Lab 6 Report: Path Planning

Team 11

Suchitha Channapatna
Luis De Anda
Amber Lien
Prince Patel

Robotics Science and Systems

April 24, 2025

1 Introduction

Path planning is a fundamental challenge in autonomous robotics as it enables a robot to navigate a map. Within a robotic system, a path planning module is typically situated one layer below a task planner and one layer above a low-level controller. Given a goal position, a path planner is responsible for generating a trajectory from the robot's position to the goal while avoiding known obstacles in the map. The ability to efficiently and safely plan trajectories is essential for navigation, safety, and task completion.

In this lab, we address the path planning and low-level control portions of the pipeline. The goal is to navigate through the environment autonomously. We explore several path planning algorithms and determine whether they successfully generate paths between a start and goal point given a map of the environment. We evaluate several metrics of paths generated in simulation and implement the A* path planning algorithm on the racecar. Once a path is generated, it is simplified via a smoothing operation. Pure pursuit was implemented to allow the racecar to follow these smoothed trajectories.

The execution of path planning incorporates several other modules developed in previous labs. We run a safety controller to ensure that the racecar does not collide with dynamic or unmapped obstacles. Additionally, in implementing the pure pursuit control, the racecar uses an estimated pose as determined by the localization module. Localization allows the racecar to determine its position in a known environment using LiDAR scan data. We implement Monte Carlo Localization (MCL). Our implementation follows the particle filter approach, which maintains a set of hypothesized poses (particles) and updates

their likelihood based on sensor and motion data. This probabilistic method is particularly well-suited for robotics applications because of its robustness to sensor noise and environmental uncertainties.

This path planning framework allows the racecar to navigate a mapped environment with minimal human interference. The implemented pure pursuit control allows for smooth tracking of trajectories and makes use of the previously developed localization module. The implemented MCL algorithm not only provides accurate position estimation, but also effectively handles the inherent uncertainty in robotics systems through a probabilistic framework. Our technical approach prioritizes computational efficiency and real-time performance, making it suitable for deployment on our physical racecar.

2 Technical Approach

2.1 Approach Overview

The path planning and execution problem involves planning, localization, and control. Planning specifies a start and goal and finds a viable path between them, using either sampling- or search-based algorithms, both of which we explore in this lab.

The racecar must estimate its pose to track the trajectory via pure pursuit. Since ground-truth isn't available in real-world settings, we incorporate localization. The localization problem involves estimating the robot's pose (x, y, θ) within a known map given sensor measurements and control inputs. Formally, we estimate the posterior probability distribution $p(x_t|z_{1:t}, u_{1:t})$, where x_t is the robot's pose at time t, $z_{1:t}$ are the sensor measurements up to time t, and $u_{1:t}$ are the control inputs up to time t.

MCL approximates this distribution using weighted particles, each representing a possible pose; its weight reflects the likelihood of being the true pose given sensor measurements. Ray casting generates simulated LiDAR ranges from each of these particles, which can then be compared to the true LiDAR ranges from the racecar. The prediction, update, and resample cycle determines the most likely pose of the racecar.

Once a path is provided and localization achieved, control allows the racecar to travel along the provided trajectory.

2.2 System Architecture

Our implementation follows a modular design with three components, depicted in Fig. 1:

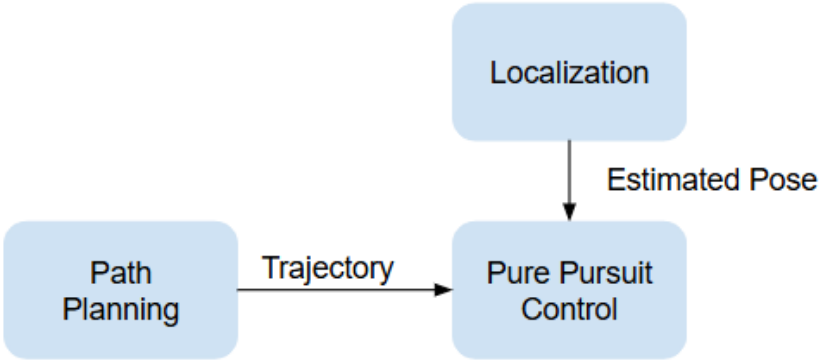


Fig. 1: **A depiction of how the path planning, pure pursuit control, and localization modules interact to enable autonomous navigation.** The path planning module provides pure pursuit a trajectory to follow and location the robot's estimated pose.

1. **Path Planning Module**: Finds a path if one exists, avoiding collisions between initial and end positions.
2. **Localization Module**: Determines the racecar's approximate pose.
3. **Pure Pursuit Module**: Publishes commands, allowing the racecar to follow a given trajectory.

2.3 Path Planning Module

The path planning module includes:

1. Pre-processing on the map (Initialization).
2. Finding a path between start and end positions if one exists (Path Finding).
3. Processing the resulting path (Path Smoothening).

2.3.1 Initialization

The racecar's starting and ending positions are provided. car_buffer is defined, representing the tolerance, in meters, around walls and obstacles for the generated path.

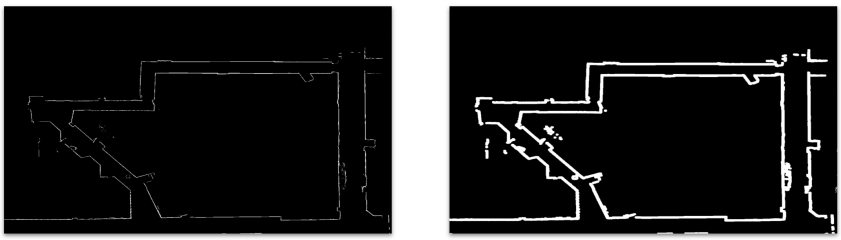


Fig. 2: **A comparison between features on the provided map (left) and dilated map (right).** The dilated map increases the size of walls and other obstacles in the map to provide additional buffer space when creating a path for the racecar to traverse.

Our algorithms model the racecar as a point. To prevent collisions with walls or obstacles, the map is dilated (Fig. 2) using a dilation function and an ellipse kernel where the ellipse' radii is given by

$$\lceil \text{car\_buffer} \cdot \text{resolution} \rceil \tag{1}$$

preventing the center of the racecar from being within car_buffer/2 of walls and obstacles when traversing a path. car_buffer was set to 1.0 m.

2.3.2 Path Finding

Path planning algorithms may be search-based or sampling-based algorithms. Search-based algorithms use discretized grids to provide the optimal path, but their performance depends on the granularity of the discretization which can limit scalability in complex environments. In contrast, sampling-based algorithms do not guarantee the optimal path or even the existence of a path. However, they are scalable to complex maps if there is an efficient sampling process. We expected that sampling-based algorithms would work well for our goal, as the optimality of the path is not as important.

Path-planning algorithms have key properties. An algorithm is asymptotically optimal if it converges to the optimal solution as the number of iterations/-computational resources increases to infinity. A single-query planner computes a path for a single start and goal position pair, while a multi-query planner builds up a representation that can be reused for multiple queries. Probabilistic Roadmaps is an example of a multi-query algorithm, but was not used for this lab, as we only need to travel between one start and end point. We rely on dilations and pure pursuit control to consider the dynamics and dimensions of the racecar and prevent collisions with the wall.

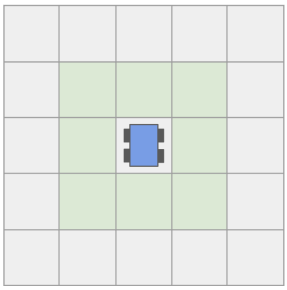


Fig. 3: **The neighbors that BFS and A* explore in each iteration of the algorithm.** The grid depicts the discretization of the map where green squares are considered neighbors of the car's current location.

The path planning module implements three path finding algorithms, some search-based and others sampling-based:

1. **BFS (search-based)**: BFS searches for a shortest path between the initial and end positions. The algorithm checks the racecar's eight neighbors (Fig. 3). Neighbors that do not cause collisions when traveled to are considered candidate points along the path towards the goal. BFS is single-query and will return an optimal path.

2. **A* (search-based)**: A* searches the same neighbors when looking for a path from source to destination. The path to neighboring nodes is additionally weighted with the heuristic function being the Manhattan distance. This algorithm is asymptotically optimal and single-query.

3. **RRT* (sampling-based)**: RRT* builds a tree by randomly sampling the free space and finding the optimal path towards the goal. RRT* is asymptotically optimal and single-query.

Later testing revealed that A* performed best (Section 3.2) and was thus chosen as the path finding algorithm for our implementation.

2.3.3 Path Smoothening

Paths found are smoothed to minimize jumps and oscillations. Two functions are run to achieve this using parameter max_attempts, defined to be 10. The first removes x_i if the path from x_{i-1} to x_{i+1} is collision-free. The second repeats this process over at most max_attempts consecutive points to find shortcuts:

```
Return if path length is less than 3;
Initialize a new path with the start of the original path;
Set i = 1;
while i < original path length - 1 do
    if path from last point in the new path to point i+1 in the original
     path is collision free then
        Skip the current point by incrementing i;
    end
    else
        Add to the new path the current point in the original path i;
        Increment i;
    end
end
Add the last point of the original path to the new path;
Return the new path;
```

Algorithm 1: Remove Redundant Points Algorithm

```
Return if path length is less than 3;
Initialize a new path with the start of the original path;
Set current_index = 0;
while current_index < original path length - 1 do
    Set best_index = current_index + 1;
    Set attempts = 0;
    for i where i starts at original path length - 1 and decrements to
     current_index + 1 do
        if the path from the point at current_index to i is collision free
         then
            Set best_index to i;
        end
        Increment attempts;
        if attempts ≥ max_attempts then
            Break;
        end
    end
    Add to the new path the point in the original path at best_index;
    Set current_index to best_index;
end
Return the new path;
```

Algorithm 2: Find Shortcuts Algorithm

2.4 Localization Module

The algorithm follows a predict-update-resample cycle, where:

1. Particles are propagated according to the motion model (Prediction)
2. Particle weights are updated based on sensor measurements (Update)

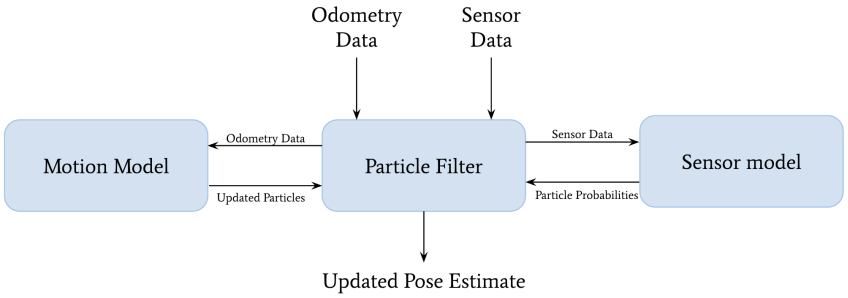


Fig. 4: **The system architecture.** Odometry and sensor data are provided to the Particle Filter module. Odometry data is then passed to the Motion Model which updates the set of particles accordingly. Sensor data is passed to the Sensor Model which returns particle probabilities based on the LiDAR scans. A new set of particles is aggregated and used to compute a pose estimate.

3. Particles are resampled according to their weights (Resampling)

The localization module follows a modular design, the motion model and sensor model composing the particle filter (Fig. 4):

- **Motion Model:** Predicts how particles move based on odometry data, incorporating deterministic updates and noise models to account for real-world uncertainties.
- **Sensor Model:** Evaluates the likelihood of each particle's pose given laser scan measurements using a precomputed probability lookup table for efficiency.
- **Particle Filter:** Initializes particles, integrates the motion and sensor models, manages particle weights and resampling, and estimates the robot's pose from the particle distribution.

2.4.1 Motion Model

The motion model implementation includes:

1. **Parameter Setup**: Declaration of ROS parameters for noise configuration.
2. **Noise Update**: Computation of noise based on current linear and angular velocities.
3. **Particle Update**: Transformation of odometry data from the robot's frame to the world frame, computation of pose in the next time step, and addition of noise.

The motion model updates particle poses based on odometry data. Given a particle at pose $(x_{t-1}, y_{t-1}, \theta_{t-1})$ in the world frame and an odometry reading $(\Delta x, \Delta y, \Delta\theta)$ in the robot's frame, the new pose (x_t, y_t, θ_t) is computed as:

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \end{bmatrix} + \begin{bmatrix} \cos(\theta_{t-1}) & -\sin(\theta_{t-1}) \\ \sin(\theta_{t-1}) & \cos(\theta_{t-1}) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} + \begin{bmatrix} \epsilon_x \\ \epsilon_y \end{bmatrix} \tag{2}$$

$$\theta_t = \theta_{t-1} + \Delta\theta + \epsilon_\theta \tag{3}$$

where ϵ_x, ϵ_y, and ϵ_θ represents noise added to model real-world uncertainties. Empirically, we found uncertainty increases with speed. These observations were incorporated into the following three noise models:

1. **Gaussian Noise**: Models sensor and motion uncertainty using a Gaussian distribution. There is a baseline standard deviation, linearly scaled by speed.
2. **Uniform Noise**: Models noise uniformly spread within a given range. The size of this range is linearly scaled by speed.
3. **Exponential Noise**: Models one-sided uncertainty (e.g., wheel slip) using a baseline scaling factor, also linearly scaled by speed.

Future data showed that Gaussian noise provided the most realistic behavior and was thus selected as the default model (Section 3.2). The noise parameters are tunable through ROS parameters, allowing for adaptation to different environments and hardware configurations.

By combining deterministic updates with stochastic noise, the motion model aims to capture the uncertainty in the robot's movement.

2.4.2 Sensor Model

The sensor model evaluates how well each particle's hypothesized pose matches observed laser scan data. Its implementation includes:

1. **Table Precomputation**: Creation of the lookup table during initialization.
2. **Map Integration**: Processing of the occupancy grid map for ray casting.
3. **Likelihood Evaluation**: Computation of particle likelihoods given observed laser scans.

We compute probabilities based on four components:

1. **Hit Model (p_{hit})**: A Gaussian probability centered at the expected distance, with weight $\alpha_{hit} = 0.74$.

2. **Short Model (p_{short})**: A linear probability that decreases with distance, modeling early reflections, with weight $\alpha_{short} = 0.07$.

3. **Max Model (p_{max})**: A spike probability at maximum sensor range, modeling failures to detect obstacles, with weight $\alpha_{max} = 0.07$.

4. **Random Model (p_{rand})**: A uniform probability over the sensor range, modeling random noise, with weight $\alpha_{rand} = 0.12$.

The total probability is a weighted sum of these models:

$$p_{hit}(z|x,m) = \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z-x)^2}{2\sigma^2}\right) \quad \text{if } 0 \le z \le z_{max} \text{ else } 0 \tag{4}$$

$$p_{short}(z|x,m) = \frac{2}{x}\left(1 - \frac{z}{x}\right) \quad \text{if } 0 \le z \le d \text{ and } d \ne 0 \text{ else } 0 \tag{5}$$

$$p_{max}(z|x,m) = \frac{1}{\epsilon} \quad \text{if } z_{max} - \epsilon \le z \le z_{max} \text{ else } 0 \tag{6}$$

$$p_{rand}(z|x,m) = \frac{1}{z_{max}} \quad \text{if } 0 \le z \le z_{max} \text{ else } 0 \tag{7}$$

$$p(z|x,m) = \alpha_{hit} \cdot p_{hit} + \alpha_{short} \cdot p_{short} + \alpha_{max} \cdot p_{max} + \alpha_{rand} \cdot p_{rand} \tag{8}$$

where z is the observed range, x is the particle pose, and m is the map.

To reduce runtime computation, we precompute a 201×201 lookup table indexed by discretized ranges (Fig. 5). For each possible ground truth distance d_j, we compute the likelihood for each possible measurement z_i (for i, j discrete). This approach significantly reduces computational overhead during runtime evaluation. Note that this additionally changes the equation defining $p_{max}(z|x,m)$, now given by:

$$p_{max}(z|x,m) = 1 \text{ if } z_i = z_{max} \text{ else } 0 \tag{9}$$

By vectorizing these operations with NumPy, we achieve the required real-time performance (> 20Hz).

2.4.3 Particle Filter

The particle filter integrates the motion and sensor models to perform MCL. It follows the predict-update-resample cycle:

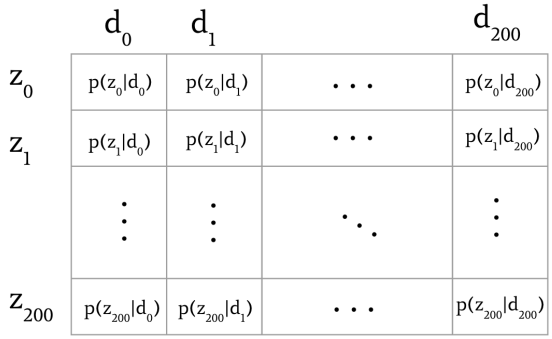


Fig. 5: **The precomputed table calculated in the sensor model.** The columns are possible ground truth distances and the rows possible measurements. Entry (i, j) in the table gives $p(z_i|d_j)$, the probability of measuring z_i given that the true distance is d_j.

```
Initialize particles around the initial pose;
while running do
    if odometry received then
        Update particles using motion model;
        Estimate robot pose from particles;
        Publish pose estimate and visualizations;
    end
    if laser scan received then
        Compute particle weights using sensor model;
        Normalize weights;
        Resample particles based on weights;
        Estimate robot pose from particles;
        Publish pose estimate and visualizations;
    end
end
```

Algorithm 3: Particle Filter Algorithm

2.4.4 Particle Initialization

Particles are initialized around the provided initial pose with added Gaussian noise, creating a diverse initial distribution.

2.4.5 Pose Estimation

Initially, we investigated k-means clustering to handle potential multimodal distributions. We ultimately decided against it due to its computational

overhead, which impacted real-time performance. Additionally, in our testing environment, the particle distribution was typically unimodal due to the particle initialization and small, feature-rich map that provided sufficient disambiguating information.

Instead, we compute the robot's pose estimate as the weighted average of particle poses. For position (x, y), we use arithmetic mean:

$$x_{est} = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad y_{est} = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{10}$$

For orientation θ, we use circular mean to properly handle angle wrapping:

$$\theta_{est} = \text{atan2}\left(\frac{1}{N}\sum_{i=1}^{N}\sin(\theta_i), \frac{1}{N}\sum_{i=1}^{N}\cos(\theta_i)\right) \tag{11}$$

This approach balances computational efficiency and accuracy for our unimodal particle distributions. In larger environments where multimodal distributions are more likely, the k-means approach could be revisited with optimizations to reduce its computational cost.

2.4.6 Resampling

We use weighted random selection to resample particles, favoring those with higher likelihood. This process focuses on promising hypotheses while maintaining diversity.

2.4.7 Concurrency

We employ thread locks to safely update particles across callbacks (odometry, laser, and pose initialization), preventing race conditions.

Our particle filter is parameterized with ROS parameters, allowing for easy tuning of the number of particles, resampling frequency, and other parameters affecting performance and accuracy.

2.5 Pure Pursuit Module

With consistent knowledge of the racecar's estimated pose, we use pure pursuit to control the racecar. Pure pursuit control provides a steering angle necessary for the racecar to stay on the precomputed path. A lookahead distance is used to constantly determine the point on the trajectory the racecar should steer towards. The linear speed of the racecar is independent of this control law and was set to a constant value during our implementation. Lookahead

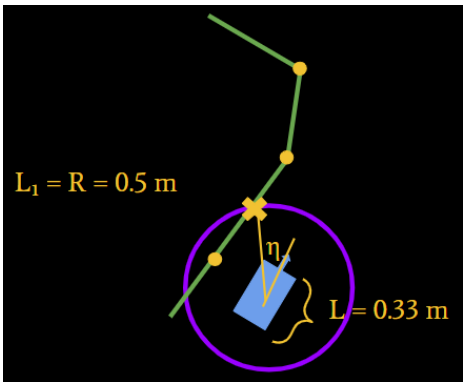


Fig. 6: **A visual representation of the pure pursuit implementation.** The yellow dots represent the closest point from the racecar to each segment in the path. The yellow cross shows the intersection between the lookahead distance and path segment, which is the lookahead point used to compute the steering angle.

distance was also set to a constant value of 0.5 meters, as we empirically found this provided the best performance and least oscillations. While dynamic lookahead distance was implemented in simulation, we were not able to determine whether it improved performance on the real racecar and found it difficult to debug. All experiments were therefore performed with the constant lookahead distance.

Given a set of points $[x, y]$ that form a path, segments can be defined as line segments formed by two consecutive points. Vector operations are used to extract useful information from these segments to compute the control angle. The pure pursuit module can be broken into 3 main steps:

1. Determine the closest point from the racecar to each segment.
2. Create a circle with the radius equal to the lookahead distance. Check points in front of the closest point to see where the circle intersects a segment. We call this intersection the lookahead point.
3. Compute the steering angle using the law $\delta = \arctan(\frac{2L \sin \eta}{L_1})$ where L is the racecar's wheelbase length (0.33 meters) and L_1 is the lookahead distance (0.5 meters).

3 Experimental Evaluation

3.1 Technical Procedures

We compared our path planning performance in simulation by measuring planning time, path length, total turning, average turning per meter, and number of way points.

Pure pursuit performance was evaluated by measuring the error between the car's trajectory and desired trajectory.

We evaluated our localization system by focusing on position and orientation error, convergence time, and computational speed. Simulated tests used a pre-recorded map, generating ground truth data by running the robot along predefined trajectories (Fig. 10). In real-life tests, we marked known positions on the floor and measured the difference between the estimated and actual poses.

3.2 Results

3.2.1 Path Planning Algorithms

We tested five start-goal pairs to include straight, turning, short, and long paths. For each algorithm, we measured planning time, path length, total turning, average turn per meter, and number of waypoints. Table 1 shows the average across trials.

Table 1: **Path Planning Metrics Across 5 Trials**

Metric	**A***	**BFS**	**RRT***
Success rate	5/5	5/5	5/5
Planning time (s)	0.1514	1.2088	0.202
Path length (m)	48.8756	48.7222	61.842
Total turning (rad)	1.8828	1.8578	3.676
Avg turning per meter (rad/m)	0.0364	0.0358	0.050
Number of waypoints	4.6	4.2	6.4

BFS and A* had comparable performance across all metrics, with A* having significantly shorter planning times. RRT* performed worst across all metrics. Hence, we chose A* as our path planning algorithm.

3.2.2 Pure Pursuit

Before combining path planning with pure pursuit, we evaluate pure pursuit's performance. To avoid offsets from inaccurate localization, the racecar

was set to track a path in simulation using ground-truth odometry. Error is defined as the shortest distance between the racecar and any point on the path. The error over time for a particular path is shown in Fig. 7. At speeds 1.0 m/s and 2.0 m/s, the error remained under 0.1 meters. At the beginning, error was higher due to the way in which the starting point is initialized. In order to satisfy the grid constraints of the planning algorithm, the clicked starting point is rounded to a position that is the closest multiple of 0.25 meters. The two spikes in each graph correspond to the two corners in the trajectory.

Additional error plots were taken on the real racecar. Here, localization was used to compute the distance between the estimated pose of the robot and the closest point on the path. As expected, the error is higher in the real world due to noise, wheel slippage, and other environmental factors. The results for two different paths and racecar speeds are shown in Fig. 8 and Fig. 9.

3.2.3 Motion Model Noise Model Comparison

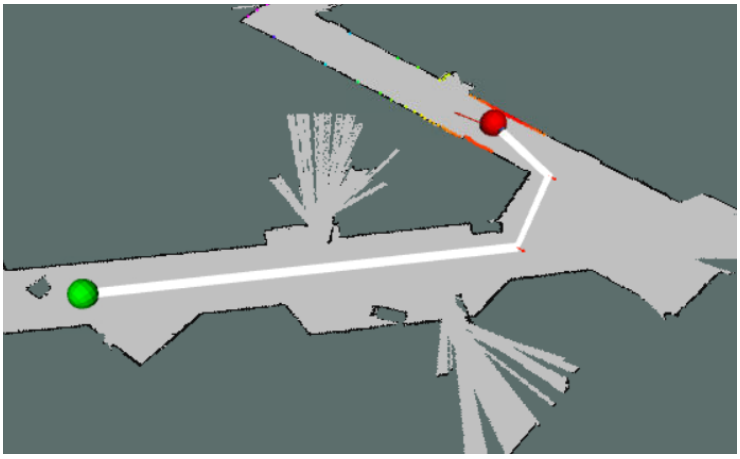
We ran our wall follower, generating a track and recording rosbags of the racecar's x-error, y-error, and θ-error, obtained by finding the difference between the estimated and true x and y positions. θ-error was obtained by converting quaternions to Euler angles and calculating the difference while accounting for wrap-around.

We varied the noise models, finding that the Gaussian noise model performed best with average position errors of 0.32 m in x, 0.02 m in y, and an orientation error of 0.31 radians. The uniform noise model resulted in larger errors: 0.35 m in x, 0.12 m in y, and 0.48 radians in orientation. Results for uniform and Gaussian models over time are shown in Figures 11 and 12 respectively. In both figures, there are certain instances of large spikes in θ-error despite correctly accounting for angle wrapping. Though uncertain, we suspect the spikes may be due to invalid estimated θ values. The exponential noise model performed worst, drifting before the car began to move, suggesting it is not suitable for modeling odometry noise.

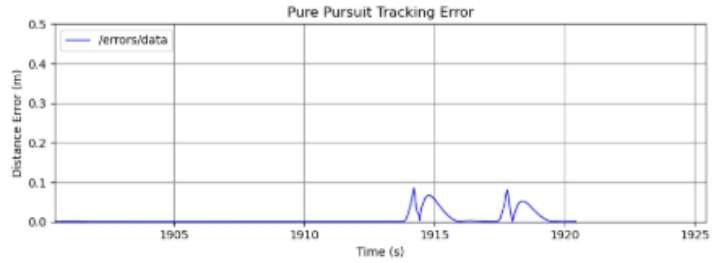
3.2.4 Particle Filter Convergence Analysis

We published an initial pose and measured the time for particles to be in tolerance around this position, where tolerance was defined as a circle centered at this position with radius 0.3 m, approximately the car's width.

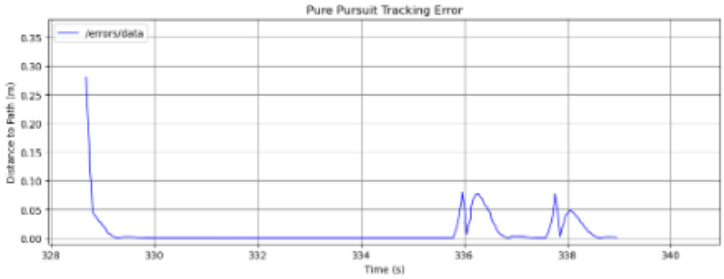
The car remained static as Gaussian noise was added. With baseline standard deviation 0.005 m, our particle filter converged within 0.5 seconds, with an average of 41 particles (out of 200) remaining out of tolerance, resulting in a 79.5% convergence rate. Increasing the standard deviation to 0.04 m increased convergence time to 1.25 seconds, with similar final performance (77.52% convergence rate) (Fig. 13, Fig. 14).



(a) The path used for testing pure pursuit algorithm.

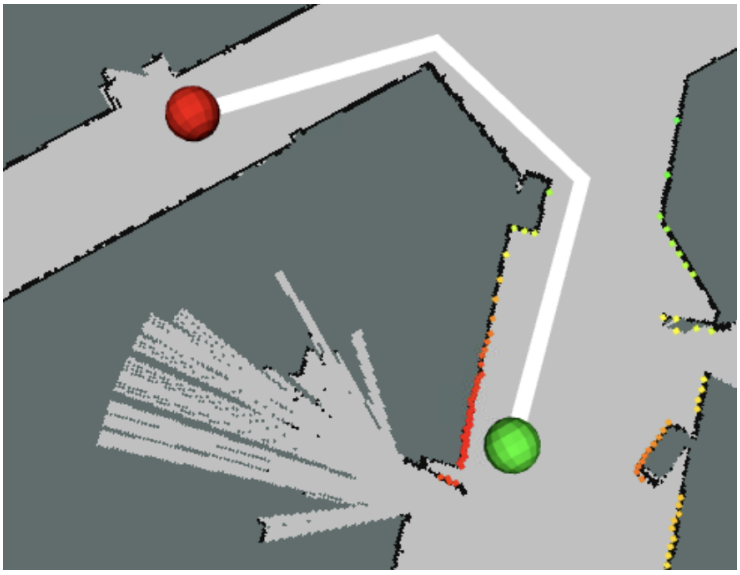


(b) Error over time at 1.0 m/s.

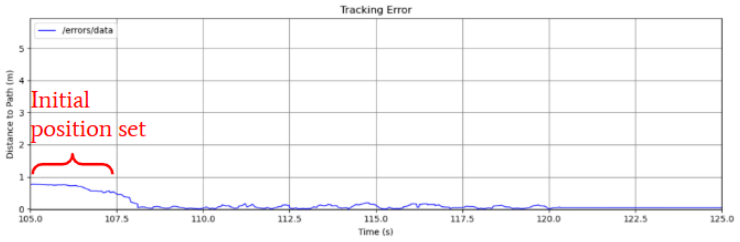


(c) Error over time at 2.0 m/s.

Fig. 7: **The distance from the path using pure pursuit control.** The error remains under 0.1 meters for most of the path at both speeds.

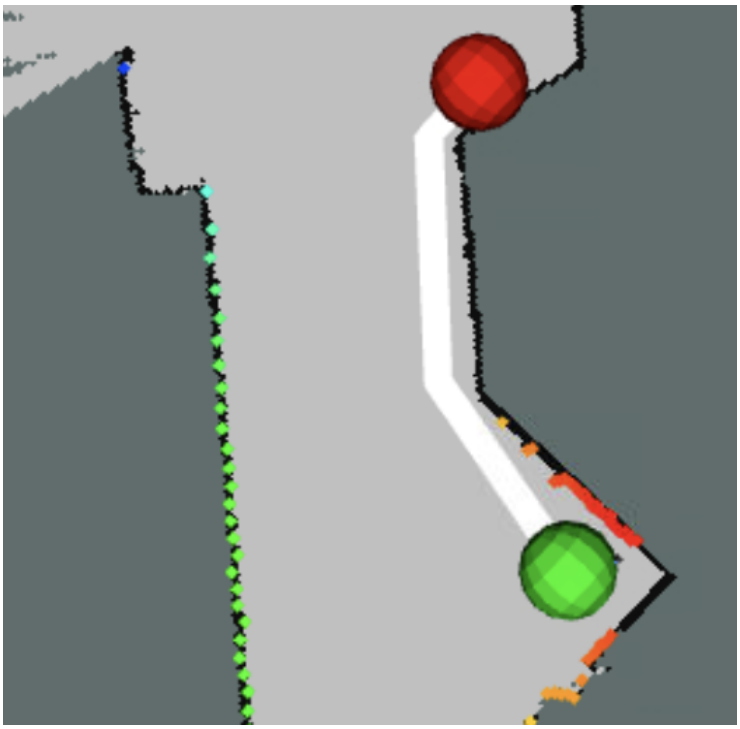


(a) The path used for testing integrated path planning and pure pursuit in the real world.

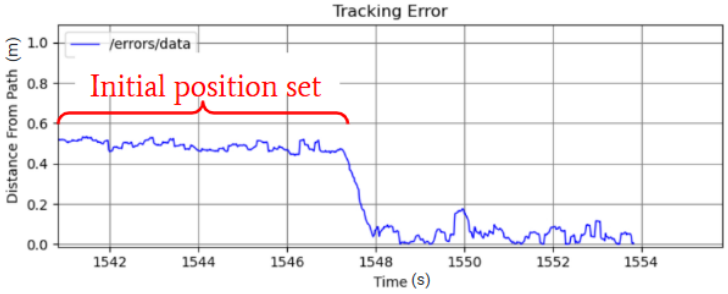


(b) Error over time at 1.5 m/s in the real world.

Fig. 8: **The distance from the path using pure pursuit control.** Once again, error is higher in the beginning. This may be due to how the starting position is initialized and due to localization error. Overtime, this localization error reduces as more LiDAR scans are gathered.



(a) The path used for testing integrated path planning and pure pursuit in the real world.



(b) Error over time at 1.0 m/s in the real world.

Fig. 9: **The distance from the path using pure pursuit control.** Once again, error is higher in the beginning. This may be due to how the starting position is initialized and due to localization error. Overtime, this localization error reduces as more LiDAR scans are gathered.

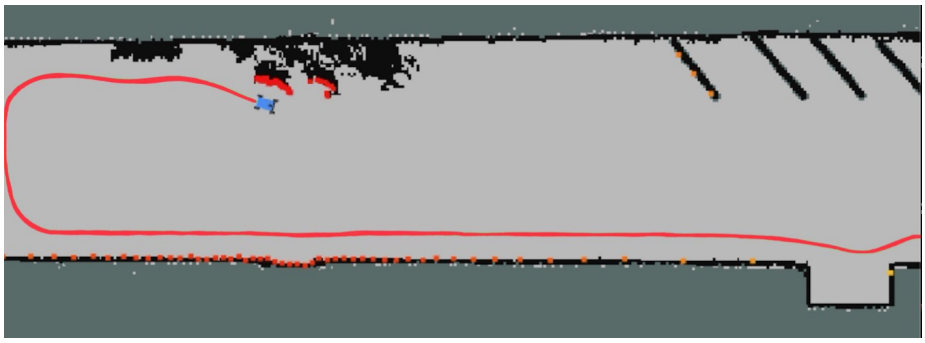


Fig. 10: **The curve depicts the path followed during our experimental evaluations.** Our track includes a straight path in addition to a turn and noisy objects.

3.2.5 Localization Real-World Performance

Analysis of real-world performance is difficult due to the lack of a ground truth measurement. In an attempt to establish a ground truth, a measuring tape was used to measure the position of the racecar relative to some feature in the environment. There is room for error in this method, as identifying the exact pixel of the feature on the map is very difficult/error-prone. We provide the results below (Table 2), but acknowledge that error analysis is best done in simulation.

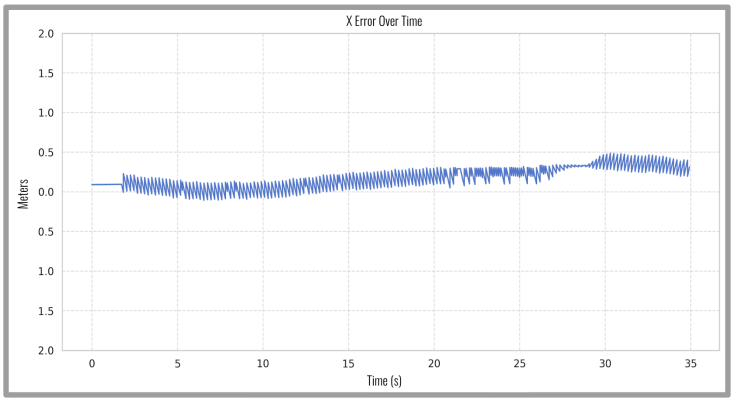
Table 2: **Localization Error Across Trials**

Trial	X Error (m)	Y Error (m)
1	0.07	0.14
2	0.09	0.11
3	0.16	0.12
Average	0.11	0.12

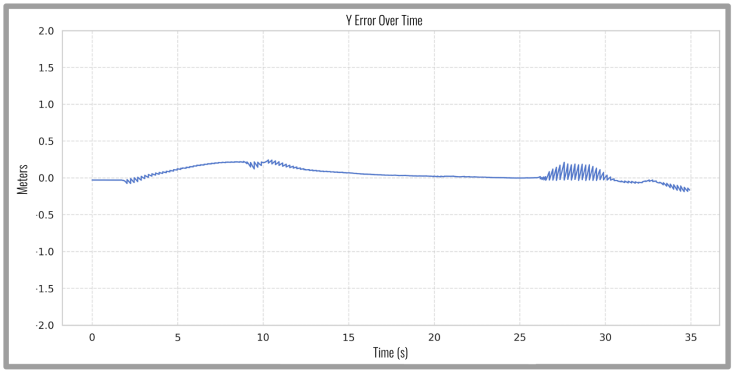
3.2.6 Computational Speed

Our implementation achieved a runtime frequency of 25 Hz on the racecar's onboard computer, exceeding the minimum requirement of 20 Hz.

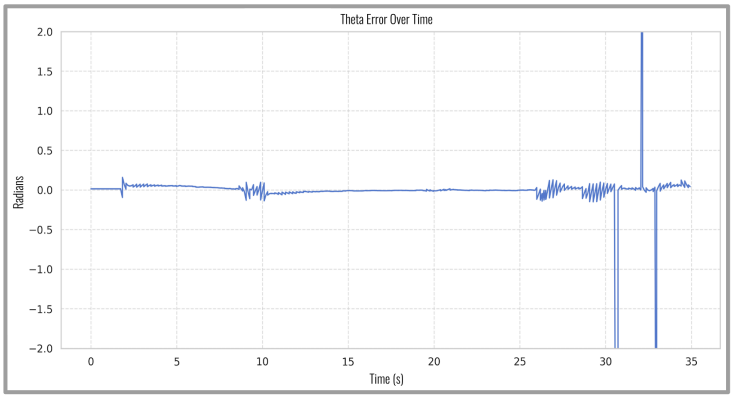
The results demonstrate that our MCL implementation localizes the robot with sufficient accuracy for autonomous navigation. The Gaussian noise model provided the best balance between exploration (maintaining particle diversity) and exploitation (converging to the correct pose). The system's real-time performance and sub-car-length accuracy make it suitable for deployment in controlled and dynamic environments.



(a)

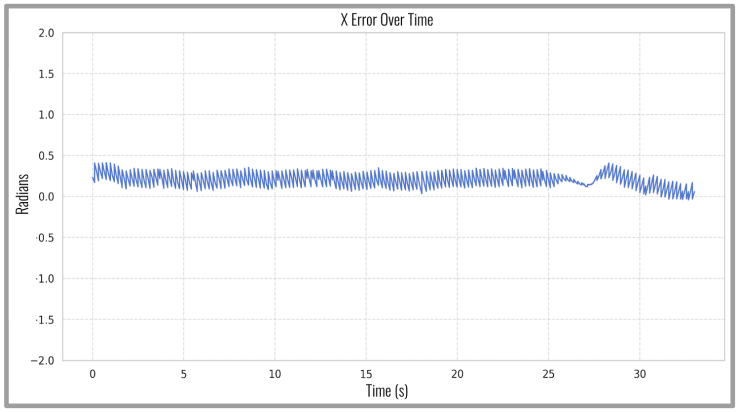


(b)

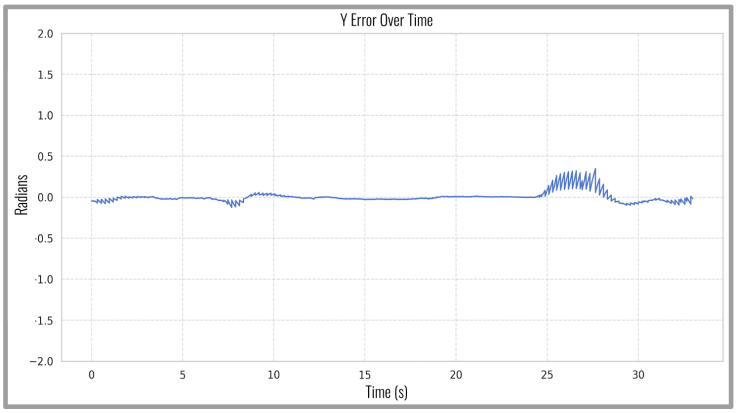


(c)

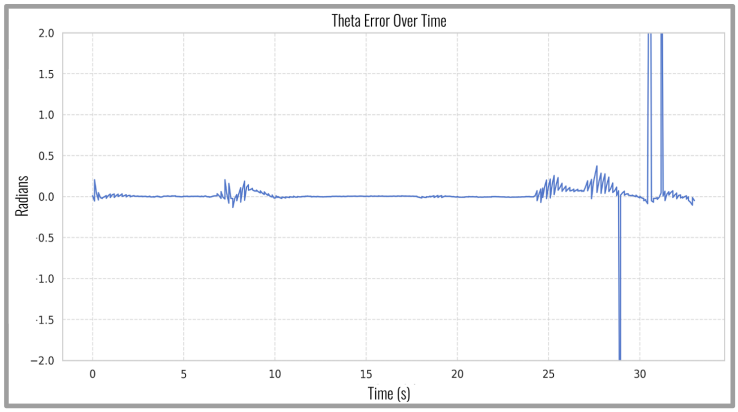
Fig. 11: **The graphs of x, y, and theta errors using noise drawn from the uniform distribution.** The x and y errors stay near zero, though x-error notably drifts over time. Theta error remains near zero radians.



(a)



(b)



(c)

Fig. 12: **The graphs of x, y, and θ-errors using noise drawn from a Gaussian distribution.** The x and y errors stay near zero, with x-error moving towards 0 even as time goes on. θ-error remains near zero radians.

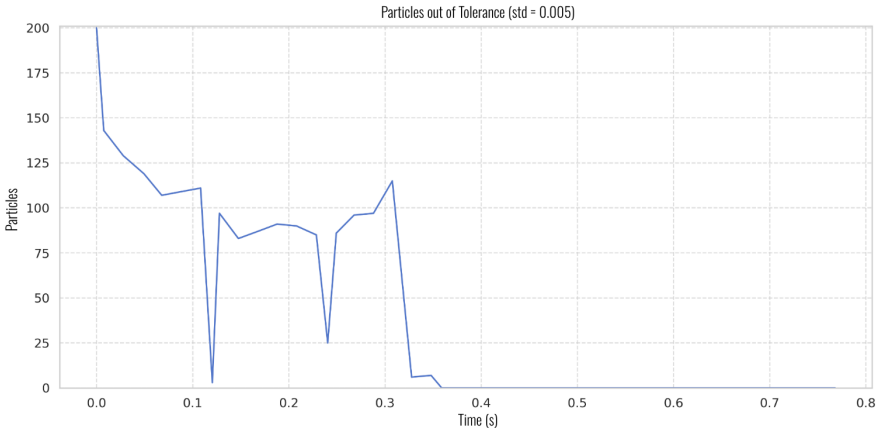


Fig. 13: **The number of particles within tolerance when the Gaussian noise model has a baseline standard deviation of 0.005 m.** The particles out of tolerance dropped from 200 to 0 within 0.5 seconds, demonstrating robust convergence.

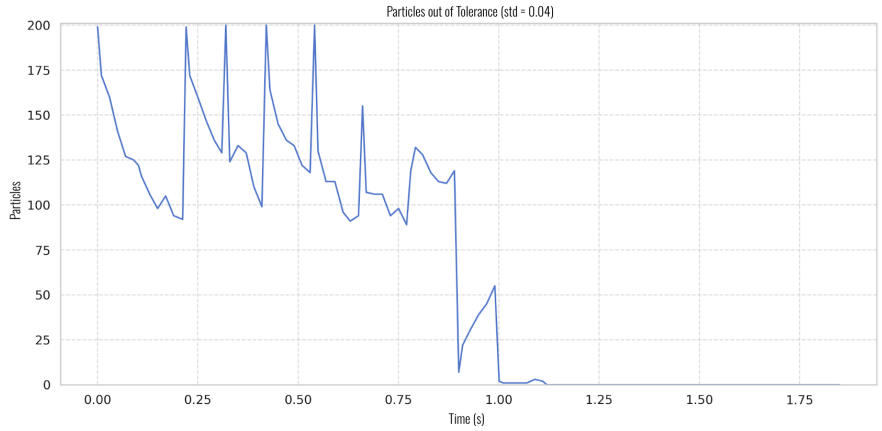


Fig. 14: **The number of particles within tolerance when the Gaussian noise model has a baseline standard deviation of 0.04 m**. The particles out of tolerance dropped from 200 to 0 within 1.25 seconds. Notably, there is more oscillation compared to the standard deviation 0.005 m case due to the increased noise. However, convergence remains resilient to increases in noise.

4 Conclusion

In this lab, we successfully implemented a path planning module to navigate through a mapped environment autonomously. Our solution incorporates Monte Carlo Localization, enabling the racecar to accurately determine its position within a known environment. Pure pursuit control is then used to steer the racecar along the planned path. To ensure safety, we use dilation of the map to avoid walls/corners and enable our safety controller to prevent collisions with unmapped obstacles.

Key achievements of our implementation include:

1. Implementation of A* algorithm for path planning.
2. Path simplification through removal of redundant points in consecutive and non-consecutive points.
3. Development of pure pursuit implementation that allows for accurate steering (less than 10 cm of error with ground truth odometry).
4. Development of a modular localization architecture with separate motion and sensor models, facilitating independent testing and tuning.
5. Implementation of a robust particle filter that efficiently samples the pose space and converges to accurate estimates.
6. Achievement of real-time localization performance (25 Hz) through algorithmic optimizations like precomputed lookup tables and vectorized operations.

While our current implementation meets the requirements of the lab, several improvements could enhance its performance:

1. Exploration of additional planning algorithms.
2. Incorporation of kinodynamic constraints in path planner.
3. Implementation of dynamic lookahead distance and speed for pure pursuit steering on the racecar.
4. Implementation of adaptive localization particle counts that increase during high uncertainty and decrease during stable operation.
5. Integration with simultaneous localization and mapping (SLAM) to update the map based on new observations.

As we move forward, this path planning and localization system will allow us to navigate through a mapped environment. This could be particularly useful when accomplishing tasks while ensuring safety of the racecar.

5 Lessons Learned

All members contributed equally to this report. Every section was written by and revised by multiple members to ensure an accurate and well-written report. Individual lessons learned and reflections are listed below.

5.1 Suchitha Channapatna

This lab helped me develop a better understanding of how different modules are integrated within a robotics system. I worked a lot more on the pure pursuit and testing side and hence had to integrate with both the path planning module and localization module. I appreciate how important it is to develop these modules to work seamlessly in isolation before putting them together. In terms of future work, I would like to try incorporating kinodynamic constraints into planning algorithms rather than relying on pure pursuit control capture the racecar's dynamics. Overall, I think this lab serves as a strong starting point for the final challenge, both in terms of which planning algorithms we use and how we test and integrate code.

5.2 Luis De Anda

This lab provided a good review on search algorithms and illustrated how efficient it is to separate our work. Knowing the basics of search algorithms coming into this, I knew that there would be ways to choose or create an effective search. I got to learn that what neighbors you check first can greatly impact how your paths form and that adding weights to spaces can still optimize our search. Specifically, on the second note, I got to learn about A* in more depth and got to implement it, seeing how using weights and a heuristic for priority can help optimize where we search and avoid searching in directions that would not produce results. Additionally, modularity in our tasks allowed us to be more efficient in advancing each of the modules while not getting in each other's way.

5.3 Amber Lien

I spent much of my time on this lab working on path planning algorithms, specifically search-based algorithms, which I have past experience with. I liked exploring the difference between the implementations created for robots and the implementations I used in the past that were more rooted in software. In lab 6, I learned the importance of real-world testing. Transitioning from path planning and execution in sim to the real world was difficult due to discrepancies in localization performance, hardware issues, and real-world constraints. For the final challenge, I aim to have things working in sim earlier to account for these issues. Additionally, I realized the importance of clear communication with my team is important, as miscommunications stemming from assumptions led to several bugs that could've easily been avoided.

5.4 Prince Patel

On the technical side, I learned how to implement a probabilistic path planning algorithm and ways to optimize it for path quality and speed of iteration. This experience will be valuable for the final challenge which requires real-time path planning. Regarding collaboration, I discovered the importance of attempting parts of the lab independently before coming together to work on integration. This allowed for independent exploration and development.