Lab 6 Report: Path Planning and Following With A*, PRM, Pure Pursuit, and MCL

Team 8

Monica Chan
Cristine Chen
Alex Franks
Kyle Fu
Asa Paparo

6.4200 Robotics: Science and Systems

April 24, 2025

1 Introduction

Author: Cristine Chen
Editor: Asa Paparo

This lab explores autonomous navigation, a core capability for mobile robots operating in complex environments. The objective was to navigate a racecar from a given start location to a designated goal location on a predefined map, using a combination of path planning, motion control, and localization. Building on the Monte Carlo Localization (MCL) algorithm developed in the previous lab, we now integrate it into a full navigation pipeline. In the real-world, autonomous navigation has wide-ranging applications: in hospitals, automated guided vehicles transport supplies and medications, and in semiconductor manufacturing, rail-guided robots operate in ultra-clean environments with precise motion requirements. These systems highlight the importance of reliable, efficient, and safe robot navigation. To achieve our objective, we organized the problem into three core components:

1. Path Planning: Generate a feasible trajectory from the start to the goal that the racecar can follow, while optimizing for factors such as planning time and staying far from the wall.
2. Pure Pursuit Control: Use the pure pursuit algorithm to track the planned path, converting it into motor commands that guide the racecar along the trajectory.

3. Localization: Leverage our previously implemented localization algorithm to estimate the car's pose in real-time, ensuring that control actions are based on accurate positional information.

Each component was first developed and tested in simulation, then deployed on the physical racecar. Through this lab, we gained practical insight into the interplay between planning, control, and state estimation in a real-world navigation system.

2 Technical Approach

2.1 Path Planning

2.1.1 Search-Based Planning

Author: Asa Paparo
Editor: Cristine Chen

Search-based planning offers a systematic approach for navigating discrete environments and is particularly effective in structured settings where both completeness and path quality are important. It guarantees an eventual convergence to an optimal path within the chosen search space if one exists.

In this lab, we used the A* algorithm to generate collision-free and efficient trajectories for an autonomous racecar operating within a 2D occupancy grid. We selected A* for its well-known balance between optimality and computational efficiency. Unlike uninformed methods such as Dijkstra's algorithm, A* incorporates a heuristic to guide the search, allowing it to prioritize more promising paths and reduce the number of unnecessary node expansions. The heuristic we used in our implementation is the Euclidean distance between the current position (i, j) and goal location $(i_{\text{goal}}, j_{\text{goal}})$ as described in (1).

$$h(i,j) = \sqrt{(i_{\text{goal}} - i)^2 + (j_{\text{goal}} - j)^2} \tag{1}$$

Because raw occupancy maps tend to be high-resolution and noisy, directly applying A* can be computationally expensive and overly sensitive to small inconsistencies in obstacle representation. To address this issue, we preprocess the map in several stages to improve efficiency and robustness. First, we perform downsampling using a max-pooling technique in (2),

$$D[i,j] = \max \left\{ O[u,v] \,\middle|\, \begin{matrix} u \in [ki, \min(k(i+1), H)] \\ v \in [kj, \min(k(j+1), W)] \end{matrix} \right\} \tag{2}$$

where D is the downsampled map and each element $D[i, j]$ corresponds to a block of size up to $k \times k$ in the original map O. Specifically, the block spans indices $u \in [ki, min(k(i + 1), H)]$ and $v \in [kj, min(k(j + 1), W)]$, where H and

W are the height and width of O, respectively. This block is referred to as the receptive field of $D[i, j]$, and is used to compute its value via a max-pooling operation over the corresponding region of O. This operation reduces the map's resolution while preserving obstacle information, significantly lowering the computational burden during planning. Next, we apply a Gaussian filter defined in (3) to blur the map at different scales,

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

where x and y represent the localized offsets from the center of the Gaussian kernel, which determine the weight assigned to each position. The central point receives the highest weight, with the weights gradually decreasing as the distance from the center increases. The amount of blurring is controlled by σ, where a larger σ corresponds to a wider influence. We applied two Gaussian filters with different standard deviations ($\sigma_1 = 3$ and $\sigma_2 = 9$) to create two blurred versions of the map (B_1 and B_2) that capture obstacle influences at multiple spatial scales. The small $\sigma_1 = 3$ Gaussian filter creates a localized inner blur which helps the path planner avoid tight spaces near walls. The large $\sigma_2 = 9$ Gaussian filter creates a broader outer blur with increased cost around obstacles for a larger area to create a buffer zone, helping the racecar maintain a safe distance from potential collisions.

We then combine the original, inner-blurred, and outer-blurred maps using a pixel-wise maximum operation shown in (4),

$$C[i, j] = \max(D[i, j], B_1[i, j], B_2[i, j]) \tag{4}$$

where $C[i, j]$ denotes the cost of occupying position (i, j) on the 2D occupancy grid. This cost is computed as the maximum of the original downsampled map $D[i, j]$, as well as the Gaussian-blurred maps $B_1[i, j]$ and $B_2[i, j]$. This cost map inflates the cost of cells near obstacles, encouraging the planner to favor safer, more open routes and avoid narrow passages.

The A* algorithm, outlined in Algorithm 1, begins by initializing the search from a specified start position on a 2D occupancy grid. The algorithm then systematically explores the map by expanding nodes in a best-first manner, considering only the four cardinal directions—up, down, left, and right—from each current cell. To determine which cells to expand first, A* evaluates each candidate using a cost function that combines the two previously discussed key components: an estimate of the remaining distance to the goal and the cost to reach the cell from the start. For each cell (i, j), the total estimated cost $f(i, j)$ is computed using (5),

$$f(i, j) = h(i, j) + g(i, j) \tag{5}$$

where $h(i, j)$ represents our Euclidean distance heuristic and $g(i, j)$ is the cumulative cost from the start to the current cell (i, j), calculated by summing

the values in the cost map $C[i, j]$ along the path taken.

As the algorithm progresses, it maintains a priority queue (a min-heap) to manage the frontier of cells to be explored, sorted by their total estimated cost $f(i, j)$. Additionally, a record of the lowest known cost to each cell is maintained to prevent unnecessary re-expansions, and parent pointers are stored to trace the path from the goal back to the start once it is found. When the goal is reached, the algorithm backtracks through these parent pointers to reconstruct the path, which is then transformed from grid indices to real-world coordinates using the map's resolution and origin.

Algorithm 1 A* Search with Obstacle-Aware Cost

```
procedure A_Star(start, goal, cost_map)
    open_set ← priority_queue()
    open_set.push(start, priority = 0)
    came_from ← {}
    g_score[start] ← 0
while open_set is not empty do
      current ← open_set.pop()
  if current = goal then
          return ReconstructPath(came_from, current)
  end if
  for all neighbor ∈ GetNeighbors(current) do
    if not InBounds(neighbor) or IsObstacle(neighbor) then
          continue
    end if
      tentative_g ← g_score[current] + cost_map[neighbor]
    if neighbor ∉ g_score or tentative_g < g_score[neighbor] then
          came_from[neighbor] ← current
          g_score[neighbor] ← tentative_g
          f_score ← h(neighbor, goal) + g_score[neighbor]
          open_set.push(neighbor, priority = f_score)
    end if
  end for
end while
return failure
```

2.1.2 Sampling-Based Planning

Author: Monica Chan
Editor: Alex Franks

To plan a trajectory, we used probablistic roadmaps (PRM) due to its runtime performance increase. At a high level, PRM samples N points in the obstacle-free space and performs A* on those points, as shown in Figure 1.

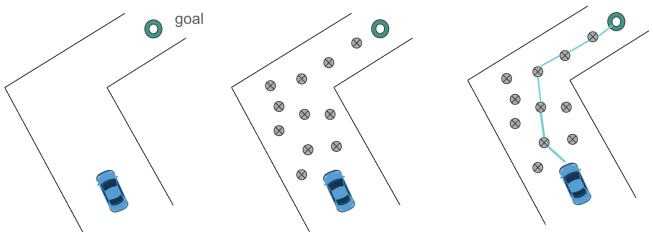

Fig. 1: An overview of the PRM algorithm. We break the process into 2 major steps: sampling points in the free space in the map and finding the path to the goal.

The first step in the algorithm is to create the graph that A* searches through, which involves finding a specified N number of points in the free space, and then adding valid edges between points. To sample free points, we just call a function N times that chooses a random point and checks whether the point is inside the free space, and it keeps trying until it finds a point in the free space. To add edges between points, we use Python's KDTree library to quickly query the nearest neighbors of the point. For each of those closest points, we draw an edge if there are no obstacles along the straight line between the points, as shown in Figure 2.

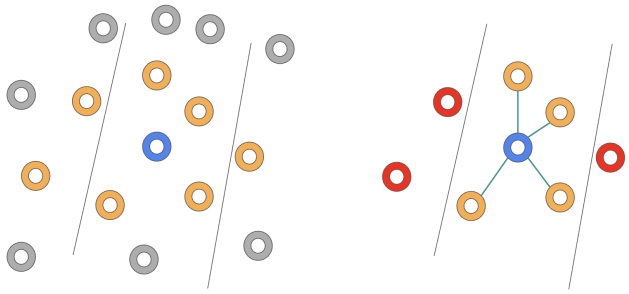

Fig. 2: To create the graph, we query x number of nearest neighbors to the point of interest as shown on the left in orange. Then, to draw edges, we consider whether a straight edge connecting the point of interest to the neighbor would intersect any obstacles like a wall. Edges are only drawn between the point of interest to edges are all in the free space.

The second step is to run A*, which is similar to as previously described, except for a modified cost function. Since A* will find the shortest path in our search space, but oftentimes the paths tend to be close to walls. To encourage the paths to be away from the walls, we modify the cost and priority function as outlined in (6) and (7),

$$\text{cost}(p,q) = ||p-q|| + \frac{w}{\texttt{DistToWall}(p,q)+\epsilon} \tag{6}$$

$$\text{priority}(q) = \text{cost}(p,q) + ||\text{end} - q|| \tag{7}$$

where p is the point of interest, q is the neighbor of p, and $w = 1, \epsilon = 0.25$ are empirically tuned coefficients. The second term of (6) is designed to penalize neighbors that are close to obstacles, where `DistToWall`(p,q) is calculated by drawing an edge between points p and q, and finding the minimum distance from any part of the wall to that edge.

Putting all these parts together yields our PRM path planning as outlined in Algorithm 2.

Algorithm 2 PRM Path Planning with Obstacle-Aware Cost

procedure PRM_Planner(*start*, *goal*, *map*, N, *kernel_size*)
  Dilate obstacles in `map` using kernel of size *kernel_size*
  Precompute `dist_to_obstacle_grid` using BFS
  $P \leftarrow$ SampleFreePoints(N, map)
  $P \leftarrow P \cup \{start, goal\}$
  $G =$ KDTree(P)
for all $p \in P$ **do**
    potential_neighbors $\leftarrow$ G.query(p, *num_neighbors*)
  for all $q \in$ *potential_neighbors* **do**
    if IsCollisionFree(p, q) **then**
      Add edge (p, q) to G
    end if
  end for
end for
  Use A* on G with cost:

$$\text{cost}(p,q) = \|p-q\| + \frac{w}{\texttt{DistToWall}(p,q)+\epsilon}$$

$$\text{priority}(q) = cost(p,q) + ||end - q||$$

  where w is the potential field weight and ϵ avoids division by zero.
  return shortest path from *start* to *goal*

2.2 Pure Pursuit

Author: Alex Franks
Editor: Kyle Fu

Given a trajectory from our path planning algorithm, we want to follow it as smoothly and closely as possible. We used pure pursuit for trajectory following,

as it is a robust and proven method. Pure pursuit finds a lookahead point on the trajectory in front of the racecar, and chooses a steering angle such that if the car kept steering with this specific angle, it would hit the lookahead point. This method allows pure pursuit to calculate a near-constant steering angle ahead of turns, making it more stable, as visualized in Figure 3.

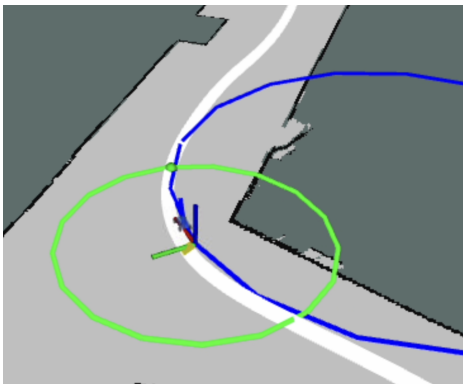

Fig. 3: Visualization of pure pursuit. The green circle around the racecar is the lookahead radius, and the green point on the trajectory marks the lookahead point. The blue circle indicates the path the racecar would take if it kept steering with the angle calculated by pure pursuit. This path is closely aligned with the trajectory path, ensuring stability.

To implement pure pursuit, we first get the minimum distance from the racecar to every segment of the trajectory by projecting the racecar's position onto a normalized $[0, 1]$ segment using (8),

$$t = \frac{\vec{v} \cdot \vec{v}}{\vec{w} \cdot \vec{v}} \tag{8}$$

where $\vec{v} = s_2 - s_1$ represents the direction vector of the segment and $\vec{w} = p - s_1$ which is the vector from the start of the segment to the car's position p. We clamp values of t between 0 and 1 in (9) to find the minimum distance from the racecar to a point on the line segment in (10). We pick the segment with the minimum distance d from the racecar, vectorizing with NumPy for efficiency.

$$t_{\text{clamped}} = \max(0, \min(1, t)) \tag{9}$$

$$d = \|p - (s_1 + t_{\text{clamped}}\vec{v})\| \tag{10}$$

Next, we find the lookahead point on the trajectory by iterating over the segments along the trajectory, including and in front of the one closest to the racecar, until a segment intersects with an L_1 radius circle centered on the racecar. L_1 is the lookahead radius. We only consider points in front of the racecar, and if there are two such intersections, we choose the one farther along the trajectory. The steering angle is then determined by (11),

$$\delta = \tan^{-1}\left(\frac{2L\sin\eta}{L_1}\right) \tag{11}$$

where L is the wheelbase length of the vehicle (distance between front and rear axles), and η is the angle between the vehicle's heading and the lookahead point. We measured $L = 0.33$ m and scaled L_1 with desired velocity v using (12).

$$L_1 = 1 + 0.5v \tag{12}$$

2.3 Localization

Author: Kyle Fu
Editor: Monica Chan

The goal of localization is to efficiently maintain a probabilistic estimate of the robot's pose (x, y, θ) over time by fusing odometry and LiDAR data. We started with our MCL implementation from Lab 5, but due to initial unresponsiveness on the racecar, we had to optimize it to run our autonomous navigation pipeline. Profiling helped us methodically locate the PyScanSimulator2D ray tracing library as the main source of slowdowns, as shown in Figure 4.

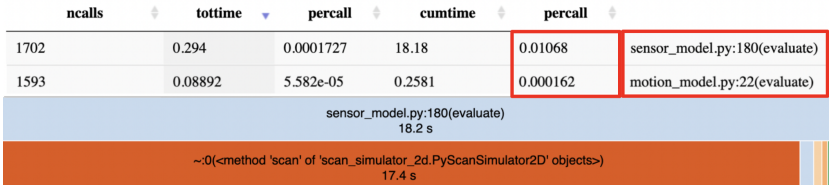

Fig. 4: Profiling results of our Lab 5 MCL code on a laptop using **200 particles** and 100 LiDAR beams per particle. Our sensor model took much more time per call than our motion model. Within the sensor model, the vast majority of time was spent in the PyScanSimulator2D library.

To alleviate the slowdowns caused by PyScanSimulator2D running on CPU, we switched to range_libc, an optimized ray tracer that runs on our racecar's GPU. At the expense of a slight startup delay, this optimization allowed us to run our navigation pipeline in real-time with an increased number of MCL particles, as evidenced by Figure 5.

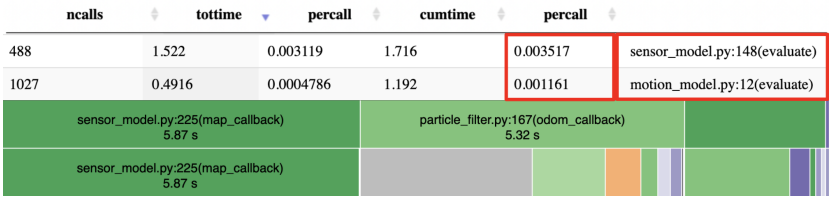

Fig. 5: Profiling results of our optimized MCL code on the racecar using **500 particles** and 100 LiDAR beams per particle. The time per call for both the sensor and motion models is under 0.004 s, and each is called ≤ 50 times per second, leaving CPU time for other modules. The startup delay (map_callback) is around 6 s.

3 Experimental Evaluation

3.1 Path Planning Evaluation

3.1.1 Search-Based Evaluation

Author: Asa Paparo
Editor: Monica Chan

To evaluate our grid-based A* search-based path planning algorithm, we recorded the time it took to generate a trajectory between a specific set of starting and destination points. Our main goal was to find an optimal set of parameters where A* paths could be generated without sacrificing our car's speed or the paths' feasibility.

Our algorithm was tested against 3 different downsampling kernel sizes. The results in Table 1 show that larger kernels cause decreased planning time because the planner must sample fewer points. However, kernels larger than 7 px could not successfully plan around the pillar in the Stata basement. Based on these results, we determined that sampling-based planning was necessary to achieve a reasonable planning time.

Table 1: Planning time vs. downsampling kernel size in grid-based A*

Downsampling Kernel Size (N)	Planning Time (s)
9	1.697
7	2.890
5	5.884

3.1.2 Sampling-Based Evaluation

Author: Monica Chan
Editor: Asa Paparo

To evaluate our PRM algorithm, we were interested in 2 main things: the amount of time it takes to plan a path, and the minimum distance to the wall. We prioritized minimum distance to the wall over path length for safety, as pure pursuit tended to cut corners during testing, and keeping the planned path away from walls was one way to prevent the racecar from cutting corners into a wall.

We were interested in comparing the sampling-based vs. search based runtime results. For each value of N, the number of points in free space sampled for PRM, we attained the runtimes in Table 2 by taking the average of 5 trials of the same path that we used in our A* evaluation. We can see that for the same path, the planning time for each of the samples is much lower than that of A*, even for a high point density of $N = 2000$.

Table 2: Planning time vs. number of samples used in PRM

Number of Samples (N)	Planning Time (s)
2000	0.857
1000	0.541
500	0.398

Qualitatively, PRM paths are found more often, but get less smooth with increasing N, as seen in Figure 6.

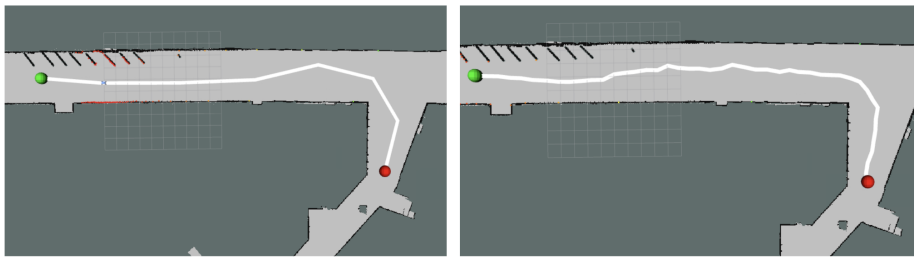

Fig. 6: PRM paths generated for different numbers of sampled points in free space. The path found for $N = 1000$ (right) is made up of many tiny, linear segments, making it less smooth than the $N = 500$ path (left), which is made up of a few long segments.

For the same paths, we evaluated the minimum distance to the wall and found it was 0.253 m for all values of N, likely due to the start and end points being the closest to the wall.

3.2 Pure Pursuit Evaluation

3.2.1 Pure Pursuit on Simulation

Author: Cristine Chen
Editor: Kyle Fu

We implemented the pure pursuit algorithm in a simulated environment to evaluate its trajectory tracking performance. We first generated a trajectory using the PRM algorithm, and then allowed the racecar to follow this path using the pure pursuit controller. The vehicle was commanded to move at a constant speed of 1 m/s using a fixed lookahead distance of 1 meter. To assess tracking accuracy, we measured the cross-tracking error, the distance between the vehicle's position and the closest point on the reference trajectory, over time. This metric served as a direct indicator of how well the vehicle stayed on course.

In the straight-line scenario in Figure 7, the cross tracking error remained consistently below 5 centimeters, indicating high-fidelity tracking under ideal con-

ditions. This result demonstrates that the pure pursuit controller can maintain precise alignment when the curvature of the trajectory is negligible.

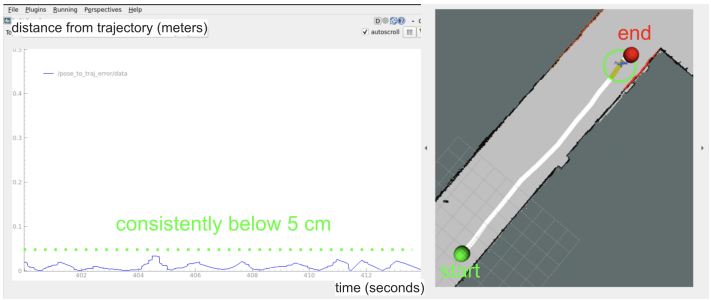

Fig. 7: The left plot shows the tracking error while following a straight-line path, and the right visualizes the path taken by the racecar during this tracking task.

We further evaluated the algorithm on trajectories involving a right turn in Figure 8 and a left turn in Figure 9. In both cases, the error remained low overall but peaked at approximately 15 centimeters during the midpoint of the turn, where curvature is highest.

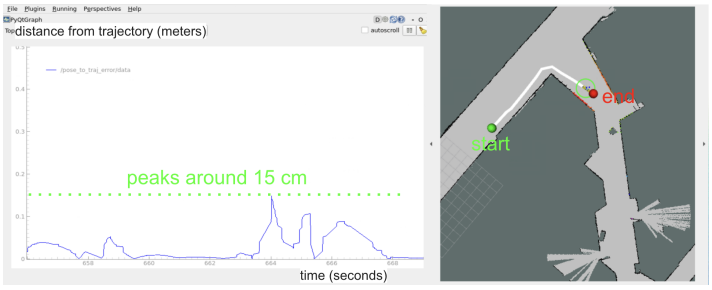

Fig. 8: The left plot shows the tracking error while following a right-turn path, and the right visualizes the path taken by the simulation racecar during this tracking task.

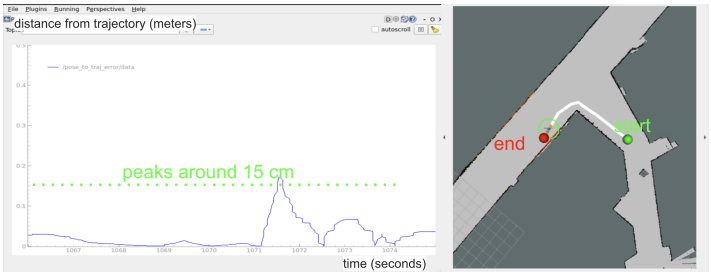

Fig. 9: The left plot shows the tracking error while following a left-turn path, and the right visualizes the path taken by the simulation racecar during this tracking task.

These results align with the expected performance of the pure pursuit controller given our choice of lookahead distance. The controller performs well on straight segments, where the target point lies directly ahead and requires minimal steering correction. However, in curved sections—particularly those with higher curvature—the fixed lookahead causes the controller to lag slightly behind the ideal path. This lag is a well-documented behavior of pure pursuit, where a longer lookahead improves stability but can reduce tracking precision in tight turns, while a shorter lookahead would increase responsiveness at the cost of more oscillatory behavior. Our findings reflect this tradeoff and illustrate how controller performance depends sensitively on the interaction between path geometry and lookahead tuning.

3.2.2 Pure Pursuit on Racecar

Author: Kyle Fu
Editor: Alex Franks

After validating our pure pursuit implementation in simulation, we deployed the controller on the physical racecar to evaluate real-world performance. We used the localization algorithm developed in the previous lab to estimate the vehicle's position and assess its ability to follow a predefined path. The racecar was driven at a constant speed of 0.75 m/s with a fixed lookahead distance of 1 meter. To quantify tracking performance, we measured the cross tracking error between the estimated vehicle position and the reference path at each timestep. As shown in Figure 10 and Figure 11, the tracking error remained consistently below 10 cm for both the right and left turn, respectively.

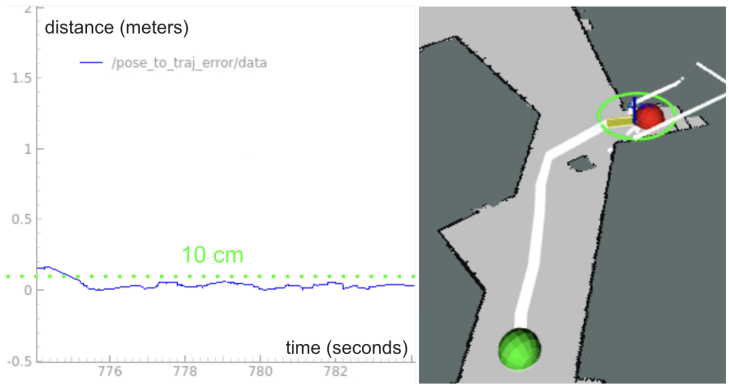

Fig. 10: The left plot shows the tracking error while following a right-turn path, and the right visualizes the path taken by the physical racecar during this tracking task.

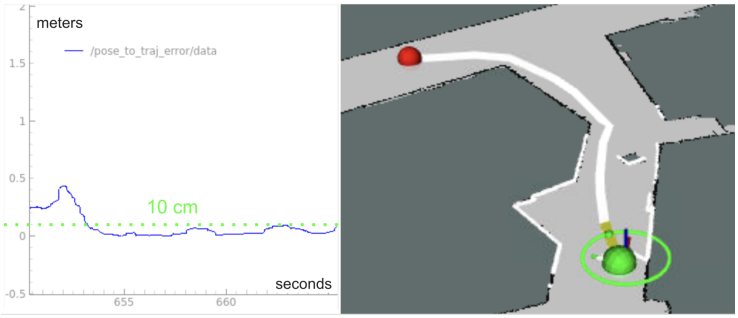

Fig. 11: The left plot shows the tracking error while following a left-turn path, and the right visualizes the path taken by the physical racecar during this tracking task.

While these results demonstrate effective path tracking under our experimental conditions, it is important to note that the accuracy of our cross tracking error measurements is limited by the quality of the localization system, which does not provide ground truth and may introduce some bias or noise into the estimated position.

4 Conclusion

Author: Alex Franks
Editor: Cristine Chen

In Lab 6, we successfully implemented a full navigation stack on the physical racecar, comprising a path planner (PRM), a path follower (pure pursuit), and a localization system based on Lab 5. Our system demonstrated the ability to navigate reliably from a start position to a goal position in both simulation and real-world environments.

We implemented and tested both a search-based path planner (A*) and a sampling-based path planner (PRM) to evaluate their performance in our navigation pipeline. PRM proved more effective in our context due to its significantly faster runtime, making it more suitable for real-time deployment on the racecar. Quantitative evaluation of the physical deployment showed low cross tracking tracking error—under 10 cm in both left and right turns—at a constant speed of 0.75 m/s with a fixed 1-meter lookahead. These results support the effectiveness of our approach under standard conditions.

However, we observed limitations in certain environments, such as the Stata Basement. In these cases, unexpected obstacles against the wall introduced errors into our LiDAR-based localization, causing premature turns due to incorrect position estimates. This issue, among others, highlights the need for more robust localization under complex indoor conditions. For future work, we plan to:

- Integrate adaptive lookahead in the pure pursuit controller to better handle sharp curves and higher speeds.
- Explore path planning algorithms that consider differential constraints, allowing the planner to generate more realistic and feasible trajectories.
- Create an updated map that incorporates fixed obstacles in the Stata Basement, improving the fidelity of localization and overall navigation robustness.

These enhancements aim to increase system performance, particularly in real-world environments that deviate from idealized simulation conditions.

5 Lessons Learned

5.1 Monica Chan

5.1.1 Technical

I learned how PRM and sampling based trajectory planning works and how to implement it. I also learned about KDTrees as an optimal way of querying nearest neighbors for graph creation.

5.1.2 CI

We struggled a bit with making our slides nice for the briefing because we started the briefing preparations a bit later than usual, so I learned both how to efficiently make nice looking slide decks and also the value of starting these things early so we can iterate.

5.2 Cristine Chen

5.2.1 Technical

I learned how to implement the pure pursuit algorithm for path tracking and successfully integrated both the path planning and path following components to work cohesively within a simulation environment. This process involved ensuring smooth communication between modules, validating the system's behavior under various scenarios, and fine-tuning parameters like the fixed lookahead distance to achieve stable and accurate navigation along the planned path.

5.2.2 CI

I learned that having a backup plan is always a smart strategy, especially when working on team projects where unexpected issues can arise. Additionally, having some meeting times as dedicated working sessions proved to be highly effective. It allowed team members to collaborate in real time, quickly ask questions, share progress, and offer help to one another, which improved overall productivity and team cohesion.

5.3 Alex Franks

5.3.1 Technical

I learned how to use NumPy to efficiently process and manipulate line segments as part of implementing the pure pursuit algorithm. This process involved leveraging NumPy's array operations to perform geometric calculations, such as determining lookahead points and calculating distances, which significantly improved the performance and clarity of the path tracking logic.

5.3.2 CI

I learned the value of communicating more openly and effectively with my teammates, especially when facing challenges or time constraints. By sharing my circumstances clearly, I was able to foster a stronger sense of mutual understanding and collaboration, which made it easier for my teammates to offer support and coordinate efforts when I needed help.

5.4 Kyle Fu

5.4.1 Technical

I gained some intuition for how to ensure things in simulation would transfer to the racecar, by intentionally introducing noise/slowed steering angle input changes into simulation (in short, trying to make the code fail in sim). I also practiced organizing Github branches to keep "production" and "development" code separate.

5.4.2 CI

I learned the importance of communicating code changes visually to the team, rather than through text, as people don't want to read through tons of text to understand what someone did, and it also helps for later briefings/reports. I'm also continuing to practice refining the report.

5.5 Asa Paparo

5.5.1 Technical

I practiced implementing the A* pathfinding algorithm as part of a motion planning system for autonomous navigation. In addition to setting up the core algorithm, I focused on fine-tuning the cost function by adjusting various weight parameters. This adjustment allowed the planner to more effectively evaluate potential paths and prioritize those that maintained a safe distance from obstacles, thereby enhancing both safety and efficiency in path selection.

5.5.2 CI

I realized the importance of starting tasks early, particularly when working on the briefings. Beginning the preparation process ahead of time not only allows for a more thoughtful and thorough approach but also significantly reduces the pressure of last-minute edits and corrections. By giving myself extra time, I will able to catch mistakes I might have otherwise missed, incorporate feedback more effectively, and ultimately deliver a more polished and professional final product.