space. Further, we could implement and compare other path planning algorithms such as RRT to A*.

5 Lessons Learned

Sera Hamilton

I learned the importance of dividing up work and staying on the same page. It was very helpful to have good communication when team members were working more individually. Then when we came together at meetings, it was useful to ensure everyone was on the same page and understood what people had done individually and where to move on as a team. On the more technical side learned about the difficulties of finding the right balance between wanting to create a more accurate representation of the map with a smaller cell size but sacrificing runtime because of this.

Carlos Sanchez

One of the key lessons I learned while working on this project is the importance of being adaptable when plans change. Staying engaged in the process as things shifted kept everyone on track and aligned. Additionally, partner programming proved to be an invaluable technique, allowing us to brainstorm and problem-solve together in real-time. This approach not only improved the quality of our work but also enhanced our communication as a team. From a practical standpoint, keeping a shared document for notes and useful commands was incredibly helpful. It served as a central resource, making it easier to stay organized and access essential information quickly. Lastly, I've learned that getting proper rest is crucial—going to bed before 2am made a significant difference in productivity and focus the following day!

Yuewei Liu

It's extremely important to make sure all team members are on the same page about the state of the code and what problems are currently being worked on and by whom. One challenge we encountered this lab was that due to the nature of collaboration with git, people were often coding asynchronously from their local copies of the code. This made it unclear who was working on what part, and at times redundancy and conflicts occurred that could have been avoided with more explicit communication. This lab also taught me about the thought that goes into designing a system with design constraints (for the particular hardware) such as functioning at high frequencies. It was cool to see how lookup tables and precomputing for discretized inputs could drastically decrease runtimes.

Miguel Padilla

I learned a lot about the importance of organization, time management, and effectively communicating the division of work. With this lab we had clearer deadlines of when we wanted certain parts to be finished. We also more strictly laid out times when everyone was available to meet and enforced meetings outside

of class time. This helped us feel much more organized as compared to previous labs. Additionally, I learned about the difficulties of debugging and pinpointing the problem. For instance, some issues we ran into were less concerned with our conceptual understanding of the algorithm and more with debugging our actual implementation.

Willow Pickernell

I learned about the importance of being active in a collaborative process, and that communication is critical to adapt to changes. During this lab, there were some changes to the distribution of work and I made assumptions regarding these changes that I shouldn't have and which harmed our ability to work together. Its important to discuss changes and ask questions and be involved for the purpose of teamwork. In terms of technical lessons, I learned about the A* algorithm, how it calculates paths, and the different heuristic functions used to estimate distance.