Path Planning Using Localization and A*

Team 10

Foland, Lexi
Fortt, Julia
Kim, Evan
Krebs, AZ
Peale, Will
RSS

April 24, 2025

1 Introduction: Julia Fortt

Path planning capability is an essential part of any mobile autonomous machine. It enables robots to navigate intelligently through their environments, and for some robots, to manipulate their limbs in ways that simulate how a human would. To explore this, the team began by evaluating two different options for path planning algorithms: A* and RRT. A* was then chosen for simplicity of implementation and combined with a pure pursuit controller to enable the team racecar to plan out and follow a path (given a start position, an end position, and a map).

2 Technical Approach

2.1 Selecting a Planning Algorithm: Julia Fortt

Since our first task was to choose an appropriate path planning algorithm, we began by carefully examining our options. Several possible algorithms were presented to us in lecture, including A*, RRT, RRT*, PRM and PRM*. Because of time constraints however, we quickly decided to narrow our focus to A* and RRT for familiarity and educational curiosity reasons, respectively.

A* path planning employs the standard A* graph search algorithm to find optimal paths. Using distance as the cost metric to determine "better" paths, it can be run on a grid or a sampling graph representing a map space, and is expected to find optimal paths. RRT, in contrast, uses random sampling to simultaneously expand and search through a space. It doesn't rewire itself if it finds a shorter path however, so it isn't expected to find optimal paths.

After examining the differences between A* and RRT, we ultimately decided to proceed using A* as our path planning algorithm because compared to RRT, A* found better paths and was simpler to implement. Although we assumed (correctly, as confirmed in a later experimental evaluation section) that A* path planning would require significantly more computation time than RRT as the size of the search space increased, we decided that choosing A* was worth the tradeoff given that the size(s) of the map(s) that we needed to plan paths on would be constrained for our purposes.

Despite moving forward with A* in our implementation, we implemented both A* and RRT path planning in simulation for learning purposes, and compare the two in the section to follow.

2.2 Implementing A*: Will Peale

In our implementation, we expand upon the typical A* algorithm, choosing a Hybrid A* approach for path planning. The planner constructs a path by discretely moving through a 3D search space (x, y, theta), employing both linear and angular motions to navigate the search space efficiently. Each state in the search space is considered a node, with each node containing information on the vehicle's position, orientation, and movement costs. The main task is navigating from the start to the goal by exploring neighboring nodes, comparing paths based on both movement and heuristic costs.

We utilize a priority queue to manage and expand nodes in order of their estimated total costs. At each step, new nodes are expanded from the current node by simulating vehicle motion using predefined steering commands. These simulations ensure that each motion respects the vehicle's non-holonomic constraints. The heuristic function, which is the Euclidean distance to the goal, assists in directing the search towards the goal more efficiently. Additionally, our implementation incorporates collision checks against a dynamically updated occupancy grid to avoid obstacles. We implemented a planning cost structure emphasizing simplicity and feasibility of paths. The costs take into account penalties for steering changes, steering angle and distance to obstacles. By tuning the costs, we can balance the need for an efficient (in terms of distance) path with a path that the vehicle can actually complete.

2.3 Implementing RRT: Evan Kim

While developing our path planning solution, we explored RRT (Rapidly-exploring Random Tree) alongside A* to determine which would better serve our needs. Our RRT implementation offered both standard planning and a kinodynamic version that accounted for our racecar's Ackermann steering constraints.

The core strategy of RRT is building a tree by sampling random points and connecting them , which is fundamentally different from A*'s grid-based search. We used a bicycle model to represent the racecar's movement capabilities, with parameters matching our hardware: 0.325m wheel base, 0.2 radian maximum steering angle, and accurate vehicle dimensions for collision detection.

To handle obstacle avoidance efficiently, we implemented a KD-tree for quick spatial queries of the occupancy grid. This was particularly valuable in the kinodynamic version, where we needed to verify that the entire projected vehicle path remained collision-free during simulated motion steps.

We incorporated other modifications to the basic RRT algorithm. A 10% goal bias helped guide the search toward the target, and our extension method considered multiple potential steering angles to create smoother paths. These refinements helped generate viable paths through the Stata basement.

Despite these successful implementations, we ultimately chose A* for our final solution. RRT paths tended to be more jagged and less predictable due to their random nature, while A* consistently produced smoother, shorter paths. RRT also occasionally failed to find a valid path in our testing environment. The computational performance advantage of RRT wasn't significant enough to outweigh these drawbacks, making A* the more reliable choice for our application.

2.4 Developing Our Pure Pursuit Controller: AZ Krebs

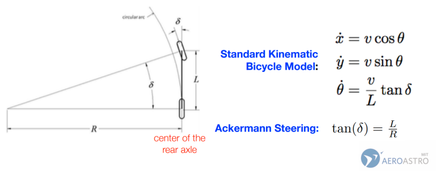
To follow the calculated path, we implemented a pure pursuit controller for our racecar. The pure pursuit controller is a geometric tracking algorithm that calculates a circular arc connecting the vehicle's current position to a goal point on a reference path. The controller commands the steering angle δ, calculated to follow the arc, to 'chase' this lookahead point along the path. However, when designing our controller, we needed to take into account the non-holonomic constraints a car-like robot presents. Thus, we adopted the standard kinematics bicycle model with Ackermann steering (see Figure 1) to model our vehicle and create our controller.

Based on these dynamics, the controller commands δ based on the lookahead distance L, and the radius of the calculated arc the car should follow to the vehicle's rear axle R. However, the geometry allows us to calculate that

$$R = \frac{L_1}{2\sin(\eta)}.$$

This substitution leaves us with

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



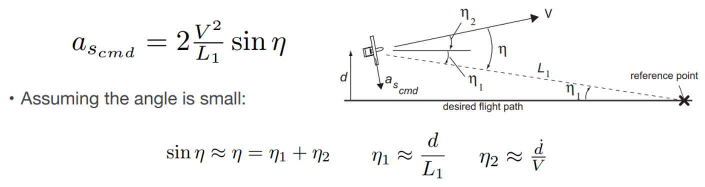
Standard Kinematic Bicycle Model:

$$\dot{x} = v\cos\theta$$
$$\dot{y} = v\sin\theta$$
$$\dot{\theta} = \frac{v}{L}\tan\delta$$

Ackermann Steering: $\tan(\delta) = \frac{L}{R}$

Figure 1: Visualization of the standard kinematic bicycle model and the associated equations of motion and the Ackermann steering angle [16.632 Lecture 05-06-control, slide 39].

Further, assuming small heading errors, the angle can be linearized as $\sin(\eta) \approx \eta$ (see Figure 2). We can decompose the total heading error into two components: one proportional to the cross-track error d, and one proportional to the rate of change of d. This resembles a PD controller with the proportional gain depending on the lookahead distance L_1 and the derivative gain depending on the vehicle's speed V.



$$a_{s_{cmd}} = 2\frac{V^2}{L_1}\sin\eta$$

• Assuming the angle is small:

$$\sin\eta \approx \eta = \eta_1 + \eta_2 \qquad \eta_1 \approx \frac{d}{L_1} \qquad \eta_2 \approx \frac{\dot{d}}{V}$$

Figure 2: Visualization of the decomposition of and the resulting linearization based on the small angle assumption [16.632 Lecture 05-06-control, slide 43].

Finally, we added the position point to be ahead of the rear axle, which improves stability (see Figure 3). This leaves us with the final commanded Ackermann steering angle:

$$\delta = -\tan^{-1}\left(\frac{L\sin(\eta)}{\left(\frac{L_{fw}}{2}\right) + l_{fw}\cos(\eta)}\right).$$

2.5 Integrating The Module on The Racecar: Evan Kim

Our racecar's navigation system integrated three primary components: localization, path planning, and path following.

The integration pipeline begins with our Monte Carlo Localization (MCL) system providing the racecar's current pose in the map frame. This localization module processes LiDAR scans and odometry data to maintain the vehicle's position and orientation estimate, with typical translational error under 0.5 meters and rotational error under 15 degrees.

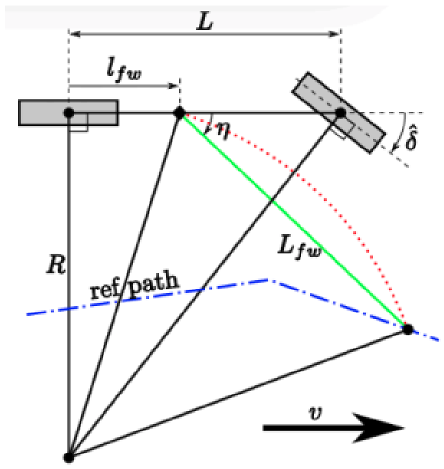


Figure 3: Visualization of the included position point ahead of the rear axle to increase stability [16.632 Lecture 05-06-control, slide 42].

Once localization establishes the racecar's current pose, this information feeds directly into our A* path planning algorithm to provide a starting position. When provided with a subsequent goal position, the A* planner generates a trajectory through the environment connecting the starting position to the target. This planning operation completes in under 1 second for paths within the Stata basement, creating trajectories that navigate around known obstacles.

The final integration step involves our pure pursuit controller, which takes the generated path and produces steering and velocity commands. The controller selects lookahead points along the planned path, with an adjusted lookahead distance (0.5m at 1m/s speeds) to balance responsiveness and stability.

We implemented a ROS2 architecture for this integration with the following communication structure:

1. Localization publishes the current pose estimate at 20Hz
2. The path planner subscribes to pose updates and publishes complete path plans
3. The pure pursuit controller subscribes to both the current pose and planned path, publishing drive commands at 40Hz

	A*: Average Computation Time [seconds]	**RRT: Average Computation Time [seconds]**
Path 1: Simple Turn	0.547	0.071
Path 2: Classroom Hallway	0.940	0.0384
Path 3: Corner to Corner of Basement	1.565	0.240

Table 1: A table comparing the average computation time in seconds of three different kinds of paths while using A* and RRT. The computation times were averaged over three trials for each kind of path. Images of the paths produced can be seen in table 3. A*'s computation times seemed to increase with path length, but they stayed considerably higher than the times produced while using RRT.

	A*: Average Path Length [meters]	**RRT: Average Path Length [meters]**
Path 1: Simple Turn	26.167	33.899
Path 2: Classroom Hallway	52.167	63.867
Path 3: Corner to Corner of Basement	88.667	108.

Table 2: A table comparing the average path length in meters of three different kinds of paths while using A* and RRT. The distances were averaged over three trials for each kind of path. Images of the paths produced can be seen in table 3. For every trial, A*'s paths were considerably shorter than those produced by RRT.

3 Experimental Evaluation: Lexi Foland

3.1 Selecting a Planning Algorithm

In this lab, we explored search-based and sampling-based planners for our racecar's navigation software. To decide the best algorithm for our needs, we gathered numerical data on the performances of search algorithm A* and sampling algorithm Rapidly Exploring Random Tree (RRT) in the simulated Stata Basement map. We considered two metrics, average computation time and average path length, over three different tests; the first featured a simple path with one turn, the second required navigation through the classrooms' shared hallway, and the third reached from one end of the basement map to the other. Table 1 shows the results of the computation time tests, and Table 2 shows those of

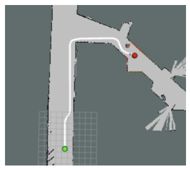
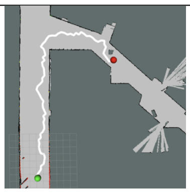
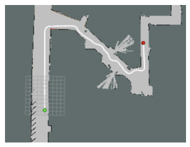
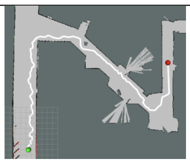
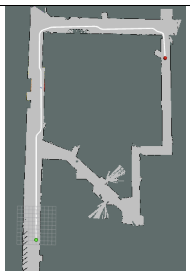
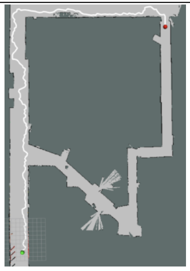
	A*	**RRT**
Path 1: Simple Turn		
Path 2: Classroom Hallway		
Path 3: Corner to Corner of Basement		

Table 3: A table comparing the shapes of the paths created by A* and RRT. A*'s trajectories tended to be smoother and track walls tightly. Contrarily, the randomness of the RRT algorithm caused its trajectories to appear jagged and roundabout.

the path length tests. While RRT required significantly less computation time than A* for each test, its average path lengths were considerably longer for all three trials.

Table 3 shows images of the paths created by the two planners for these three test cases. Due to its use of random point selection, RRT also created much more jagged paths than A* and occasionally failed to construct a trajectory. For these reasons, our team decided to move forward with A*, a search-based planner, as our primary algorithm.

3.2 Tuning the Pure Pursuit Controller

To follow the trajectories our algorithm created, we needed to develop a pure pursuit controller to send drive commands to our racecar. After doing so, we evaluated the accuracy of our pure pursuit controller at following simulated trajectories. Equation 1 shows the error metric used for this evaluation; this calculation represents the horizontal distance, or y-directional error, between the car's position and the lookahead point chosen in the pure pursuit algorithm, observed from the racecar's frame. Note that the x-directional error was not considered because pure pursuit controllers choose lookahead points that are a preset distance ahead in the racecar frame's x-direction.

$$\mathcal{E} = |target_y - vehicle_y| \tag{1}$$

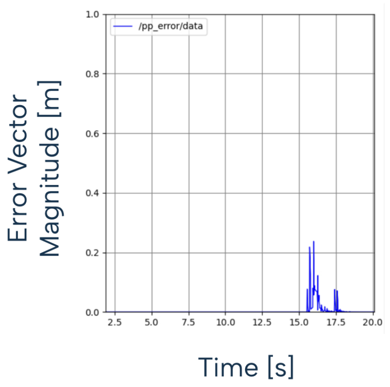
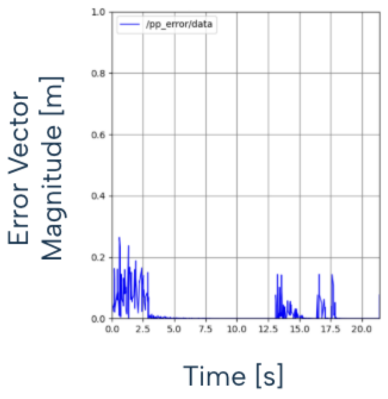


Figure 4: With a lookahead distance of 0.5 meters, this graph displays the magnitude of the y-directional error vector from Equation 1 against time while the racecar executed a basic trajectory using pure pursuit. The error stayed relatively low through the execution, only spiking to roughly 0.2 meters when the racecar took a sharp turn.

First, we evaluated the pure pursuit controller with a lookahead distance of 0.5 meters and a speed of 1 meter per second; Figure 4 shows the error metric graphed over time as the racecar followed a simple trajectory for roughly 20 seconds. The results showed consistently low error with a singular uptick around 16 seconds, which corresponded to the racecar taking a sharp turn. To tune the lookahead parameter, we also evaluated with similar speed and the distance set to 0.1 meters and 1.0 meters; the error recorded in these trials can be seen in Figures 5 and 6, respectively. Figure 5 shows that the 0.1-meter test experienced upticks in both the first few seconds and near 16 seconds, again corresponding to the trajectory's sharp turn. Figure 6 indicates that the 1.0-meter test had little error up until the sharp turn, similar to the default test; however, increased error was sustained from that point on, suggesting poor recovery after the turn.



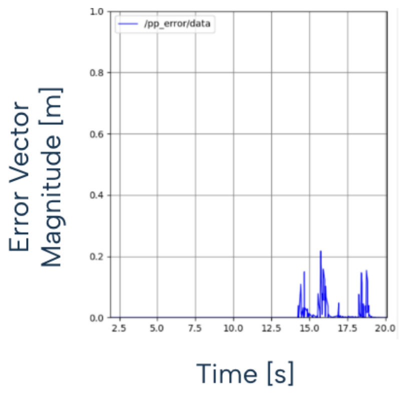


Figure 5: With a lookahead distance of 0.1 meters, this graph displays the magnitude of the y-directional error vector from Equation 1 against time while the racecar executed a basic trajectory using pure pursuit. The error varied during this trial, with spikes both in the beginning of the trial as well as consistent error before and after a sharp turn at around 16 seconds.

Figure 6: With a lookahead distance of 1.0 meter, this graph displays the magnitude of the y-directional error vector from Equation 1 against time while the racecar executed a basic trajectory using pure pursuit. Although the error remained considerably low during the beginning, the value spiked due to a sharp turn at around 16 seconds and failed to lower again quickly, unlike the trial with 0.5 meters as the lookahead distance.

After considering these evaluations, our team decided to keep 0.5 meters as our lookahead distance parameter when driving at 1 meter per second. We also noted that usage of higher speeds would likely require a larger lookahead distance; to test this assumption, we performed the same evaluations for a simulated speed of 2.0 meters per second. The optimal lookahead distance was approximately 0.75 meters, with evaluation results shown in Figure 7.

3.3 Evaluating the Integrated System on Hardware

The assignment required the integration of our path planning algorithm, our pure pursuit controller, and our localization software onto the real racecar, prompting an assessment of the effectiveness of the entire workflow. Given a trajectory from an initial pose to a goal pose planned via A*, as well as a pose estimate gathered via our localization software, we observed how well our pure pursuit controller directed the racecar in following the path. To test this, we again used our pure pursuit evaluation metric described in Equation 1, measuring the y-direction deviation from the path in the racecar's frame.

First, we tested the system on a short trajectory near the top of the basement

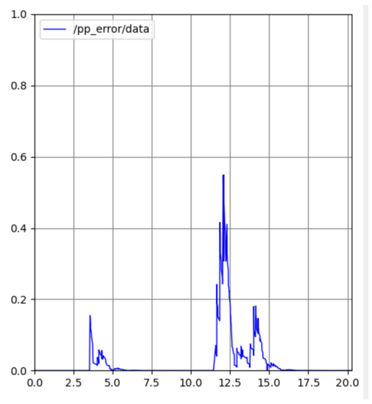


Figure 7: With a lookahead distance of 0.75 meters and a speed of 2.0 meters per second, this graph displays the magnitude of the y-directional error against time while the racecar executed a basic trajectory using pure pursuit. The error experienced a few spikes, likely due to the increased speed, with the highest jump appearing around 12 seconds while turning a sharp corner. Of the lookahead distances tested for 2.0 meters per second, the results for this trial were the best.

map, at a speed of 1.0 meters per second. The y-directional error, expressed in meters, was graphed over time as the racecar followed the path; the results are expressed in Figure 8. While the error stayed mostly below 1.0 meters, the significant oscillations encouraged us to reassess the accuracy of our software. After tuning our lookahead parameters and visually debugging our path planner, we still experienced similar results; ultimately, the issue was with the noise parameters of our localization model.

We raised the Gaussian noise added to the motion model in our particle filter, which solved the issues we were experiencing. We then tested the accuracy of our software on a longer path in the Stata Basement; the error metric in Equation 1 was graphed in comparison to the time elapsed, shown in Figure 9. Within a few seconds, the error dropped from its initial value and stayed considerably low for the rest of the run. Considering that the racecar seemed to also correctly follow the path in the real world, we took this as an indicator that the localization, path planning, and pure pursuit components were working well together.

While we were satisfied with the results of these experiments, we recognize that raising the speed to higher values, namely 4 or 3 meters per second, could potentially show different results. We anticipate needing to raise the lookahead distance in the pure pursuit controller for higher speeds and plan to test this theory in anticipation of the final challenge.

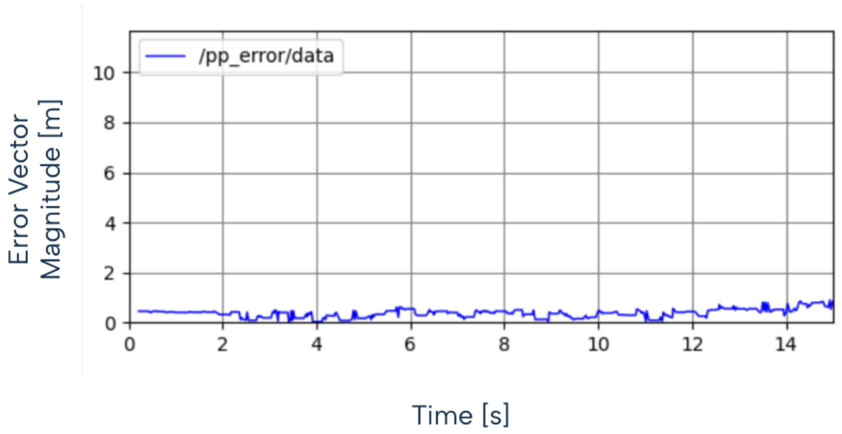


Figure 8: This graph shows the magnitude of the y-directional error vector in meters versus time in seconds as the racecar followed a short A*-generated trajectory in the real world. In this case, the current position of the racecar was estimated via localization software, and the desired position was the y-coordinate of the lookahead point used in the pure pursuit controller. Although error was relatively low, significant oscillations were still observed, and error eventually reached 1 meter towards the end of the trial.

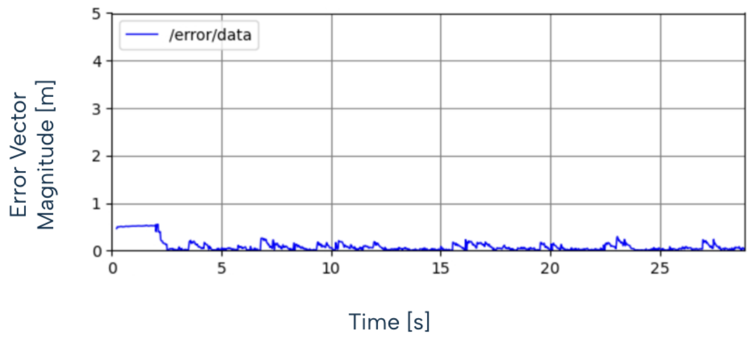


Figure 9: This graph shows the magnitude of the y-directional error vector in meters versus time in seconds as the racecar followed a long A*-generated trajectory in the real world. The current position of the racecar was estimated via localization software, and the desired position was the y-coordinate of the lookahead point used in the pure pursuit controller. After some initial error at the beginning of the trial, the metric converged to a relatively low value and stayed that way for the remainder of the test.

4 Conclusion: Julia Fortt

Having evaluated our path planning module both quantitatively and qualitatively, we're reasonably satisfied with our lab solution's accuracy. That said, we have more work we'd like to do to improve it, time permitting. Included among our future goals is to improve the speed of the path planning/pure pursuit pipeline, as well as the speed at which the racecar can reasonably (and safely) drive while following a planned path. Both of these goals have the potential to improve the practical functionality of our path planning module dramatically, but they're not our only goals. We'd also like to run additional tests planning/-following long paths for quantitative evaluation purposes, which we hope would make our module more robust and reliable.

5 Lessons Learned

5.1 Lexi Foland

This lab was a valuable introduction to a topic I have always been curious about: path planning in robotics. I was grateful for the opportunity to learn about this subject. I also realized how valuable teammates can be; a few of Team 10's members had prior experience with both search and sampling-based planners, enabling them to answer any questions I had about the project. Additionally, I learned the value of starting early. A working program implementation by Sunday night allowed us to complete more rigorous testing before the briefings and reports were due. Lastly, I now know the importance of checking over my additions to the briefings with teammates; they were able to point out some potential flaws in my explanations that ultimately made our presentation go much smoother.

5.2 Julia Fortt

I thought this lab was really interesting because I've always been curious about how robots can determine routes/actions autonomously. Although some of my assumptions about the process were correct, learning the differences between path planning methods like RRT and A* showed me that the answer to that question wasn't as straightforward as I imagined. Also, as a computer science major, it was also really cool to see a standard graph search algorithm like A* used effectively in practice.

5.3 Evan Kim

Having previously used A* only in theoretical contexts, seeing its application for robot navigation provided valuable practical insight and was very rewarding. Implementing RRT for the first time was also super instructive, particularly while tuning sampling parameters and observing its computational efficiency compared to A*. It was also very productive to work with others while testing the full integrated system, since their feedback revealed issues that might have gone unnoticed while working individually.

5.4 AZ Krebs

I learned a lot about the difference between sampling and search based path planning. I had no prior experience with path-planning so the higher-level decision making process was very informative. I gained a good understanding of the pros and cons of each type of path planning, and high level differences between the different methods (like RRT vs PRM and A* vs Dijkstras).

5.5 Will Peale

Working on this lab I learned a lot about the benefits and drawbacks of various planning techniques in robotics, and how there is no “right” solution. I also learned an important lesson that no matter how good your planning stack is, your localization has to be up to par as well or else the whole vehicle will not work.