Lab 6 Report: A* and RRT Path Planning with Pure Pursuit

Team 12

Adan Abu Naaj
Amina Luvsanchultem
Andy Yu
Arthur Hu
Dylan Gaillard

6.4200: Robotics Science and Systems

April 24, 2025

1 Introduction

Author(s): Adan Abu Naaj

In this lab, we built upon our previously implemented Monte Carlo Localization algorithm from Lab 5, which allowed the robot to estimate its position and orientation using odometry and LiDAR scan data. This system provided a robust foundation for autonomous navigation by employing a probabilistic particle filter. The filter combined a **motion model**, which updates the robot's state using noisy odometry data, with a **sensor model**, which weights each particle by comparing predicted and actual LiDAR scans to maintain an accurate estimate of the robot's location on a known map.

In Lab 6, we extended our navigation system to trajectory planning and trajectory following, enabling our robot to drive from a specified start to goal pose. This capability advanced our navigation approach from purely reactive movement towards goal-driven autonomous behavior.

To achieve this functionality, we implemented two essential modules:

1. **Path Planning:** We developed a trajectory planner that computes collision-free paths through a known occupancy grid map (Stata basement), utilizing either a search-based (A*) or a sampling-based (RRT) planning algorithm. This involved discretizing or sampling our configuration space to efficiently navigate around obstacles and generate feasible paths.
2. **Pure Pursuit Controller:** We implemented a trajectory-following algorithm using the Pure Pursuit method, allowing the robot to follow predefined or dynamically planned paths by steering toward a lookahead point. By combining localization data from our previously implemented particle filter with real-time path following, our robot could effectively execute planned trajectories.

Finally, we integrated these components — localization, planning, and trajectory tracking — into a unified system tested both in simulation and on the physical racecar. The integrated solution marked a significant step toward complete autonomous navigation.

2 Technical Approach

Author(s): Adan Abu Naaj, Amina Luvsanchultem, Andy Yu, Dylan Gaillard, Arthur Hu

2.1 Problem Statement

The goal of this lab was to enable the robot to plan and follow collision-free trajectories between a given start and end position. Building on our Monte Carlo Localization system from Lab 5 for accurate pose estimation, we implemented A* and RRT for path planning and applied a Pure Pursuit controller for trajectory execution.

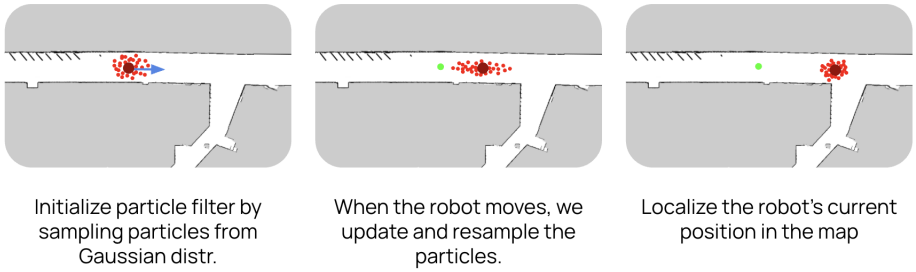

Figure 1: We use MCL to track our car's position and orientation on the map based on odometry and LiDAR data. The localization process is displayed in the figure above, where we propagate, update, and resample particles based on the odometry and LiDAR data.

2.2 Initial Setup

Our initial setup for this lab consisted of a racecar robot equipped with:

1. Onboard LiDAR sensor
2. 4-wheel-drive and steering motors (an NiMH battery)
3. An NVIDIA Jetson Nanodeveloper kit (and battery) running ROS2 in a docker container
4. A router equipped with Ethernet for communicating with the Jetson
5. A Logitech joystick controller for teleoperation

2.3 Monte Carlo Localization (MCL)

2.3.1 Particle Initialization

Using a Gaussian distribution centered at the current pose ($\sigma_{x,y} = 0.5, \sigma_\theta = 0.1$), we initialize multiple particles (possible robot positions) around an initial pose estimate with equal weight. This starts the localization loop where the sensor and motion models continuously update the particles, their weights, and the robot's estimated pose.

2.3.2 Sensor Model

We use a precomputed probability table to find the probability of a particle's simulated scan given current LiDAR observations. Each entry of the table is the likelihood of a LiDAR measurement given a discretized ground truth distance, which ranges between 0 and 200 meters as given. The likelihoods are determined using four different distributions:

We determined our max distance, z_{max}, to be 200. $z_k^{(i)}$ is the measured range, and the ground truth range is d.

$p_{\text{hit}}(z_k^{(i)} \mid d)$ accounts for when the measured value is close to the ground truth value. In our implementation, $\sigma = 8$.

$$p_{\text{hit}}(z_k^{(i)} \mid d) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^{(i)}-d)^2}{2\sigma^2}\right) & \text{if } 0 \leq z_k^{(i)} \leq z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$p_{\text{short}}(z_k^{(i)})$ accounts for when the measured values are shorter than expected.

$$p_{\text{short}}(z_k^{(i)} \mid d) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} & \text{if } 0 \leq z_k^{(i)} \leq z_{\max} \text{and } d \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$p_{\max}(z_k^{(i)} \mid d)$ accounts for when the sensor outputs the maximum distance reading, which usually means that there are no obstacles within the range of the LiDAR. In our implementation, $\epsilon = 0.01$.

$$p_{\max}(z_k^{(i)} \mid d) = \begin{cases} \frac{1}{\epsilon} & \text{if } z_{max} - \epsilon \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$p_{\text{rand}}(z_k^{(i)} \mid d)$ accounts for randomness in the readings.

$$p_{\text{rand}}(z_k^{(i)} \mid d) = \begin{cases} \frac{1}{z_{max}} & \text{if } 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

We combine these four distributions using a weighted average to calculate the probability where α_{hit}, α_{short}, α_{max}, α_{rand} are the associated weights. In our implementation, the alpha values were 0.74, 0.07, 0.07, and 0.12, respectively.

$$\begin{aligned} p(z_k^{(i)} \mid d) = \alpha_{\text{hit}} \cdot p_{\text{hit}}(z_k^{(i)} \mid d) \\ +\alpha_{\text{short}} \cdot p_{\text{short}}(z_k^{(i)} \mid d) \\ +\alpha_{\text{max}} \cdot p_{\text{max}}(z_k^{(i)} \mid d) \\ +\alpha_{\text{rand}} \cdot p_{\text{rand}}(z_k^{(i)} \mid d) \end{aligned} \tag{5}$$

We then take the product of these probabilities across all sensor scans or beams to calculate each particle's weight for the table. This process is described further in the pseudocode below:

Algorithm 1 evaluate(particles, observation)

```
Initialize probability array P
Obsv ← ground truth scan data
scale and clip Obsv to contain distances up to 200
for all particles do
    Scan ← simulated LiDAR scans from the particle
    scale and clip Scan to contain distances up to 200
    P' ← array of probabilities P(Scan|Obsv)
    append ∏ P' to probability array P
end for
return P
```

2.3.3 Resampling

Particles are sampled with replacement based on their normalized weight. The result is a new particle set that concentrates around high-probability regions from the previous time step.

While this algorithm successfully localized the robot on a map of the Stata basement in simulation, real-world testing revealed additional challenges. Particles would sometimes pass through or remain inside walls, and the robot struggled to localize in long, featureless hallways.

To address the wall issue, we added a check; if a particle is found to be inside a wall or an occupied space (determined by the occupant grid of the map), we set its weight to zero.

To handle hallway localization issues, we introduced a pre-resampling check to maintain particle diversity in ambiguous environments. We compute the effective sample size using the equation:

$$n_{eff} = \frac{1}{\sum p_i^2} \tag{6}$$

where p_i is the weight of particle i. In an ideal case with uniform weights, n_{eff} equals the total number of particles. As particles diverge and few maintain high probabilities, n_{eff} drops. If n_{eff} falls below 60% of the total particle count, we trigger resampling to maintain a tight distribution around our predicted location. Otherwise, we retain the existing particles and weights, assuming the distribution is adequately spread.

2.3.4 Motion Model

With added Gaussian noise to simulate real-world uncertainty, our motion model updates particle positions using robot linear and angular velocity odometry data in the robot-centric frame. We found standard deviations of 0.1, 0.1, and 0.05 for X, Y, and theta noise, respectively, worked well. For each particle, the noisy odometry data is then converted into a transformation matrix. This allows us to compute new particle poses in the map frame. Finally, we convert these updated matrices into $\Delta x = (\delta x, \delta y, \delta\theta)$ format.

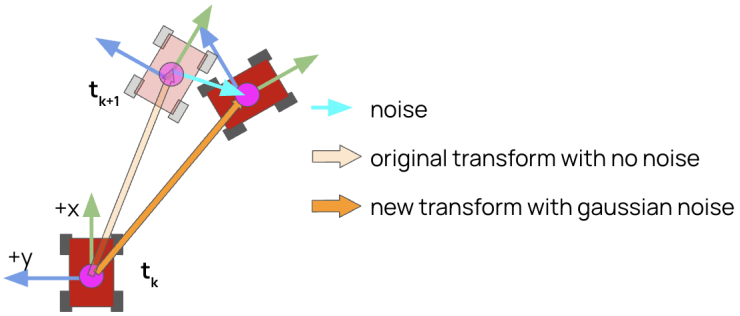

Figure 2: The motion model updates the particles' positions and orientation by applying the odometry transform and integrating uncertainty using noisy odometry messages.

2.3.5 Localization Loop and Pose Estimation

Our localization loop proceeds as follows (order may vary based on timing):

1. An initial position is provided — initializing the particles and their weights.
2. The motion model updates particle positions based on odometry data from the racecar.
3. The sensor model adjusts particle weights based on the latest LiDAR scan.
4. A predicted pose of the racecar is computed from the weighted average of the particle set. This process is detailed further below.

After each call to the sensor or motion model, we publish the estimated pose: a weighted average of the particles' X and Y positions, and a weighted circular mean for θ, representing the robot's position and orientation on the map.

2.4 A*

2.4.1 Map Pre-processing

A* is a search-based algorithm that operates over a discretized grid space. To create this representation, we processed the provided

`OccupancyGrid` of the Stata basement, which discretizes the environment into cells labeled as occupied, free, or unknown. We treated unknown cells as occupied.

To account for the robot's geometry and avoid wall collisions, particularly from paths generated too close to obstacles, we expanded the occupied regions by applying a disk dilation with a 10-pixel radius.

We preserved the original map resolution, which increases the runtime of A*. The performance was still deemed acceptable, as planning is only executed once per desired path. A detailed runtime analysis of A* is included in the Experimental Evaluation section.

2.4.2 Algorithm

A* is an informed search algorithm that prioritizes exploration of nodes with the lowest estimated total cost, combining the cost to reach the node and a heuristic estimate to the goal. It continues this process until the goal is reached or all possible nodes are explored without finding a valid path.

In our implementation, we use Euclidean distance as the heuristic. Each node considers its 8 immediate neighbors. Nodes that are occupied or have already been visited are skipped.

The cost of reaching a neighboring node is calculated by adding the movement cost (1 for adjacent, $\sqrt{2}$ for diagonal) to the current node's movement cost, along with the neighbor's Euclidean distance to the goal.

The cost of a node is defined as:

$$f(n) = g(n) + h(n) \tag{7}$$

$g(n)$ is the cost from the current node to node n, and $h(n)$ is node n's heuristic estimate to the goal:

$$h(n) = \sqrt{(x_n - x_{\text{goal}})^2 + (y_n - y_{\text{goal}})^2} \tag{8}$$

If the goal is reached, the path is reconstructed by tracing back through the stored parent of each node along the solution path. This sequence of grid coordinates is then converted to world coordinates using the known transformation between the map and world frames, along with the map's resolution. The resulting path is published as a `PoseArray` for later visualization. If no valid path is found, nothing is returned.

2.5 RRT

We also planned using the Rapidly-Exploring Random Tree (RRT) algorithm. RRT leverages a growing tree structure using random samples in the car's configuration space to reach a goal point. Specifically, the algorithm follows the pseudocode below.

Algorithm 2 RRT($x_{\texttt{start}}$, $x_{\texttt{goal}}$, map, `goal_dist`)

$V \leftarrow V \cup \{x_{\texttt{start}}\}$
for all i=1,...,N **do**
  $z_{\texttt{rand}} \leftarrow \texttt{Sample}(X)$
  $z_{\texttt{nearest}} \leftarrow \texttt{Nearest}(V, z_{\texttt{rand}})$
  $x_{\texttt{new}} \leftarrow \texttt{Steer}(z_{\texttt{nearest}}, z_{\texttt{rand}})$
  if $\texttt{CollisionFree}(x_{\texttt{new}})$ **then**
    $z_{\texttt{new}} \leftarrow x_{\texttt{new}}(T)$
    $V \leftarrow V \cup \{z_{\texttt{new}}\}$
    $E \leftarrow E \cup \{(z_{\texttt{nearest}}, z_{\texttt{new}})\}$
  end if
  if $||z_{\texttt{new}} - x_{\texttt{goal}}|| \leq$ `goal_dist` **then**
    return $\texttt{GetPath}(z_{\texttt{new}})$
  end if
end for

We first specify a maximum number of iterations to plan over, N. Further, we initialize the tree over which RRT plans as with a node at the provided `start` pose.

As part of our `Steer` function, using the provided `OccupancyGrid` (map), we use `numpy`'s built-in `random.uniform` to sample a random (x, y) pair within the xy-bounds of our map. If this new sample is itself in collision, as specified by the map, we resample a new (x, y).

Our `Nearest` function returns the parent node (pose) on the tree nearest to the random sample in Euclidean distance. Then, our

RRT algorithm uses a Pure Pursuit-inspired approach to find the nearest navigable point `wp` to the sample given the parent pose. Using a prespecified lookahead l_d distance, we find the point a distance l_d from the parent along the line to the sampled point. We then find the required steering angle to reach this point by inverting the Pure Pursuit relation:

$$\texttt{steer\_angle} = \arctan\left(\frac{2L\sin\theta_{\texttt{wp}}}{l_d}\right) \tag{9}$$

where $\theta_{\texttt{wp}}$ is the angle to the sampled point `wp` relative to the parent pose heading, and L is the length of the car's wheelbase. If `steer_angle` is outside the allowed steering angle range ($\pm\pi/4$), we clip the value. Then, we recompute the new point using the (possibly clipped) value of `steer_angle` and the Pure Pursuit formula:

$$\theta_{\texttt{wp, new}} = \arcsin\left(\frac{l_d\tan(\texttt{steer\_angle})}{2L}\right) \tag{10}$$

With this equation and l_d, we now have the pose of the new, steered point relative to the car's current position:

$$\begin{aligned} x &= l_d\cos(\theta_{\texttt{wp, new}}) \\ y &= l_d\sin(\theta_{\texttt{wp, new}}) \end{aligned} \tag{11}$$

where $\theta_{\texttt{wp, new}}$ is the net change in heading, as specified by the Pure Pursuit relationship, yielding a point $x_{\texttt{new}}$ and pose `new_pose`.

Next, we check if the path from the designated parent to the steered point is collision-free by comparing 1000 points between them to the occupancy grid, ensuring all are unoccupied. If so, we add the new pose to the tree.

If this new point is within a prespecified `goal_radius`, we return the path to the new point by using (tracing) stored parent pointers from $x_{\texttt{new}}$'s respective node to $x_{\texttt{start}}$, and publishing a `PoseArray` as with A*.

2.6 Path Processing

If desired, we simplify generated paths by iterating through each point in the original path and evaluating whether it lies on a straight segment. If the cross product of adjacent path segments is below a tolerance, 0.0018 in our case, the point is considered to lie on a straight line and is excluded from the simplified path. The final point is always retained, as it represents the goal.

2.7 Pure Pursuit

Our pure pursuit algorithm iteratively calculates the steering angle required to reach a point located a specified lookahead distance ahead of the racecar along a given path. This is based on the bicycle model, yielding a feasible trajectory for our Ackermann-steered racecar. We measured our racecar's wheelbase to be 0.1 meters.

Given a planned piecewise linear path, we find the lookahead point with a circle centered on the racecar with a radius equal to the lookahead distance and finding its intersection(s) with the path. If multiple, we select the one furthest along the path. If none, the lookahead point defaults to the closest point on the path to the racecar.

After identifying the lookahead point, we compute the angle between the robot's current heading, θ_{wp}, and the lookahead point using the robot's pose estimated from localization and calculate the robot's steering angle.

$$\texttt{steer\_angle} = \arctan\left(\frac{2L\sin\theta_{\texttt{wp}}}{l_d}\right) \tag{12}$$

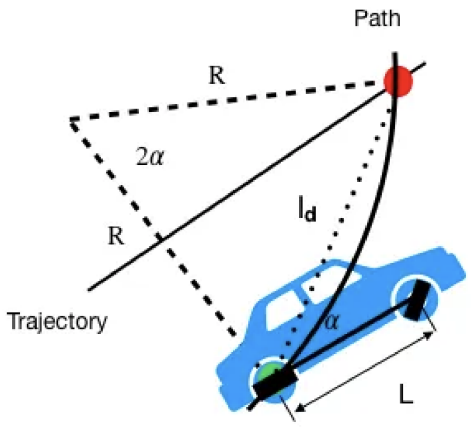

Figure 3: This diagram illustrates the geometric relationship used in the Pure Pursuit algorithm, where we determine the curvature required to reach its goal point. Figure and algorithm adapted from [1]

3 Experimental Evaluation

Author(s): Andy Yu

3.1 A* Evaluation

We evaluated our A* implementation by measuring the time to generate the internal map representation upon receiving an `OccupancyGrid`, and path computation time after receiving a start and goal position. All evaluations were conducted in simulation on a 2020 MacBook Air with an M1 chip running macOS 15.4.

We defined three test scenarios —short, medium, and long — with a fixed start ((0,0)) and varying goal positions of (-15.0, 12.0), (-20.0, 34.0), and (-55.0, 35.0), respectively. We collected 11 data points per scenario. These are shown in the figure below.

On average, map generation took 1.23 seconds. Path planning for the short, medium, and long tests averaged 2.78, 8.20, and 10.28 seconds, respectively.

Figure 4: Visualization of test cases: Top-left—medium, top-right—short, and bottom—long. Green spheres represent start points, red spheres represent end points, and the white path shows the generated trajectory.

3.2 RRT Evaluation

Our RRT implementation consistently failed to complete the short and medium tests due to narrow passages near a pillar, limiting path discovery. RRT runtime averaged 559.00 seconds over three successful runs in the long case. Given RRT's significantly higher runtime and reduced reliability in constrained environments, we concluded that further evaluation was unnecessary and relied exclusively on A* for future tests.

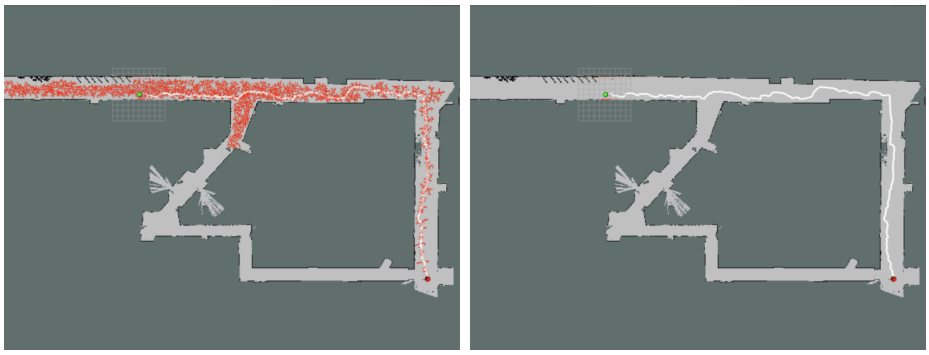

Figure 5: Visualization of RRT in the long test case: Left—search tree visualized by samples represented as red arrows. Right—the generated path (in white) from start (green) to end (red).

3.3 Path Processing Evaluation

We evaluated path simplification with the short, medium and long test cases. For each, we compared the number of points before and after simplification, qualitatively assessing the visualized paths to ensure accurate approximation of the original.

In the short case, the original path contained 579 points. With thresholds of 0.0018 and 0.003, the simplified paths contained 92 and 2 points, respectively.

In the medium test case, the original path had 1232 points. With thresholds of 0.0018 and 0.003, the simplified paths contained 90 and 3 points, respectively.

In the long test case, the original path included 1734 points. With thresholds of 0.0018 and 0.003, the simplified paths contained 74 and 2 points, respectively.

Qualitatively, the paths simplified using the 0.0018 threshold were visually indistinguishable from the originals. In contrast, paths simplified with the 0.003 threshold did not accurately represent the original trajectories. This data is presented in the table below and illustrated in the figure below.

Test Case	Original Points	Points (0.0018)	Points (0.003)
Short	579	92	2
Medium	1232	90	3
Long	1734	74	2

Table 1: Path simplification results using two cross-product thresholds (0.0018 and 0.003) across short, medium, and long test cases.

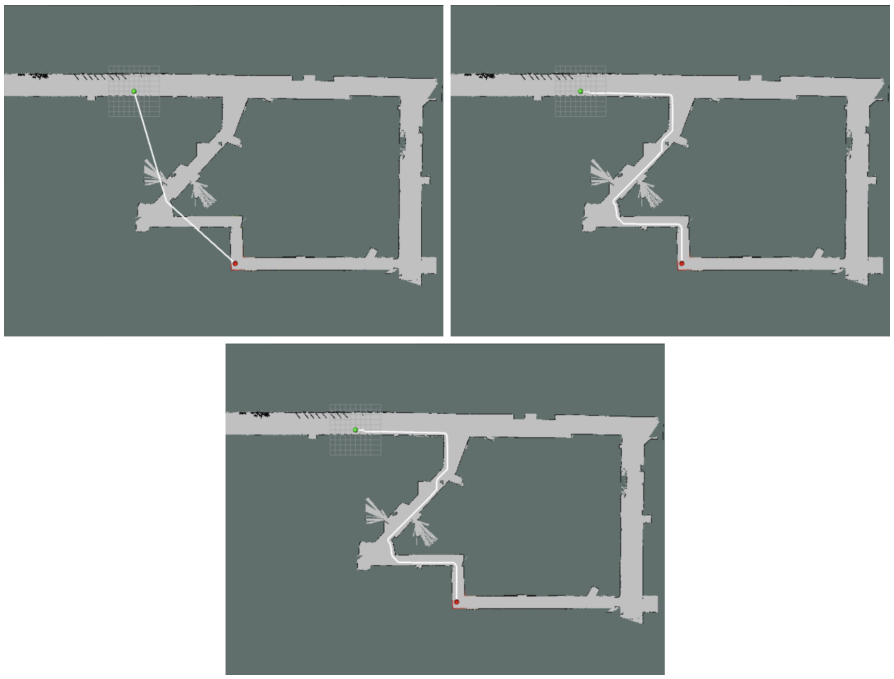

Figure 6: Visualization of path simplification with different threshold values: Top-left—0.003, top-right—0.0018, and bottom—no threshold. Green spheres represent start points, red spheres represent end points, and the white path shows the generated trajectory. The 0.0018 and no-threshold images are nearly identical, while the 0.003 threshold image shows an oversimplified path.

3.4 Particle Filter Integration Evaluation

We evaluated path-following cross-track error in simulation using both the ground truth position and the localized position estimate. We used a lookahead distance of 1 meter and a constant speed of 5 meters per second.

Results between ground truth and localized pose estimate were vi-

sually similar. The mean and standard deviation of the cross-track error for the ground truth runs were (-0.0395, 0.2200), (-0.0107, 0.2524), and (-0.0314, 0.1669) for the short, medium, and long test cases, respectively. For the particle filter runs, the corresponding values were (-0.0177, 0.1668), (-0.0031, 0.1844), and (-0.0052, 0.1210). These results indicate our localization system integrates effectively with the path planning/following algorithms.

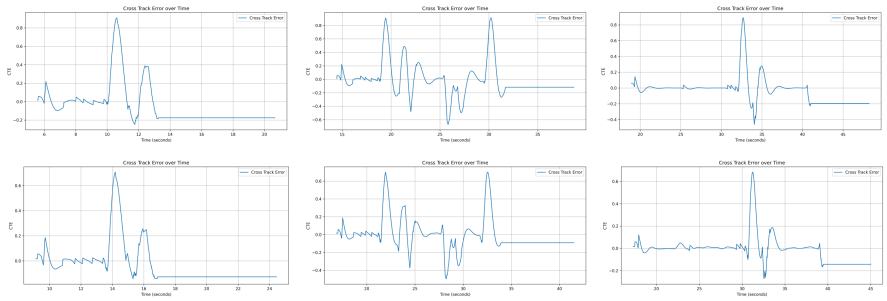

Figure 7: Cross-track error plots over time: From top-left to bottom-right: Ground truth, short test case (Mean, Std: -0.0395, 0.2200); Ground truth, medium test case (-0.0107, 0.2524); Ground truth, long test case (-0.0314, 0.1669); Particle filter, short test case (-0.0177, 0.1668); Particle filter, medium test case (-0.0031, 0.1844); Particle filter, long test case (-0.0052, 0.1210).

3.5 Lookahead Distance Evaluation

We tested lookahead distances of 0.25, 0.5, 1.0, and 1.25 meters at a constant speed of 1 meter per second to ensure easier translation to similar, expected, real-world conditions. Initially, we planned to evaluate performance using our particle filter's estimated pose. However, due to localization instability in parts of the test cases, we opted to use the ground truth pose provided by the simulation for more consistent evaluation.

The results showed a 0.25-meter lookahead distance yielded the best performance, with minimal cross-track error. The mean and standard deviation of cross-track error for each distance were as follows:

1. 0.25 m: (0.0046, 0.0643)
2. 0.5 m: (0.0070, 0.1031)
3. 1.0 m: (0.0251, 0.2793)

4. 1.25 m: (0.0887, 0.4575)

The cross-track error over time is visualized in the figure. Based on these results, we concluded that a 0.25-meter lookahead distance provides the most accurate path tracking at 1 m/s.

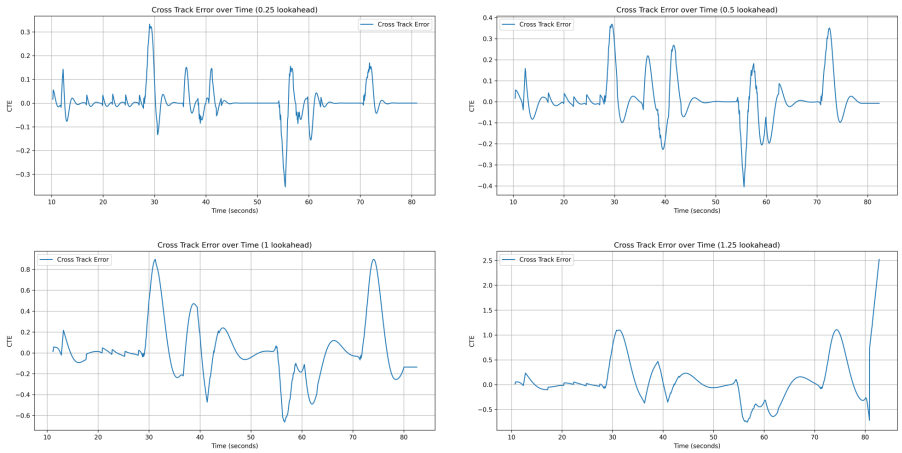

Figure 8: Cross-track error plots over time for varying lookahead distances: From top-left to bottom-right—0.25 m (Mean, Std: 0.0046, 0.0643); 0.5 m (0.0070, 0.1031); 1.0 m (0.0251, 0.2793); and 1.25 m (0.0887, 0.4575).

3.6 Real World Evaluation

Finally, we evaluated real-world performance by setting short paths between start and goal points in a section of the Stata basement where our localization performs reliably. Our primary evaluation metric was qualitative: we used RViz to visually assess whether the estimated pose and generated trajectory aligned with the car's actual behavior. Additionally, we measured cross-track error between the estimated pose in the basement and the planned trajectory using a successful ROSBAG run.

Through iterative testing, we found that a 2-meter lookahead distance minimized oscillation and provided the most stable performance. While this value differs from the optimal 0.25-meter lookahead identified in simulation, the discrepancy is likely due to increased localization uncertainty in real-world conditions. The longer

lookahead distance helps smooth out erratic localization estimates.

Qualitatively, the racecar followed the planned paths reliably in the real world, with the vehicle's motion and estimated pose closely matching the output on RViz. Quantitatively, we observed that the cross-track error was lower than in simulation and predominantly positive, with a mean of 0.2143 and a standard deviation of 0.1124, as shown in the accompanying figure. We attribute this to the low curvature of the test path and the specific nature of the initial conditions. A visualization of our observations is shown in the figures below.

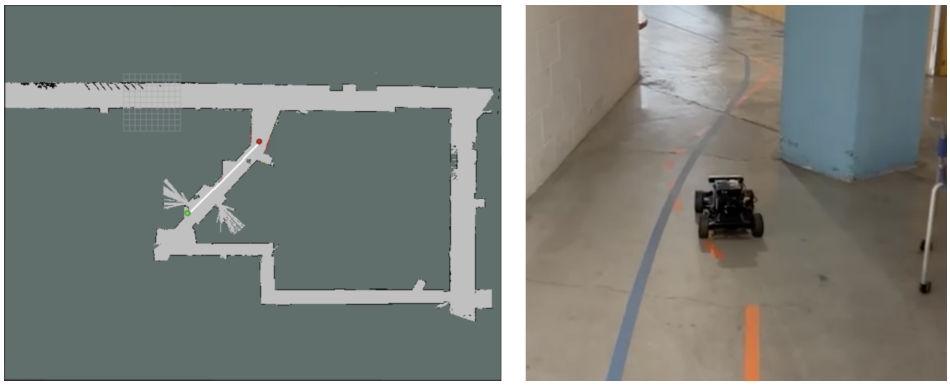

Figure 9: Visualization of real-world qualitative evaluation: Left—RViz visualization used to assess alignment with the real world. Right—the racecar navigating the real environment.

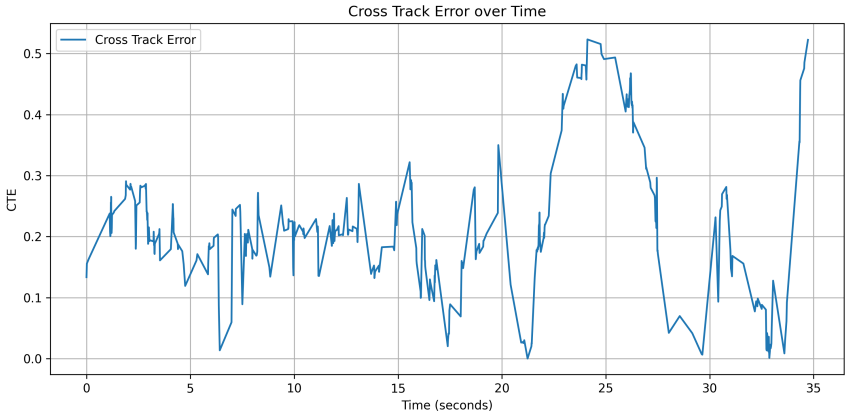

Figure 10: Cross-track error over time between the estimated trajectory and the planned trajectory during real-world tests.

Overall, we conclude that in a feature-rich environment with a low-curvature path, our racecar is able to successfully generate and follow a trajectory in real-world conditions.

4 Conclusion

Author(s): Adan Abu Naaj, Amina Luvsanchultem, Dylan Gaillard

In this design phase, we utilized our MCL solution from Lab 5 and expanded its usage to shift to implement A* and RRT path planning algorithms. We implemented a full navigation pipeline that enabled our racecar to plan and follow collision-free trajectories in both simulation and the real world. We integrated a search-based A* path planning algorithm with a Pure Pursuit controller to guide the car from a starting point to a goal pose.

In implementing A* and RRT, we compared the difference between search-based and sampling-based path planning algorithms. We observed that our implementation of search-based A* was generally faster and produced smoother paths, successfully navigating and finding a path even when the goal was far from the start within at most roughly 10 seconds. In contrast, while RRT is generally better for complex maps where a discretized search space becomes infeasible, we observed declining performance in cases involving long-range

planning or narrow spaces — making it less reliable than our A* implementation. This was evidenced, for example, by the near 10-minute runtime of RRT on the "long" test case compared to the 10-second runtime of A*, and RRT's failure in other tests to navigate narrower areas of the map.

Looking ahead, we aim to improve the performance of RRT on a broader range of test cases, reduce Pure Pursuit's tracking error — especially around sharp corners — and tune our robot on higher speeds in preparation for the final challenge. This will help prepare us for the cumulative design of the final challenge, where robust, reliable, adaptive, and high-speed autonomous navigation will be essential for our success.

5 Lessons Learned

5.1 Adan Abu Naaj

This lab was a major step forward toward building a complete autonomous car. It was exciting to build upon and combine concepts from previous labs like MCL and Pure Pursuit to implement path planning and following. One of the main things I learned was the difference between search-based and sampling-based planning algorithms, and the trade-offs between A* and RRT in terms of speed, smoothness, and path-complexity.

One challenge in this lab was evaluation—it wasn't immediately clear what metrics we should use to assess our path planning approaches. However, that pushed me to think more critically and reason through our results more carefully. It made the CI component more interesting, and I found myself improving my presentation skills and becoming more confident in communicating my ideas, both in our report and during the briefing.

5.2 Amina Luvsanchultem

I felt that Lab 6 had deepened my understanding of the interplay between localization, path planning, and autonomous navigation. In using our MCL implementation for this lab, it was interesting to see

the connection between our previous lab and this lab, and adapting our previous solution to path planning applications. Working on the RRT section of the lab helped me better understand the algorithm's strengths and weaknesses in contrast with the A* algorithm — insights we can take into the final challenge — particularly its advantages in navigating complex environments and the limitations we encountered in our implementation. From a team perspective, dividing responsibilities across sections allowed us to make steady progress; however, I think we could improve our workflow by setting intermediate goals prior to the lab deadline to ensure that every team member remains aligned and informed about the team's progress.

5.3 Andy Yu

The biggest lesson I learned was the necessity of splitting the work between teammates. If we had all invested our time into a single path planning algorithm, we wouldn't have had a working robot in any capacity in time for the lab briefing. In this lab, we somehow naturally divided the work in a way that allowed us to finish the lab. I hope that we continue to partition this effectively for the final challenge. Additionally, I realize now that this is the worst time to get complacent. We started to focus on lab work relatively late, and I believe that cost us greatly. However, it is extremely challenging to stay motivated, especially after getting repeatedly hit over the head by this class (and sure, our other ones as well) the entire semester. I hope we can scrape what drive we have left to create a reasonable implementation by the final challenge deadline.

5.4 Arthur Hu

My biggest lesson learned from Lab 6 was about how to integrate complex systems together. It was really satisfying to be able to take what we build before with our MCL and directly build on top of it to get our path planning algorithm. While the other labs have taken bits and pieces from previous weeks, I really felt like everything was coming together this time, and really drove home how developing good versatile modules can really help when building a bigger system. I think we also learned some lessons about how even though things are not always in our control, there are still local

optima that we can hit and victories we can get. As an example, our RRT initially was not faster than our A* algorithm, and we lost many hours one night to a hardware problem with our batteries, but in the end we managed to pull through anyways.
I think we also learned quite a bit about the difficulties involved in optimization and the trap of fine tuning parameters. It was often tempting to just try to change some numbers around and run the car out for another test when things weren't working, but in the end, it was the software optimizations and changes in our algorithm that truly brought us most of the way.

5.5 Dylan Gaillard

My main takeaway from Lab 6 is the insight I gained into the impact of small optimizations to our solution approach in the context of overall performance. Throughout this lab, a common challenge that I, along with my teammates, encountered was the significant inefficiency of our implementation of RRT. At a glance, our problem approach seemed sound, but after further review we found that making a small change to our approach structure — rejecting out-of-bounds samples before attempting steering—significantly improved its performance. Seeing how a seemingly minor change impacted our performance at the highest level has brought to my attention the importance of understanding both the high-level structure of any technical project, as well as the relation of specific individual components to each other. Additionally, I feel that this lab overall reinforced my understanding of how dividing a problem and creating modular solutions to individual problems — planning, localization, control — can easily lend itself to accomplishing a larger goal, such as navigating through the Stata basement.

6 References

[1] Y. Ding, "Three Methods of Vehicle Lateral Control: Pure Pursuit, Stanley and MPC," Medium, Aug. 20, 2020. https://dingyan-89.medium.com/three-methods-of-vehicle-lateral-control-pure-pursuit-stanley-and-mpc-db8cc1d32081