# Lab 6 Report: Localization and Path-Planning Stack for Autonomous Race Car

Team 4

Adelene Chan
Alan Chen
Paul Gregory
Eghosa Ohenhen

6.4200 Robotics: Science Systems

April 23, 2025

## Contents

# 1 Introduction (Paul, Adelene)

We previously developed a probabilistic localization system for a miniature autonomous racecar operating in a known map. We implemented Monte Carlo Localization (MCL), also known as a particle filter, to estimate the racecar's pose using LiDAR data and a known map. The MCL algorithm maintains a set of potential locations of the robot, and continuously updates them using probabilistic motion models and sensor measurements.

Once we were able to identify the robot's pose using our localization node, we could move onto a higher level of control, path planning. The focus of this second lab involved two primary parts: determining the path and following the path. Given a known map, start and end points, and the known pose of the robot, we could use a path planning algorithm to generate a safe path that avoids obstacles and running into walls.

In this lab, we chose to implement and compare two path planning methods: a search-based method and a sampled-based method. The two methods have pros and cons in computation time, path optimality, and implementation difficulty. We were also tasked with implementing a pure pursuit controller that allows the racecar to follow a variety of paths by adjusting the steering angle.

This lab was more complex with more moving parts and dependencies, such as dependency on an accurate and fast localization. We chose to evaluate several metrics including path planning success, planning time, and pure pursuit evaluation via distance to the path. This allowed us to evaluate the reliability and quantify each part. Building reliable parts that integrate properly will be important as we move towards developing more complex systems for our autonomous racecar.

# 2 Particle Filter (Alan)

We use a particle filter approach for localization.

We maintain a list of poses of the car (we will call each a "particle"), where we expect that the true pose of the car is close to at least one particle, and that the mean of the particles is a good estimator of the true pose.

We accomplish this by updating the particle list using sensor data; We make use of

1. Odometry from wheels, which estimates the current linear and angular velocity of the car, updated at approximately 50 Hz. We explain how we update our particle list by Odometry data in 2.1.
2. LiDAR measurements, which provide 1081 distance measurements ranging from $-135$ degrees to 135 degrees from the forward direction. Our LiDAR updates at approximately 50 Hz. We explain how we update our particle list using LiDAR data in 2.2.

## 2.1 Motion Model (Adelene)

As the racecar moves, we receive wheel odometry messages which include translational and angular velocities and the time difference between last and current messages. This allows integration to obtain a displacement pose, Eq. 1, with respect to the previous robot frame, $k-1$.

$$\mathbf{\Delta x} = \begin{bmatrix} dx \\ dy \\ d\theta \end{bmatrix} \tag{1}$$

We convert the $k-1$ poses into transforms and compose the transforms to give us the transform from the map to the current $k$ robot frame.
Wheel odometry will not be completely representative of the actual motion of the racecar so we add noise to the term $\mathbf{\Delta x}$. $\mathbf{\Delta x_{noise}}$, Eq. 2. We chose a Gaussian distribution for $dx$, $dy$, $dz$ with standard deviations of 0.1, 0.025, 0.1, respectively.

$$\mathbf{\Delta x_{noise}} = \begin{bmatrix} dx_n \\ dy_n \\ d\theta_n \end{bmatrix} \tag{2}$$

The transform is applied to each of the current particles $\mathbf{T}_k^W = \mathbf{T}_{k-1}^W \cdot \mathbf{T}_k^{k-1}$. Eq. 3 shows what transform composition for a single previous pose $k$ and a $\Delta x$ with noise.

$$\mathbf{T}_k^W = \begin{bmatrix} cos(\theta_{k-1}) & -sin(\theta_{k-1}) & x_{k-1} \\ sin(\theta_{k-1}) & cos(\theta_{k-1}) & y_{k-1} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(d\theta_n) & -sin(d\theta_n) & dx_n \\ sin(d\theta_n) & cos(d\theta_n) & dy_n \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

The resulting transforms are then converted into poses of the form shown in Eq. 4

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} \tag{4}$$

We return the list of these poses which present a new set of particle guesses of the new robot poses.
With the addition of noise, which is necessary to capture the imperfect translation of wheel odometry under real world disturbances, the motion model will lead to a divergence of particle poses. This motivates the sensor model which will take in these particles and return a set of more likely particles.

## 2.2 Sensor Model (Alan)

In our sensor model, we update our particles list by:

1. Computing a score for each particle.

   We do this through ray-tracing each ray starting at the particle on a constant pre-given map; We compute the score from the extent the sensor data matches distances obtained from ray tracing, where a close match results in a higher score.

   We explain details of computing the score in 2.2.1.

2. Using those scores, sample a list of new particles independently.

   Each one is sampled by first sampling a random particle from the old particles list, with weights proportional to its score; We then add a Gaussian noise to produce a new particle.

   The parameters of the Gaussian noise is adjusted empirically based on localization performance.

As the car moves, the distance of the particle closest to the ground truth to the ground truth generally increases. We aim to decrease it with our sensor model. We assume that old particles closer to the ground truth will be assigned higher likelihoods; We will then sample many new particles a Gaussian noise away from each of the high-likelihoods old particles, where with a high probability some of them are closer to the ground truth than all the old particles.

### 2.2.1 Scoring Particles

We need to produce a score (sampling weight) for each of the old particles, with the goal of assigning higher score to those closer to the ground truth.

We compute the score for a particle by:

1. For each angle $i$, compute the ray tracing distance $r_i$ from that particle at the specified angle, using a constant pre-given map.

   Let $o_i$ be observed sensor measurement of that angle; We look up a score $s_i = T(o_i, r_i)$ for that ray, where $T$ is a pre-computed constant table representing the extent the observation $o_i$ is compatible with $r_i$.

2. We output $S$ as our score, where

$$S = \left( \prod_{i=1}^{n} s_i \right)^{1/n}$$

$T$ can be thought of as a probability distribution: for a fixed $r$, the distribution

$$P_r(o) \propto T(o, r)$$

Can be thought of as the probability of obtaining measurement $o$ given that we traced a ray of distance $r$, where we are assigning higher score to more likely measurements.

Because of this, we start by setting $T$ to the probability densities from an example probability model $P^*(r|o)$ given in the starting repository. We then adjusted the value of table $T$ empirically.

Our table $T$ is parameterized by a small set of parameters, which we adjust based on localization performance, measured by visual inspection of whether sensor distances plotted from the mean of the particles matches the walls.

This adjustment is done in simulation as well as on recorded data from the real racecar.

# 3 Path Planning

## 3.1 Pre Processing: Map Dilation (Paul)

To ensure that our path finding algorithms do not get dangerously close to obstacles, we pre-process our map. We inflate obstacles so that our paths will not get too close without a collision detected. To do this, we use a morphological dilation, which slides a kernel along our map grid to extend the boundaries of our obstacles, to ensure a proper safety buffer.

## 3.2 Motion Planning Approach No. 1: A*

### 3.2.1 Workspace Representation: 2D Grid(Eghosa)

We used a 2D grid to represent the workspace, i.e. the full map with obstacles and free spaces, to make the world information digestible (as a discretized state

space) for the search algorithms. The map topic publishes an OccupancyGrid message when the racecar simulation is launched. The representation grid is 1730 pixels by 1300 pixels, with free cells marked as 0, occupied cells marked with 100, and indeterminate cells marked with a -1.

#### 3.2.2 Tree-based Search Algorithm: A* (Eghosa)

In this path planning method, we used the A* search algorithm to produce a shortest path from a single source. A* generate moves by creating a search tree of nodes, exploring the "cheapest" next. The variant of the A* search algorithm we used for planning in discrete state space can be modelled as follows:

- *Input*: 2D Grid Representation
- *Action Space*: 8-connected neighbours
- *Cost Function*: Movement cost from position $p$ at time t to position $p$ at time t + 1 represented as

$$c(p_t, p_{t+1}, O) = \begin{cases} \sqrt{2} & \text{if diagonal move} \\ 1.0 & \text{if straight line move} \\ 100.0 & \text{if } p_{t+1} \in O \end{cases}$$

  where $O$ is the set of cells marked as obstacles.

- *State Space*: possible robot states represented as coordinates $(x, y) \in S$ where S is set of coordinates in the 1730 x 1300 grid representation.
- *Heuristic*: The estimated cost from node x to goal, modelled by Euclidean distance.

$$h(p_t, p_{t+1}) = \sqrt{(p_t^x - p_{t+1}^x)^2 + (p_t^y - p_{t+1}^y)^2}$$

For our A* variant, we used a non-uniform cost function as diagonal moves create a longer path than straight line moves, thus must be weighted accordingly. Using domain knowledge to bias search order by using a heuristic that favors nodes closer to the goal, the overall movement cost in our A* search algorithm is evaluated as follows:

$$f(p_t, p_{t+1}, O) = c(p_t, p_{t+1}, O) + h(p_t, p_{t+1})$$

Path Generation

Our A* variant outputs a hashmap of nodes visited on the optimal path. To get our path, we traverse the hashmap backwards starting from the end node and generate a list of grid coordinates in which we then turn into real world coordinates.

## 3.3 Motion Planning Approach No. 2: RRT*

### 3.3.1 Path-planning - Sample-based RRT* (Paul)

We also implement the Rapidly Exploring Random Tree Star (RRT*) algorithm for path planning. Essentially, this algorithm aims to build a tree from our starting point to our ending point that avoids collisions by randomly sampling points in the map. This tree is made up of nodes in the continuous 2d space of our map, each node with an x coordinate, y coordinate, parent pointer, and cost value representing the length of the path from this node to the start node following parent pointers. As our algorithm builds a tree in continuous 2d space while our map is a grid, to check for collisions we apply a world to grid transformation that finds the corresponding cell in the grid in order to check our point against the map.

At each iteration of our algorithm, we do the following steps:

1. With probability=0.3, sample the end goal; otherwise sample a random (x,y) point on the map.
2. Find the nearest node that is currently in the tree to our point.
3. Steering: from this nearest node, we create a potential new node that is a step size of 0.3 meters in the direction of our sampled point. If the path from the nearest node to the potential new node is collision free, we add this node to the tree.
4. To find our new node's parent pointer, we find the node within a 1 meter radius that if connected to our new node does not introduce any collisions and minimizes the cost of our new node.
5. Rewiring: within the same 1 meter radius, we check if any nodes can have their cost decreased by making our new node their new parent. If so, we update the necessary parent pointers.
6. If we reach our goal node, we follow our parent pointers backwards to the start node to form our path and return.

If, at the end of 5,000 iterations of this algorithm, we have not found a valid path, then the search has failed. A visualization of our algorithm can be seen in Fig.1.

## 3.4 Post Processing: Path Sparsification (Paul)

After each of our path planning algorithms, we are left with a collisionless path from our starting point to our end goal. However, we can do better. Our paths often contain unnecessary turns and curves. We implement a path smoothing algorithm to "straighten out" our paths. We do this by checking if shortcuts exist between non consecutive points in the path. From our starting point, we

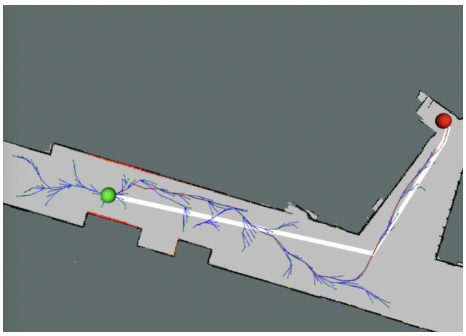


Figure 1: A visualization of the tree produced by our RRT* algorithm, with the final path after smoothing in white.

check how far ahead we can shortcut without collisions, starting with the end node. We skip ahead as far as we can and repeat this process until we reach the end of our path. Fig.1 shows an example of our path smoothing algorithm working on a path found by RRT*.

# 4 Path Following (Adelene)

## 4.1 Controller

After determining a path, we have the racecar follow the path using a pure pursuit controller. This controller works by looking for "lookahead point" on the path and then adjusting the steering angle towards that point, as shown in 2. Since we receive the trajectory from the path planning as a list of points, we approximate the path with line segments connecting each point. The pure pursuit controller is implemented as a ROS node that listens for the racecar's pose returned from the localization node. At each time a new pose is received, it goes through the following steps:

1. Current pose: $(x_p, y_p, \theta_p)$, trajectory: $[(x_0, y_0), (x_1, y_1), (x_2, y_2), ...]$.
2. Get the **nearest point** by projecting the point onto each segment and returning indices of segment corresponding to minimum distance.
3. Look for **intersection** of a circle of a set look-ahead distance and the trajectory path.
4. Iterate through the trajectory points starting from the nearest point previously returned. At each line segment, look for an intersection with the circle by computing the discriminant of the solution to the intersection equation: $(x - x_p)^2 + (y - y_p)^2 = L^2$. The sign of the discriminant leads to different edge cases.

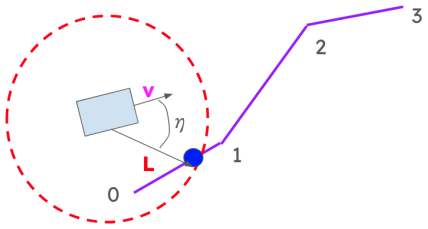



Figure 2: This shows the high level concept of pure pursuit that takes advantage that the racecar will constantly move towards the intersection without ever reaching it, given a large enough lookahead distance.

(a) **No Intersection:** Discriminant is less than 0 or intersections outside segment.

(b) **1 to 2 Intersections:** Choose the intersection closer to the second point in the segment.

(c) **More than 2 Intersections:** Track index of last point passed to avoid going backwards, such as in sharp turns.

5. After identifying the intersection point, we return this as the **lookahead point**.

6. We use that to calculate the lookahead point to calculate the **new steering angle** of the car based on the difference between the current heading of the racecar and the angle to the lookahead point, $\eta$ which allows us to compute the curvature to travel along: $R = \frac{L}{2 \sin \eta}$. Finally, we can calculate the steering angle: $\delta = \tan^{-1}(\frac{1}{R})$

7. Finally, the steering angle and speed are sent to the racecar.

## 4.2 Tuning and Controller Evaluation

After initially fixing lookahead distances, we updated the pure pursuit to have variable lookaheads that accommodate different levels of curvature as shown in 3. Essentially, at higher curvatures, such as straight lines, the lookahead distance will be higher than at smaller curvatures.

While tuning, we realized the difference that localization accuracy and frequency made. In simulation, the pure pursuit had a smooth following of the path that could use smaller lookaheads and be able to react quickly with larger and fast-switching steering angles. However, in the realworld, the pure pursuit would have aggressive turning that would lead to irrecoverable overshoots. Using a

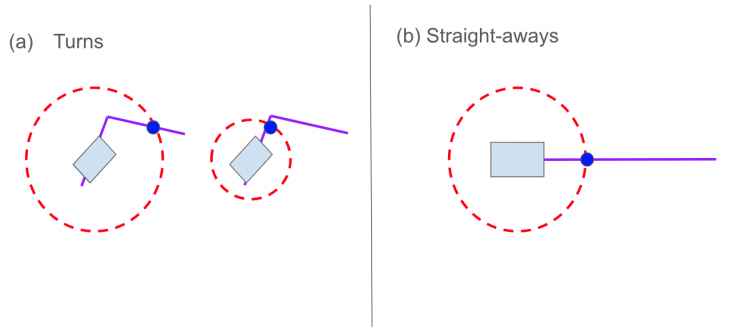



Figure 3: (a) This shows that the lookahead distance has a large effect on the lookahead point at tight curvatures. If the lookahead point is too large, then the racecar will not track a turn as well. In (b), we see that the lookahead distance is less critical to proper tracking.

lookahead range of 0.75 to 1.0m worked well in simulation but was generally too small since it would result in an aggressive steering angle on the real racecar. Our tuning was largely empirical, using a published lookahead marker to see and testing various lookahead ranges. The two figures 4 and 5 demonstrate the real racecar tracking the grid-based generated path for a shorter path navigating past a pillar in the Stata Basement and a longer path curving around a corner. These results show a successful tracking of the path trajectory, however there is a slight offset in a majority of the tracking which can be the result of the larger lookahead distance that does not require a large correcting steering angle.

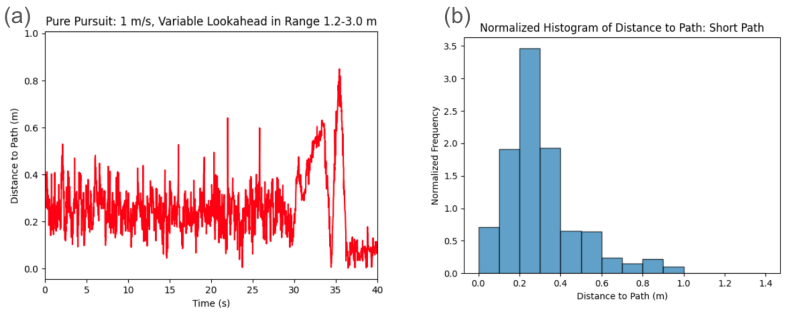



Figure 4: (a) This shows the error between the real racecar's actual pose and the shorter path as the car traverses the trajectory. The oscillation between 0.2 and 0.3 m shows that the car was following the trajectory at an offset before arriving at the goal in the final parts of the path. (b) This visualizes the same error data as a histogram demonstrating that the error was more likely to be in the range of 0 to 0.5 m.

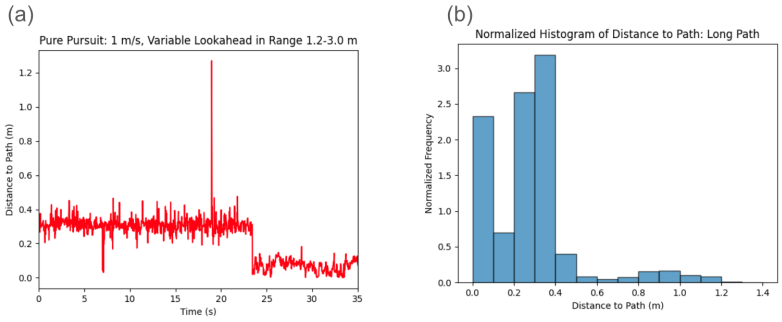



Figure 5: (a) This shows the error between the real racecar's actual pose and the path as the car traverses the trajectory for a longer path curving around a corner. Like the shorter path, the car appears to track the trajectory at an offset before arriving at the goal in the final parts of the path. (b) The histogram shows that the car's error is usually not more than 0.4 m

# 5 Experimental Evaluation

## 5.1 Localization

### 5.1.1 Simulated Racecar Evaluation - Noisy Odometry (Paul)

In a real-world robotics system, sensor data is rarely perfect, unlike in simulation. To test the resilience of our localization system under realistic conditions, we evaluated its performance using simulated odometry data corrupted with Gaussian noise. By comparing the estimated pose against ground truth location, we can quantify the error introduced by noisy odometry.
To obtain this data, add gaussian noise at a specified standard deviation to our input odometry and publishes this to be used by the particle filter. We then capture the error over 10 seconds while our wall follower is running. Fig.6 is our plot with no added noise, Fig.7 is our plot for standard deviation of 0.1, and Fig.8 is our plot for standard deviation of 0.3. As you can see, when no noise is added our localization performs perfectly in the simulation, and our error gets worse with the more noise we add.

### 5.1.2 Simulated Racecar Evaluation - Convergence (Eghosa)

Convergence of the pose estimation is an important metric that signifies how confident the controller is about the pose of the robot at that time $t$. Naturally, a fast convergence plot with minimal spikes afterwards is desired. To measure the convergence of the pose estimation particles, we measured the standard deviation of the particles while running localization on a wall follower. All of the convergence plots were generated while following a straight line. To show the robustness of the pose convergence, we plotted the convergence of the robot in

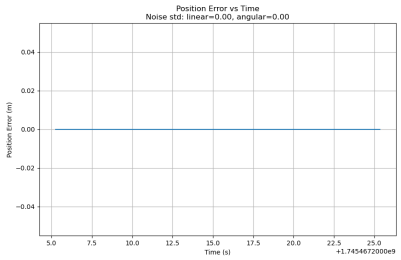



Figure 6: Our error plot when no gaussian noise is added to our odometry data.

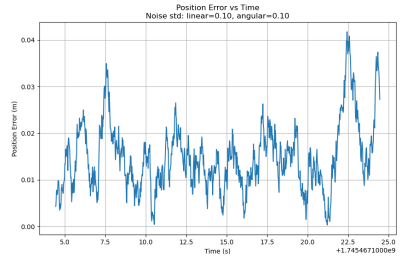



Figure 7: Our error plot when gaussian noise with standard deviation of .1 is added to our odometry data.

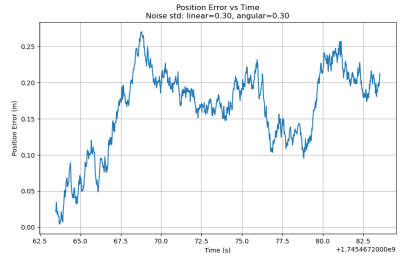



Figure 8: Our error plot when gaussian noise with standard deviation of .3 is added to our odometry data.

3 states: no noise (Fig 9), 0.01 gaussian noise (Figure 10) and 0.1 noise (Figure **??**). As expected, adding Gaussian noise increased the standard deviation of the pose estimate, even when it had converged. Regardless, for all levels of noise the pose estimate converged to a reasonably smaller standard deviation which shows our localization is robust to noise. In Fig. 10 and 11, we see the localization stabilizes around 0.1m for x and 0.025 for y which makes sense as the robot was wall following a straight line.

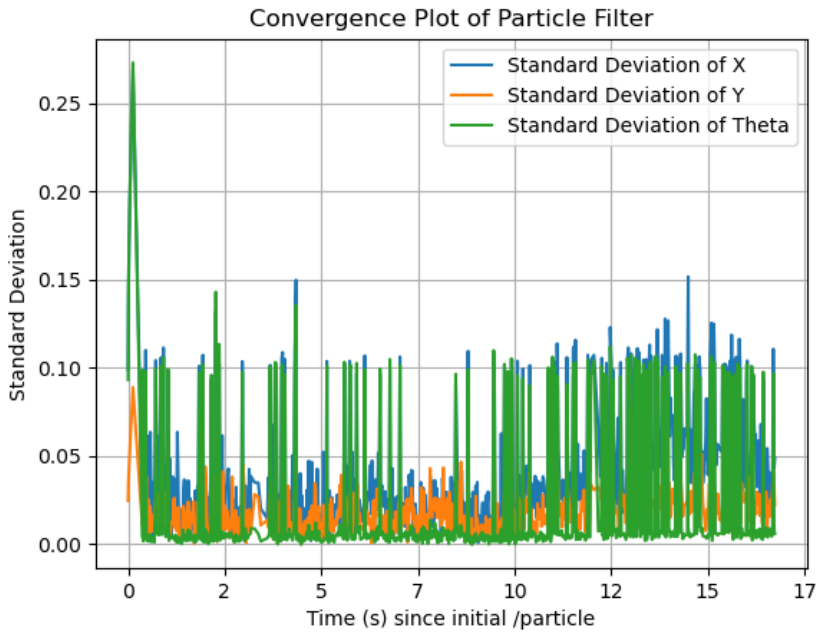



Figure 9: Convergence Plot in Simulator with No Gaussian Noise
The blue, orange, and green show the standard deviations of the spread of the x, y, and theta of the particles. The higher spikes indicate less certainty in the estimated pose. The small amount of noise led to some points of uncertainty but mostly certain pose estimate. This was while running the wall follower.

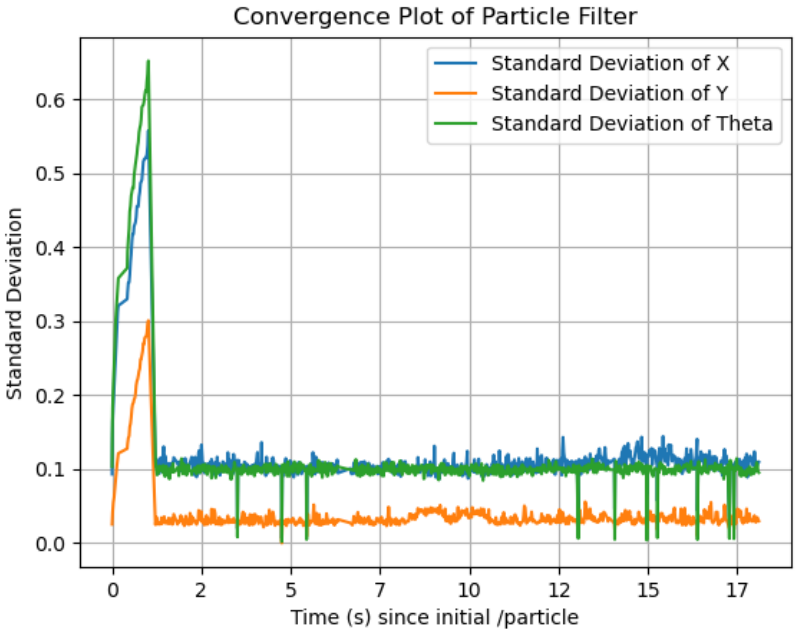



Figure 10: Convergence Plot in Simulator with 0.01 Gaussian Noise

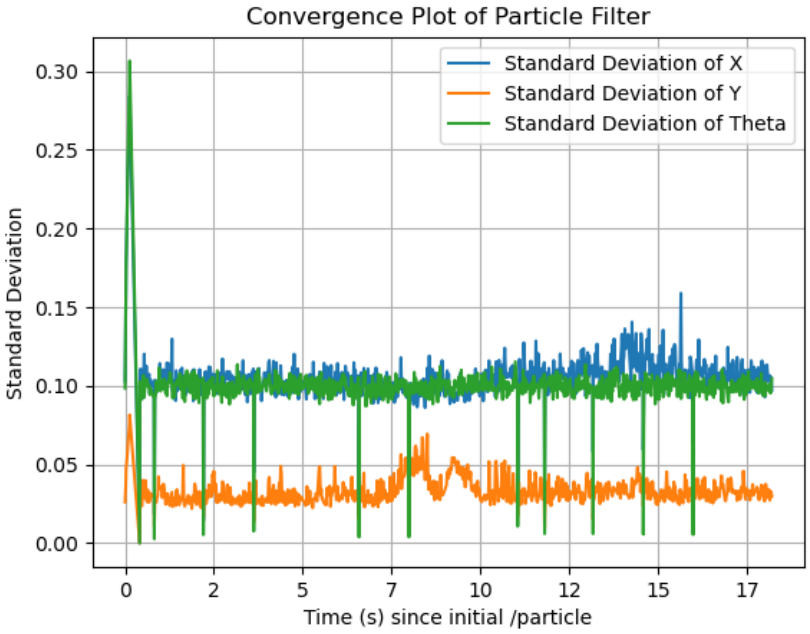



Figure 11: Convergence Plot in Simulator with 0.1 Gaussian Noise

#### 5.1.3 Real Racecar Evaluation (Adelene)

In order to identify correct localization without ground truth odometry like the simulator, we had quantitative and qualitative comparisons. We used a measuring tape and the RVIZ2 measuring tool to compare the physical and simulated distances. We also compared the laser scan with the predicted laser scan as produced by ray tracing with the estimated pose to check for alignment. In the first comparison method, we measured the distance from a unique feature to a marked point. Then we drove the car in teleop to the marked goal location, approaching at different speeds for different trials. Afterwards, we used the RVIZ2 measure tool to measure the estimated pose to the same unique feature, as shown in 12. The results in Table 1 show accuracy within a 10 cm range for the unique hallway whereas there the particle filter would sometimes output incorrect estimates, 4 times the estimated distance, in the uniform hallway where there is a lack of unique features.

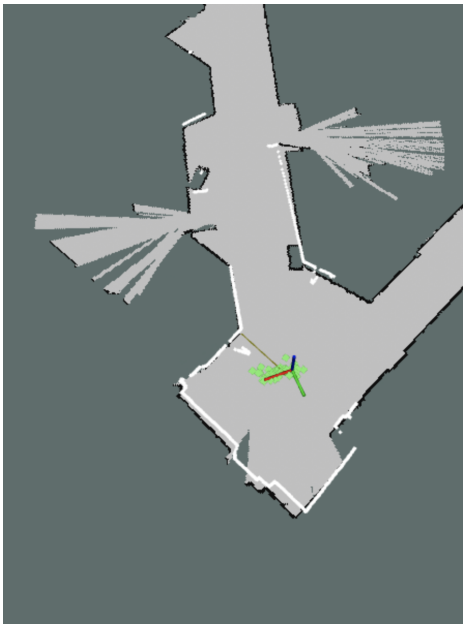


Figure 12: This image shows the estimated pose measured using the RVIZ2 measuring tool, shown as the olive colored line. The green dots are the particles of the particle filter.

Table 1.

| **Location** | **Measured Distance** | **Simulated Distance: Slow (cm)** | **Simulated Distance: Fast (cm)** |
|---|---|---|---|
| Unique Hallway | 218 | 217 | 211 |
| Unique Hallway | 209 | 203 | 202 |
| Straight Hallway | 114 | 115 | 420 |
| Straight Hallway | 121 | 404 | 119 |

Figure 13 shows the localization accuracy of a unique feature in the Stata basement where the actual laser scan lines up with the ray-traced scan of the estimated pose very closely. We note that there exists a line in front of the racecar which was a hardware issue with the LiDAR pointing towards the ground leading it to see a smaller range.

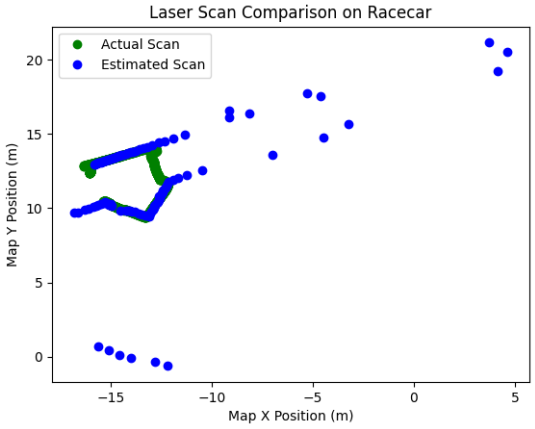



Figure 13: Laser Scan of Stata Basement, Outside Classroom

We also used convergence plots to see how quickly the racecar estimated its pose to a high confidence. Figure 15 shows that in a map area with a high number of unique features, the particle filter leads to higher confidence in its estimated pose as shown by the smaller standard deviations in the convergence plot.

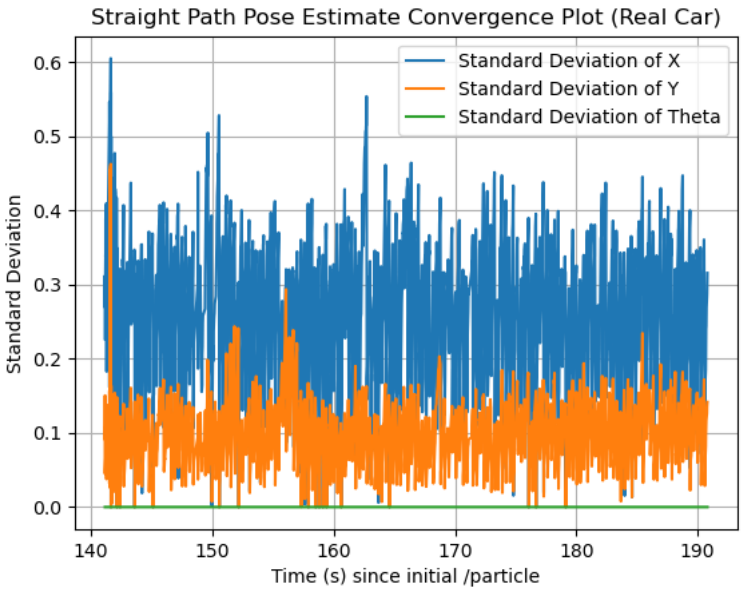



Figure 14: Convergence Plot of Stata Basement, Straight Path Outside Classroom*

The pose estimates converge fairly quickly but has a much greater variance in stdev in comparison to the higher than that of the simulation, oscillating in the 0 to 0.5 range for x and 0 to 0.2 range for y. This makes sense given the greater amount of real world noise.

**The theta estimate was not published, hence why it is 0.*

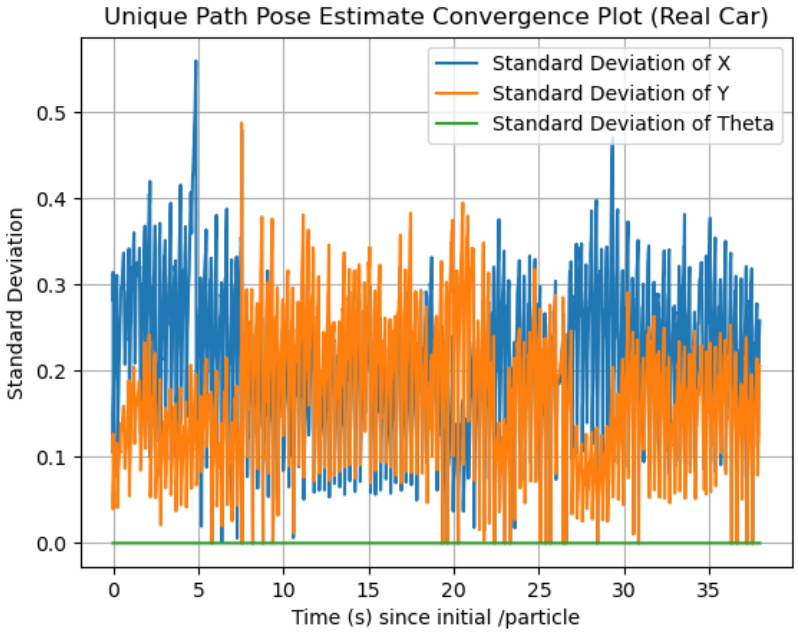



Figure 15: Convergence Plot of Stata Basement, Non-Straight Path Outside Classroom*
For the non-straight path the variance in stdev is similar to Fig. 14. However, this path has a turn from around 10 to 20 seconds. During the turn the stdev of pose estimate error of Y increases to around the X stdev. This is because the car is not just moving along the X axis but also the Y in a turn.
**The theta estimate was not published, hence why it is 0.*

## 5.2 Path-planning Evaluation (Alan)

We evaluate two Path-planning algorithms: RRT* and A* by running them with endpoints that are randomly sampled from free points on the map that are at least 0.5 meters from a wall.

The following plots are created from a fixed set of 300 endpoint pairs. All evaluations are run with a fixed seed.

Unless otherwise stated, we run with the following defaults:

1. The algorithms are instructed to stay at least 0.3 meters away from all walls. We will call this the “safety buffer”.

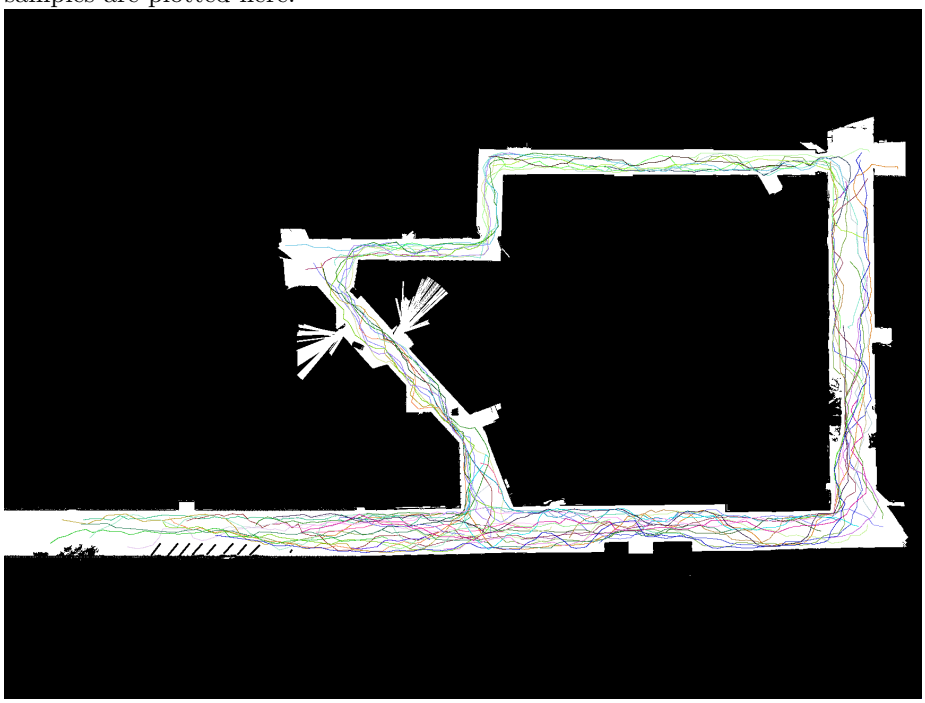
2. RRT* refers to RRT* with smoothing.

3. A* is always run without smoothing; smoothing does not work due to a technical problem.

#### 5.2.1 RRT*

Our $RRT*$ was able to produce 292 valid paths out of 300 endpoint pairs. The rest is unable to find a path within a certain amount of steps. This is expected as the steps needed for $RRT*$ have high variance, and that certain endpoints are difficult for the algorithm.

We first present paths created by our RRT* without smoothing. 60 random samples are plotted here:

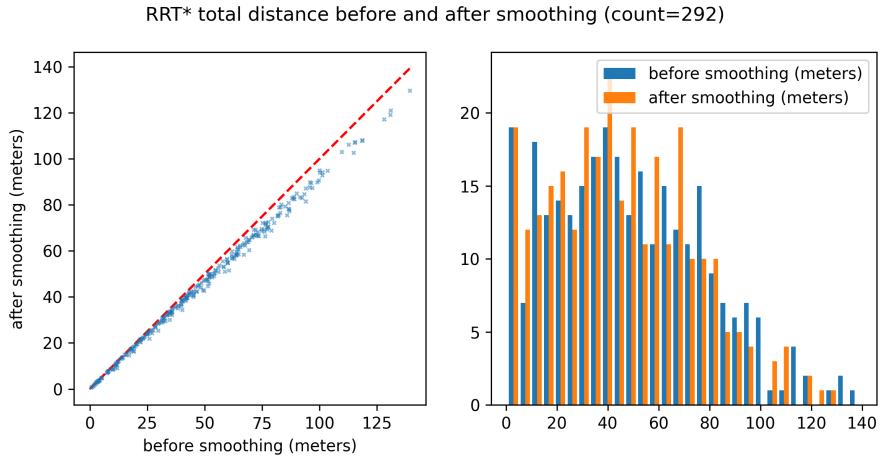


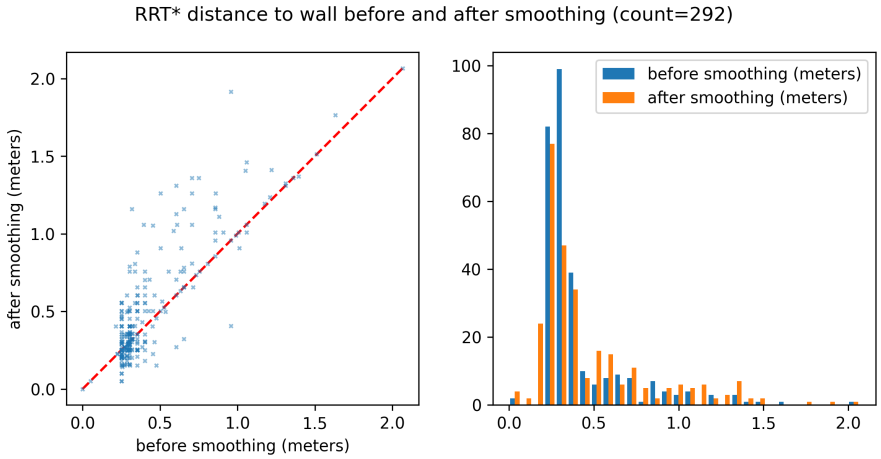
Since our RRT* implementation stops immediately after reaching the target, we expect that the path is usually made out of segments, which is consistent with what we observe.

After smoothing, the paths become:

Where most paths reduce only to a few segments.

We look more closely at how smoothing affects total distance and the minimum distance from a path to the wall:






We observe that smoothing usually decreases distance, but only by a small amount.

We also note that a small portion of the smoothed paths cuts into the "safety buffer" of 0.3 meters. This is expected as we do not ensure the "safety buffer" constraint in the smoothing. This can be compensated by setting a larger "safety buffer", or rejecting smoothed paths that go too close to a wall.

### 5.2.2 A*

Our $A*$ implementation was able to find a valid path for all 300 endpoint pairs tested.

60 samples of paths planned are plotted here:

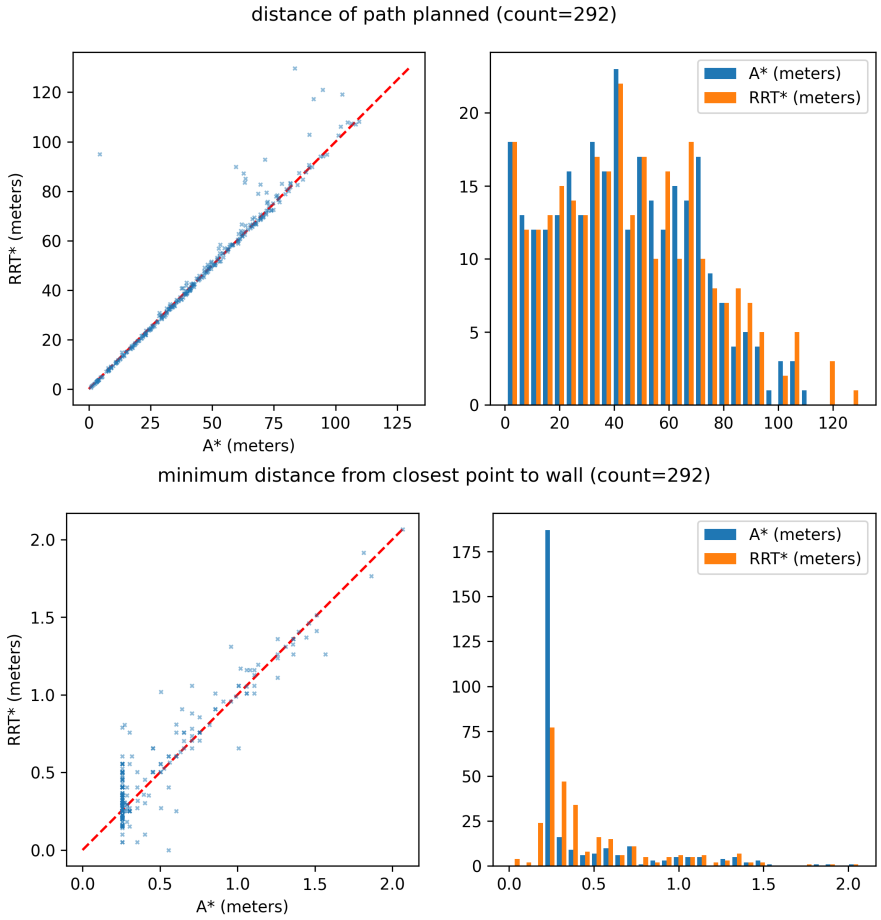
We observe the following behavior of $A*$:

1. There are lots of segments that are orthogonal or diagonal; this is consistent from our $A*$ considering orthogonal or diagonal neighbors are neighbors in the graph.

2. Sometimes segments stay exactly on the edge of "safety buffer". This is consistent with $A*$ needing to find an optimal path.

#### 5.2.3 RRT* and A*

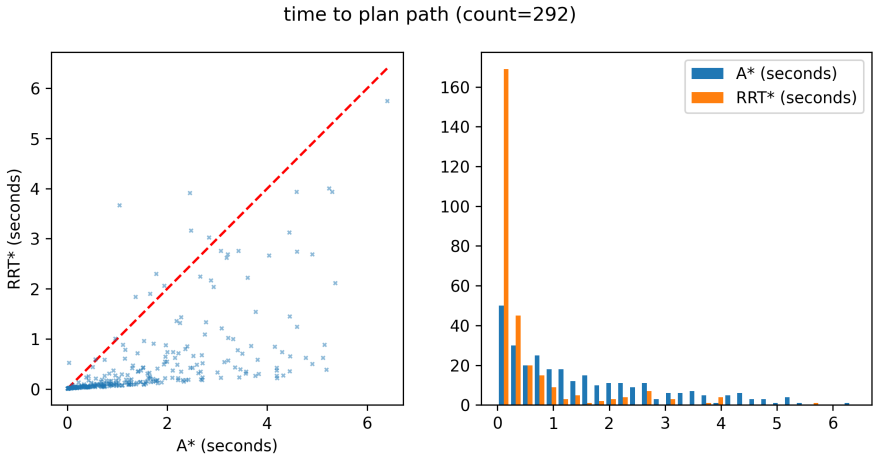
We compare paths created by RRT* smoothed and A*:




We see that $RRT*$ occasionally differs from the optimal path by a significant amount ($A*$ always find the optimal path, so its distance can be used as the optimal distance.)

We also compare the time RRT* takes to produce a path, with the time A* takes.

Note that these data are from a Github Actions Runner, NOT from the racecar.




From these data, we see that RRT* is generally faster than A*.

This is not consistent with our expectations: since A* has a runtime upper bound that is small on our input data, we expect A* to finish quickly and always within a set time. The discrepancy is probably due to our specific implementation.

# 6 Conclusion(Paul)

In this lab, we built on our previously developed Monte-Carlo Localization (MCL) system to develop path planning and path following algorithms. For path planning, we implement both grid-based and sample-based algorithms for finding a collision free path from a starting point to an ending point. For path following, we use a pure pursuit controller to allow the racecar to autonomously follow a given path.

When evaluating our path planners, we found that the grid-based with A* consistently produced optimal paths but had unexpected computational overhead. Conversely, RRT* exhibited faster computation times at the cost of occasional suboptimal paths, a compromise addressed partly by implementing a path smoothing algorithm, however there were still some cases where RRT* was unable to find a path at all.

For our pure pursuit, we found success with variable lookahead distances that could handle different levels of curvature. In the real world, our pure pursuit displayed some difficulty with proper steering angles, highlighting the importance of efficient and accurate localization.

Overall, we were able to successfully implement localization, path planning, and path following on our racecar, enabling it to autonomously drive to a given point. Future work should focus on continuing to fine tune our algorithms both for run time as well as performance, especcially in the real world.

# 7 Lessons Learned

(Adelene)
This lab showed me how to split up a larger project into smaller parts. We split based on evaluation, path planning algorithms, and path following. Even with smaller parts, cross-communication is still important to make sure our endpoints match and that we can effectively test each other's work. We still have room to grow in how we manage our tasks and priorities but this has been a good step in developing a team workflow.
(Paul)
In this lab, I learned about the importance of having synergy between implementation and evaluation, as evaluation can be used for testing an implementation to make it better. I also learned about the importance of documenting code, as when working on a team it can sometimes be very difficult to understand what someone elses code is doing or how to run it.
(Eghosa)
I learned a lot about keeping in touch with a team this lab and also about the difficulty of migrating multiple products into one. I have worked on github repos with multiple people before, but usually we start from scratch, working together to implement an idea. However, for this lab, we have each came in with our own working implementations and side projects and had to integrate them together not only via Github, but also when sshed onto the robot's WiFi. Trying to push our code from the robot to Github was interesting since we were essentially on the same machine, so a lot of the usual abstractions used when collaborating with Github couldn't be used. This was a very valuable insight for me.
(Alan)
In this lab we implemented automated testing and Github actions, which is more useful and efficient than I expected: it is a lot more efficient and reliable than generating each plot by hand. I also learned to use ROS Quality Of Service settings, fixing many flakiness problems with our ROS Nodes.