Lab 6: Real-time Path Planning and Execution for Autonomous Navigation Between Two Points

Team #11

Timber Carey
Marcos Espitia-Alvarez
Subin Kim
Amy Shi
Christian Teshome

6.420 Robotics: Science & Systems

April 26, 2024

1 Introduction

Subin Kim

The core challenge in the robot's ability to navigate autonomously through an environment lies in the development of an efficient path planning and trajectory following. Those algorithms enable the robots to move from point A to point B effectively while avoiding obstacles. This lab report delves into the comparative analysis of two primary planning strategies: sample-based and search-based algorithms. Additionally, we explore the implementation of trajectory following techniques, namely the pure pursuit and Stanley Controller, evaluating their effectiveness through simulations and real-world applications.

Path planning serves as the backbone of autonomous navigation, allowing robots to calculate viable options when navigating through a complex environment. It involves generating a path that the robot can follow from the starting pose to the end pose (the goal), given that the robot circumvents any obstacles in the way.

This lab report focuses on two contrasting approaches to path planning. The first, a sample-based method, specifically Probabilistic Roadmap (PRM), relies on randomly sampling the space and connecting these samples to form a graph or a roadmap. The second, a search-based method, which systematically explores the environment to find a path using Breadth-First Search (BFS) algorithm.

The shortest path necessitates the ideal implementation of the path planning algorithm, in various scenarios the robot can find itself in.

2 Technical Approach

Subin Kim

This lab focuses on evaluating and comparing two primary path planning strategies: Sample-based and Searched-based methods. Each method has its unique strengths and weaknesses. Sample-based planners, such as the PRM that is implemented in this lab, are generally faster in high-dimensional spaces (more complex), though it might not always guarantee the most optimal path. In contrast, Search-based planners such as BFS, excel when we have to do frequent path calculations with a given prior information about the spaces. Search-based offers more effective routes, but at the expense of higher computational demands.

For trajectory following, we incorporated pure pursuit and Stanley methods. The pure pursuit uses a geometric approach in calculating the necessary steering angle based on a look-ahead point, while the Stanley uses both the cross-track(CTE) and the heading errors for making adjustments to the steering angle.

2.1 Path Planning

2.1.1 Obstacle Dilation

Timber Carey

Search algorithms tend to cut corners when planning an optimal trajectory to minimize speed or time. When feeding these planned trajectories into a driving controller, the controller may cut corners even more. However, because the car has some width, it cannot follow a path right up against a wall or other obstacle. In order to prevent the car from colliding with obstacles due to corner cutting, the obstacles were dilated before using the map to plan a trajectory.

The obstacles were dilated by performing a morphological operation called "erosion" on the occupancy grid. Using a disk element of 10 pixels, any space within 10 pixels of an obstacle on the original map would be considered "occupied" and the path planning algorithm would avoid those areas. A disk size of 10 pixels was chosen after an empirical analysis of different sizes. The disk size needed to be large enough so that the robot could properly avoid obstacles, but not so large that the space between walls would be completely blocked off. The chosen disk size balanced these considerations well.

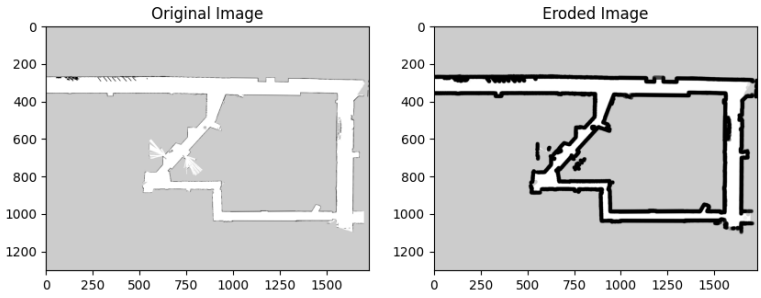


Figure 1: Occupancy grid map before and after erosion with a 10-pixel disk element. Black pixels are "occupied".

2.1.2 Transformation

Subin Kim

Transformation in robotic navigation plays a crucial role in planning paths accurately, and executing navigation through those trajectories. These are essential when robots need to localize themselves in a a map, comprehend their surroundings, and successfully navigate through the given map.

$$\mathbf{p}' = (\mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}) \cdot \mathbf{p} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{R}' & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}' & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{R}'\mathbf{S}' & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This is the form of the general-purpose transformation matrix

Figure 2: Equations used for transformation

The figure above encapsulates the operations we performed in this lab to transform coordinates for navigation, making sure that the robot understands its pose relative to its environment and can move accordingly. R represents the rotation matrix, S is the scaling matrix, t is the translation vector, and (x,y,z) are the original coordinates of the 3-D space that turns into (x',y',z') transformed coordinates.

When a robot receives its goal position in the global (world) frame, it must convert this data into its local frame of reference to plan the trajectory accordingly. Conversely, sensor data collected by the robot's local frame must be transformed into the global frame to successfully complete the mapping/localization process.

Thus, our group implemented two separate functions: "pixel_to_world", and "world_to_pixel" that performs those operations to allow the set of transformations between pixel coordinates and the world coordinates. This ensured that the robot navigated through the environment accurately, following the planned paths with precision.

2.1.3 Sampling-Based
Amy Shi

Two of the most commonly used sampling-based planning algorithms are the Probabilistic RoadMaps (PRM) and Rapidly-exploring Random Trees (RRT). There are many similarities between the two algorithms when considering their asymptotic optimality and complexity. They are both not asymptotically optimal, have a space complexity of $O(n)$, have a processing time complexity of $O(n \log(n))$, and are probabilistically complete. The Probabilistic RoadMaps algorithm is good for multiple queries and environments that have a roadmap. On the other hand, RRT is useful for single-queries and dynamic applications. It does not require a roadmap, which can often be hard to obtain. After considering both of the algorithms, we decided to implement our sampling algorithm based on the PRM algorithm due to the availability of the roadmap.

The PRM algorithm was implemented as follows:

1. First, generate a random set of points within the grid.
2. Then, each of the points are used to index into the given occupancy grid to determine if it is an available path or if there is an obstacle.
3. For the available coordinates, the closest k number of neighbors are found, and each edge and node is added to a graph structure.
4. Finally, the shortest path through the graph was found using the Dijkstra algorithm.

To optimize the existing algorithm, we implemented a number of steps. If there was a straight path from the start to the end pose with no obstacles in the way, the points are immediately added as a trajectory to the path. This enables

the algorithm to avoid the PRM altogether for straight paths. Furthermore, on our specific map, there were large sections of the map where there is no path available, so instead of generating points over the entire map, we narrowed the scope of the map so less randomly generated points can be used.

2.1.4 Search-Based

Christian Teshome

While the sampling-based path planning method works well in most cases, there were a few areas where there was still room for improvement. One such area is the computation time required to find the path. Also, if the distance between the start point and end point is very large, and there are a lot of obstacles in the way, sometimes it is unable to find a path due to the nondeterministic nature of the algorithm. Therefore, we decided to try out a search-based approach instead of a sampling-based approach.

There were a lot of possible search algorithms we could have used, including A*, Dijkstra's, and Breadth-First Search (BFS), however we concluded that BFS was the most appropriate algorithm for this task. Dijkstra's and A* both introduce a lot of unnecessary complexity, since they are designed to handle weighted graphs (graphs where each edge has an associated weight), and they also require us to make a graph structure of nodes and edges in our code to represent the map. With BFS, every edge has the same weight, and we can directly use our map to represent our graph.

Our implementation of BFS was fairly straightforward. We maintain a queue data structure, which initially contains our starting point, to represent the points we still need to traverse. The queue contains pairs of (point, path taken to get to that point). Then, the algorithm proceeds as follows.

1. Select and remove the point and path at the front of the queue
2. Check if the point is within range of the ending point, if so return the corresponding path
3. For each neighbor of this point, if we haven't already seen this neighbor yet and the neighbor isn't obstructed on the map, add it and its path to the end of the queue.

The algorithm always explores all paths of length n before any paths of length n+1, so it is guaranteed to always return the shortest path.

In order to determine the neighbors of a point, we decided to use an edge length of 3 pixels. For example, the neighbors of the point (11,141) would be (8,141), (14,141), (11,138), and (11,144). The trade-off when choosing to use 3 pixels was that, if our edge length was too small, the BFS would take too long to run. But, if our edge length was too high, then it might miss paths that go through tight passageways with obstacles, such as the pillar in the basement of Stata.

Overall, our BFS implementation was able to efficiently and reliably generate paths that are straightforward and easy for the car to follow, in comparison to the sampling-based method.

2.1.5 Path Smoothing

Timber Carey

After the trajectories were planned, there were still some inefficiencies where the path would not go straight to the next point. We decided to implement some post-processing to the planned paths to rectify this inefficiency. The smoothing algorithm would take the first point in a path and iterate through the points that followed, checking if a new path could be drawn between the first point and each point after without a collision. After finding a point along the path that would run into a collision, it would go one point back and add that collision-free path to the new trajectory. The algorithm would then start over and check collisions from the new path endpoint to each of the points after that. This process would repeat until reaching the end of the original trajectory, resulting in a much smoother path without excessive curves and extra nodes.

The smoothing algorithm efficiency was analyzed in terms of path length and path-planning time for both the search-based and sampling-based planning algorithms, with results shown in the tables below.

Table 1: Smoothing time for 5 different path types, for search-based and sampling-based algorithms.

	Search-based	Sampling-based
Corner 30m	0.000896s	0.000708s
Corner 20m	0.000909s	0.000388s
Corner 10m	0.000345s	0.000412s
Straight 20m	0.000723s	0.000111s
Straight 10m	0.000390s	0.000332s

Table 2: Unsmoothed and smoothed path distances for 5 different paths, for search-based and sampling-based algorithms.

	Se. unsmoothed	Se. smoothed	Sa. unsmoothed	Sa. smoothed
Corner 30m	37.76901m	31.87783m	35.49212m	32.85013m
Corner 20m	27.85132m	21.82571m	25.75461m	24.00701m
Corner 10m	13.7592m	11.04613m	12.84132m	11.45070m
Straight 20m	28.3248m	21.29389m	20.15527m	20.15527m
Straight 10m	14.0616m	10.63314	9.767203m	9.767203m

The path smoothing adds a negligible amount of time to the path planning algorithm, but there is a non-negligible increase in efficiency in terms of path distance. On corner paths, the smoothing yields a path with a distance 19% shorter on average for search-based paths and 8% shorter on average for

sampling-based paths. On straight paths, it yields a path with a distance 25% shorter on average for search-based paths. The distance efficiency increase is negligible for straight sampling-based paths because for straight paths without obstacles, the sampling-based algorithm simply skips the sampling process and returns a straight line between the start and end points.

2.2 Trajectory Follower

Marcos Espitia-Alvarez

The representation of our trajectories by the path planner gives us the capability to inform our controller efficiently. Our trajectory is given as an ordered list of tuples, with each tuple representing a node in the trajectory. This can easily be turned into an array of points, where the row index is the node's position in the trajectory

$$\overrightarrow{\boldsymbol{J}} = \begin{pmatrix} x_0 & y_0 \\ x_1 & y_0 \\ \dots & \dots \\ x_i & y_i \end{pmatrix} \tag{1}$$

Selecting a 'go-to' node allows us to do calculations based on only relevant information rather than doing computations over the whole trajectory. This is in response to a recommended strategy given to us by the original lab documentation - find a point on the nearest trajectory segment to the car then find a goal/look ahead point on that segment and use a Pure Pursuit Controller. Although that process can be made more efficient with matrix calculations in NumPy, it is excessive, especially considering that we know the order of our segments. Instead, we select a 'go-to' node, initialized as the node at index 0, check until we reach a certain criterion, and switch to the next node. The criterion we selected was that the car was less than one meter from the 'go-to' node. Once the car is within that distance of the final node, we stop our car.

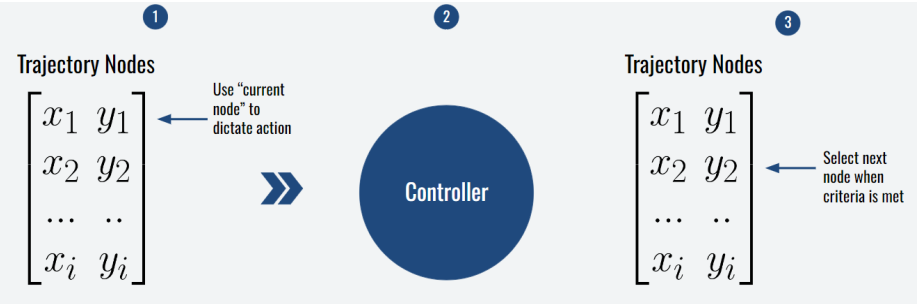


Figure 3: Diagram of the high-level process of our trajectory follower.

We implemented a Stanley Controller, over a Pure Pursuit Controller, to exploit the representation of our trajectory and do minimum computation. We do not need to consider intersection points or look-ahead distance like we normally would on a Pure Pursuit Controller. We find the segment we are currently

following by using our 'go-to' point as the end of the line and the previous point as our start. We create a vector representing this line.

$$\overrightarrow{l} = \overrightarrow{n_i} - \overrightarrow{n}_{i-1} \tag{2}$$

For a Stanley Controller, the steering angle, δ, is determined by the equation:

$$\delta = K_h\psi + tan^{-1}(\frac{Ke}{K_s + v}) \tag{3}$$

Where K_h is a gain to determine how response our steering is to heading error ψ, e is Cross Track Error (CTE), K_s is a softening constant to ensure that the arctangent is never undefined, K is another gain to determine responsiveness to CTE, and v is the speed of the car. For our model we used $K_h = \frac{1}{3}$, $K_s = 0.000001$, $K = 0.74$, and $v = 1.0\frac{m}{s}$ For the cases where δ is less than or greater than the max steering angle, which is 34 degrees for the car, the number gets clipped to the bounds. For example, if the controller determines we need a steering angle of 42, it gets clipped to 34. If it were -100, it would be clipped to -34.

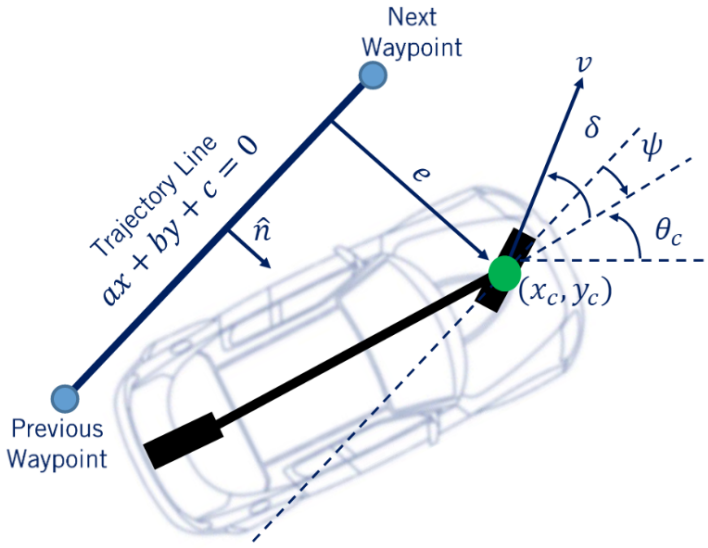


Figure 4: Schematic of a vehicle and the necessary quantities for a Stanley Controller. As the image implies, we only need to use two points without the need to determine intersections. With our implementation, we did not need to calculate the trajectory line and opted for vectorized operations, which removes the possibility of erroneous behavior when the line is right in front of the car (slope $= \infty$).

When the car's pose is given by $\overrightarrow{p} = (x_p, y_p)$. CTE is give through this process. $\overrightarrow{w} = \overrightarrow{n}_{i-1} - \overrightarrow{p}$ and $\hat{l} = \frac{\overrightarrow{l}}{|l|}$. Then $|CTE| = |\overrightarrow{w} \cdot \hat{l}|$ and the sign is given by the sign of $\overrightarrow{w} \times \overrightarrow{l}$. The signed CTE, e, is then $|CTE| * sign(\overrightarrow{w} \times \overrightarrow{l})$.

The Heading Error, ψ, is given by this process. $\alpha = \angle \overrightarrow{l}$ contained to $-\pi < \alpha \leq \pi$ (arctan2). The car's heading is θ_p, which is given to us. The final Heading Error, ψ, is given by $\psi = \alpha - \theta_p$ wrapped such that the angles are mapped to $-\pi < \psi \leq \pi$.

3 Experimental Evaluation

3.1 Simulation - Path Planning

Amy Shi

After implementing both the search-based and sampling-based algorithms, we evaluated their performance based on the run-time of the program and efficiency of the path. The efficiency of the path is determined by the distance from the start to the end point. There were five different paths (of start and end points) used to evaluate the performance of the path-planning algorithms. To account for driving in a straight path versus making a turn, we decided to evaluate planning corner paths with distances of 30 meters, 20 meters, and 10 meters. For straight paths, paths with distances of 20 meters and 10 meters were used for the starting and ending points. The algorithms that were evaluated were the search-based algorithm and variations of the sampling-based algorithm that had different number of random points to generate.

S-B: Search-Based Algorithm
Sa-n: Sampling-Based Algorithm with n randomly generated particles

Table 1: Time to Plan Path for Search and Sampling Algorithms

	S-B	Sa-10000	Sa-7500	Sa-5000
Corner 30m	0.047668s	0.173031s	0.181309s	0.168947s
Corner 20m	0.060877s	0.144650s	0.134534s	0.152766s
Corner 10m	0.024196s	0.159155s	0.141946s	0.136779s
Straight 20m	0.029518s	0.000388s	0.000380s	0.000653s
Straight 10m	0.008854s	0.0002663s	0.000355s	0.000260s

It is worthwhile to note that while the sampling-based algorithms had a significantly faster run-time in terms of the straight paths, this is a result of the optimization described in the Technical Approach section. The optimization could be implemented for the search-based algorithm as well, so the straight path run-times above does not have a bearing on the actual efficiency of the PRM algorithm itself.

Table 2: Distance of Path Based on Algorithm Algorithms

	S-B	Sa-10000	Sa-7500	Sa-5000
Corner 30m	31.87783m	32.85013m	32.25962m	33.66555m
Corner 20m	21.82571m	24.00701m	23.20827m	22.83109m
Corner 10m	11.04613m	11.45070m	11.86916m	12.12411m
Straight 20m	21.29389m	20.15527m	20.01282m	20.22974m
Straight 10m	10.63314m	9.767203m	9.764862m	9.978945m

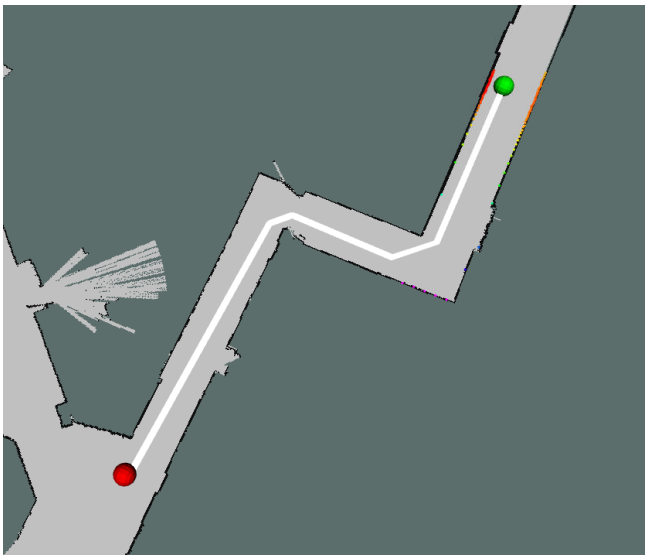
Overall, in terms of run-time, search-based algorithm performed significantly better, with a run-time that's 268% faster than the fastest sampling-based average run-time. As for distance, the search-based and sampling-based algorithm

did not vary significantly in their performance. However, the search-based algorithm produced, on average, paths that were 0.088m shorter than the sampling-based algorithm. Therefore, we decided to go with the search-based implementation for lab 6 because it was much more efficient computationally.

3.2 Simulation - Trajectory Follower

Marcos Espitia-Alvarez

Evaluation of the two controller methods proved that a Stanley Controller is a better option. The controllers were tested on two example trajectories, with the car maintaining a constant 1 $\frac{m}{s}$.



(a) Example Trajectory 3

(b) Example Trajectory 2

Figure 5: The example trajectories used for experimental evaluation of the controllers. The curving paths and closeness of the walls provide a good test of the effectiveness of the controller.

The Cross Track Error (CTE) and Heading Error accumulated by the Stanley Controller are less than those of the Pure Pursuit Controller across multiple trajectories. Not only that, but the Pure Pursuit controller would often create arcs that crash into the walls, meaning that such a path is not feasible

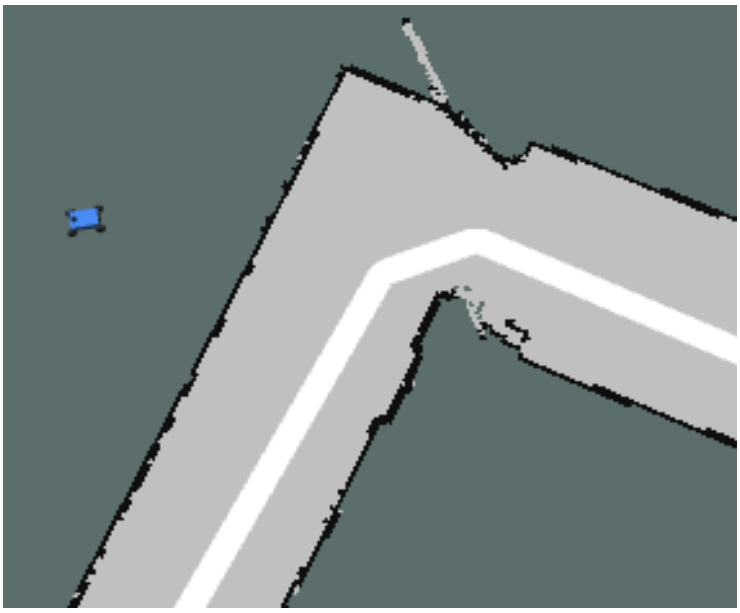
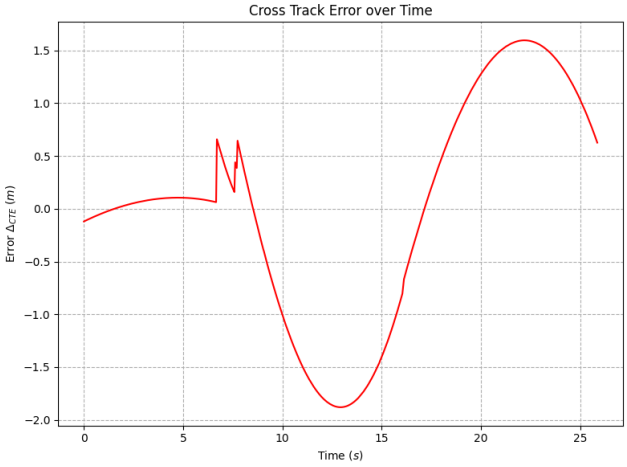
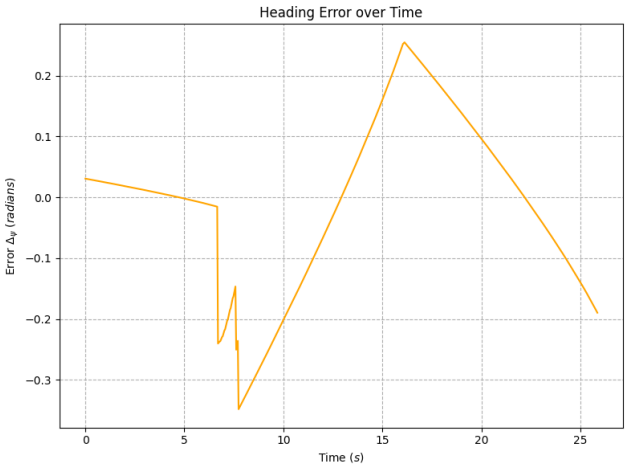


Figure 6: Image of the car following a trajectory that proceeds outside the bounds, therefore infeasible. This occurred during the evaluation of the Pure Pursuit on Example Trajectory 3.

The resulting plots of the CT and Heading errors across the two trajectories exemplify the power of the Stanley Controller. The following are the plots for Trajectory 2:

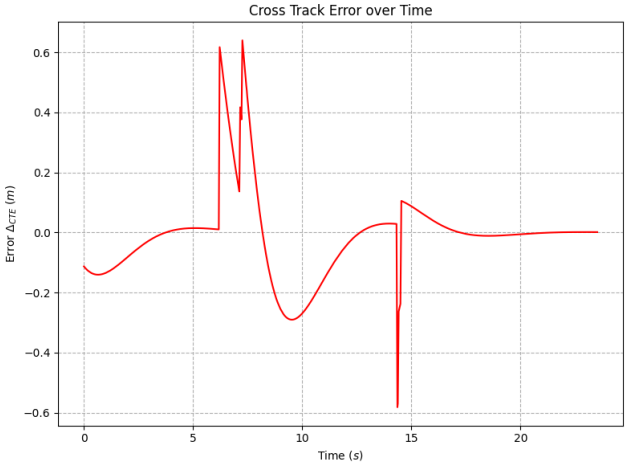


(a) CTE over Time for the Pure Pursuit Controller on Trajectory 2

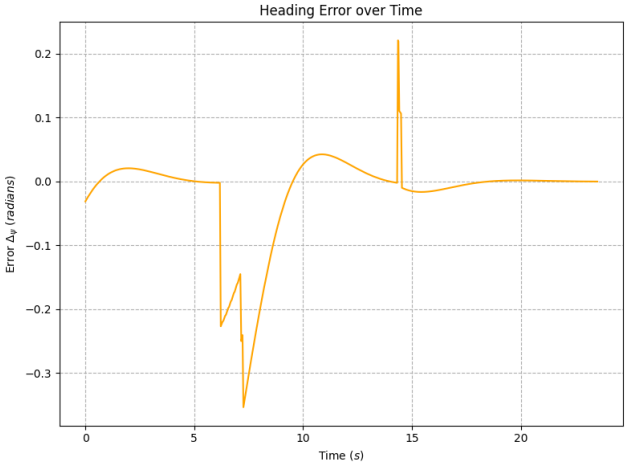


(b) Heading Error over Time for the Pure Pursuit Controller on Trajectory 2

Figure 7: Plotted is the CTE and Heading errors over time using the Pure Pursuit Controller on example trajectory 2 at 1 meter per second. The discontinuities indicate error being calculated when the 'go-to' node switches. For this trajectory the Pure Pursuit remained within the bounds of the map and peaks of the CTE were contained between 1.55 meters and -2 meters. The peaks of the Heading Error were contained between 0.25 radians and -0.35 radians



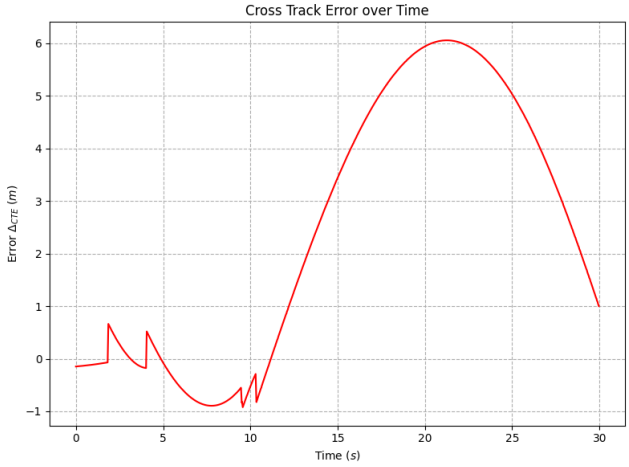
(a) CTE over Time for the Stanley Controller on Trajectory 2



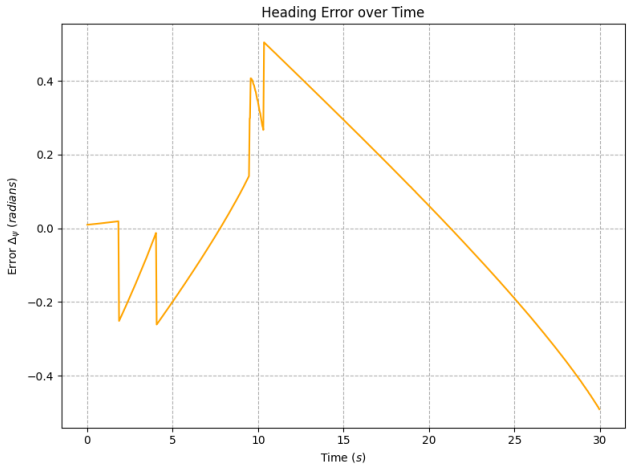
(b) Heading Error over Time for the Stanley Controller on Trajectory 2

Figure 8: Plotted is the CTE and Heading errors over time using the Stanley Controller on example trajectory 2 at 1 meter per second. The discontinuities indicate error being calculated when the 'go-to' node switches. For this trajectory the peaks of the CTE were contained between 0.6 meters and -0.6 meters. The peaks of the Heading Error were contained between 0.25 radians and -0.35 radians, however, the controller would converge more quickly to 0 after discontinuities, unlike the Pure-Pursuit. The Stanley Controller kept the car within the bounds.

The following are the plots for Trajectory 3:

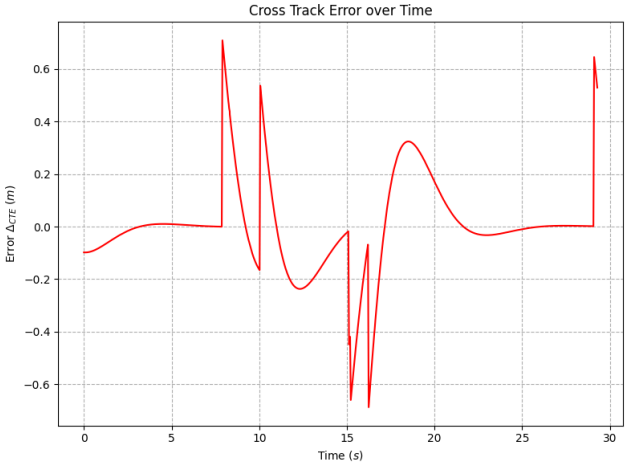


(a) CTE over Time for the Pure Pursuit Controller on Trajectory 3

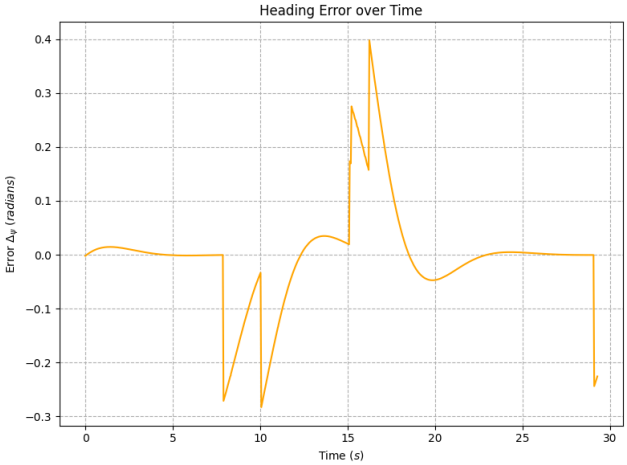


(b) Heading Error over Time for the Pure Pursuit Controller on Trajectory 3

Figure 9: Plotted is the CTE and Heading errors over time using the Pure Pursuit Controller on example trajectory 2 at 1 meter per second. The discontinuities indicate error being calculated when the 'go-to' node switches. For this trajectory the Pure Pursuit did not remain within the bounds of the map and peaks of the CTE were contained between 6 meters and -1 meters. The peaks of the Heading Error were contained between 0.45 radians and -0.4 radians.



(a) CTE over Time for the Stanley Controller on Trajectory 3



(b) Heading Error over Time for the Stanley Controller on Trajectory 3

Figure 10: Plotted is the CTE and Heading errors over time using the Stanley Controller on example trajectory 2 at 1 meter per second. The discontinuities indicate error being calculated when the 'go-to' node switches. For this trajectory the peaks of the CTE were contained between 0.65 meters and -0.65 meters. The peaks of the Heading Error were contained between 0.4 radians and -0.3 radians, however, the controller would converge more quickly to 0 after discontinuities, unlike the Pure-Pursuit. The Stanley Controller kept the car within the bounds.

Overall, the Stanley Controller outperforms the Pure Pursuit Controller by

a wide margin, converging to a steady state faster and more safely than the Pure Pursuit Controller, across different trajectories.

3.3 Real Life

Christian Teshome

When testing out our trajectory follower on the robot, we found that it was able to roughly follow its given trajectory, even with a lot of twists and turns in the path, as you can see in the figure below.

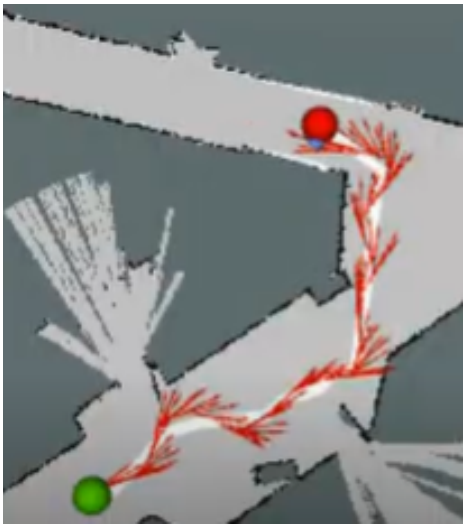


Figure 11: A demonstration of our trajectory follower being run in real life. The white line is the trajectory generated by the path planner, and the red arrows are the pose of the robot as it was following the trajectory.

Although the car was able to successfully reach the goal, we noticed that the car was oscillating back and forth a lot even when the trajectory it was following was relatively straight. Of course, it would be impossible for the car to follow the trajectory 100 percent accurately, due to the path planning algorithm not taking into account the steering limitations of the physical car, but we decided to try tuning the parameters of the Stanley Controller so that it follows the trajectory more tightly and with less wasted movement, even though it might not be perfect.

Adjusting the weights that the Cross Track Error and Heading Error had on the steering angle in our drive command allowed the car to follow trajectories much more smoothly. First, we tried dividing the heading error by 2, so that the car would respond less aggressively to errors involving its orientation in comparison to the trajectory it was following. Implementing this change was able to slightly mitigate the issue demonstrated in figure 10, however more improvements could still be made. After changing the weight of the heading error,

we then tried changing the weight of the cross track error, dividing it by 3, so that the car would respond less aggressively to errors in the car's perpendicular distance to the trajectory. This further improved our car's response to the controller. In the end, we decided on dividing both the cross track error and heading error by 3, as this led to the car being able to smoothly follow straight paths without unneeded oscillations, while also still being able to respond fast enough to sharp turns in the trajectory.

Table 3: Average Errors of Stanley Controller on Racecar

	Original	Both Divided by 3	Both Divided by 3 (Sharp Corner)
CTE	0.2359m	0.1672m	0.1526m
Heading	0.25151m	0.1419m	0.1696m

As you can see in the table above, dividing both our heading error and CTE error by 3 when determining the steering angle in our drive command significantly reduced our error, even in the case of a sharp corner in the trajectory, resulting in a stable controller.

4 Conclusion

Amy Shi

Through this lab, we learned about the various path-planning algorithms including BFS, Probabilistic Roadmaps, Rapidly-exploring Random Trees, and more. We were able to successfully complete the objectives of lab 6, which included implementing a path-planning algorithm and a trajectory follower. After various experimental evaluations to the various approaches, we found our most efficient path-planning algorithm to be the BFS search-based algorithm. It was the chosen algorithm because it allowed us to implement path-planning without having to account for weighted graphs, graph of nodes and edges representing our map, and other complexities that were unnecessary for the objective. As for the trajectory follower, the final implementation used a Stanley Controller because it allowed us to consider Heading and Cross Track errors while not worrying about intersection points and look-ahead distances. It also made the most sense given our vector representation of the trajectories. We were able to tune the parameters of the Stanley Controller to achieve a very low CTE and heading error.

However, there are still many improvements that can be made in preparation for the final challenge. In terms of the path-planning algorithm, some factors that can be taken into account when determining a path should include the steering and breaking limitations of the vehicle. In addition, to ensure the safety of the vehicle and the people around it, a similar approach to a wall-detecting algorithm can be implemented. Using LiDAR scans to dynamically alter our trajectory during path-following will allow the robot to avoid unaccounted obstacles that were not on the original occupancy grid. There can also be possible improvements in terms of the collision detection algorithm for two

points on the map for a smoother path between two points. As for the trajectory follower, our parameters can be further tuned for an even smoother drive in real-world path following.

With theses potential improvements to our current algorithms, our path-planning algorithm will be ready for the final challenge.

5 Lessons Learned

5.1 Timber's Lessons Learned

In this lab, I learned about the technical aspects of what goes into path planning. I've seen path planning used in different areas in the past, such as video games and other robotics applications, but I've never seen what actually goes into designing the algorithms. I've also had experience using search algorithms in the past, but haven't used them on any real applications. It was cool using breadth-first search for path planning and being able to see what kinds of things simple algorithms can do. It was the same with the erosion of the map, where I have seen morphological operations used on an image, but hadn't applied them to solving an actual problem. Throughout the whole lab, I got to apply skills that I've learned in many different places to solve the lab challenges, which was very eye opening and interesting.

In terms of non-technical skills, I learned how to better communicate with my team and solve problems together. In the past, it has been difficult to contribute to a portion of the lab that a teammate had already gotten started on, but I was able to ask my teammates about what they were working on and what they needed help with in order to contribute more effectively. For example, Amy started work on the sampling-based path planner, and I jumped in to contribute a bit later and helped with refining the code.

5.2 Marcos's Lessons Learned

While working on Lab 6, I gained valuable experience implementing a new kind of controller and working together with a team effectively while async. I have a heavy interest in controls and implementing a new controller is extremely rewarding and was an opportunity to go outside my comfort zone. To implement the controller effectively I also had to brush up on my linear algebra which was a welcomed opportunity, to say the least. I felt that by the end of this lab, I had gained more overall confidence in my technical abilities

While writing the report, I was not on campus and had to communicate with my teammates effectively to ensure I was still a helpful contributor. Not having the chance to work face-to-face with my teammates meant I had no safety net; I could not simply talk about changes or what I was doing with my teammates in

real time. However, this was a blessing too. The increased responsibility made me explicitly state what work I would be doing and the deadline. I would have to adhere to it because I knew the most about the topics I wrote about and wanted to be a reliable team member.

5.3 Subin's Lessons Learned

The various algorithms and concepts that were necessary to complete the lab included materials that I had been exposed to prior to taking this class. It was more smooth to pass the ideas back and forth with my teammates and debug the code. Being able to implement the algorithms that turn the wheels on the robot car was satisfying. Being with the team, our team were able to split up and communicate the progress throughout the duration of the lab. Through this, I feel like we were able to grow in the communications aspect of working with team members who are

5.4 Amy's Lessons Learned

During this lab, our team was able to better split up the tasks compared to previous labs. Especially with this lab containing a lot of components, delegating tasks became essential for great teamwork. It was precisely because of this that we were able to implement both the search-based and sampling-based path-planning algorithms.

As for the technical lessons learned, there were certainly a lot. I learned about potential ways to drastically improve the asymptotic optimality of the Probabilistic RoadMaps algorithm. This included dynamically changing the value of k neighbors to evaluate for each node. While I was not able to implement it, it is an interest I'd definitely like to further pursue. I also learned so much about how to carefully debug. This is one of the most important skills for programming, and the lab definitely forced me to develop a systematic approach for it due to the sheer size of the code.

5.5 Christian's Lessons Learned

On the technical side of things, I learnt a lot from this lab. For example, I learnt how to compare and contrast different algorithms and use my knowledge of the task at hand to determine which algorithm would be most suitable. Also, when implementing BFS for the search based path planning algorithm, I was able to practice the process of manipulating and transforming data (the map in this case) so that you can apply a black-box algorithm on it smoothly.

On the non-technical side of things, I was also able to learn a lot with regards to teamwork. For example, assisting my teammates with debugging and figuring out the best way to define and split up tasks so that we are all making progress. In this lab specifically, although the lab wasn't fully modularizable, we were still able to parallelize work much more than in previous labs, especially

near the deadline when it was time to start preparing evaluations, trying out different algorithms, refactoring code, etc.

Formatting/editing by Everyone.