Lab 6 Report: Search-Based Path Planning and Trajectory Pursuit

Team 19

Michael Wong
Alan Yu
Joshua Sohn
Owen Matteson
Ragulan Sivakumar
Shenzhe Yao

6.4200: RSS

April 27, 2023

1 Introduction (MW)

In the previous lab, we successfully achieved the localization of our car and accurately determined its position on the map. Now, we have taken a step further to make our autonomous car truly autonomous by learning how to drive it without human intervention. The focus of this laboratory exercise is on the core aspects of autonomous operation: planning and control. Our goal is to determine the safest and shortest path to our destination and ensure that our car drives along it smoothly.

The planning phase involves calculating trajectories from our car's current position to the goal pose on a known occupancy grid map using either a search-based or sampling-based motion planning method. We must ensure that the planned path is free from obstacles to guarantee a smooth and secure ride to our destination.

To execute the planned path, we will use the pure pursuit algorithm, which uses the particle filter and pure pursuit control to follow the predefined trajectory on the map. This algorithm plays a big role in allowing our car to navigate autonomously and precisely follow the path we have planned.

By combining these two components, we have successfully planned the path and achieved full autonomy as the car autonomously drives along the path.

2 Technical Approach

2.1 Problem Formulation (AY)

In this problem, we have access to a map M with occupancy grid O_M. Our objective is to take in a desired start and end pose $(\mathbf{p}_{start}, \mathbf{p}_{end})$ and perform the following objectives:

1. Plan a *feasible* path P between $\mathbf{p}_{start}$ and $\mathbf{p}_{end}$.
2. Follow the path with minimal error and stop at the end position,

where a feasible path is one that does not collide with any occupied cell in O_M.

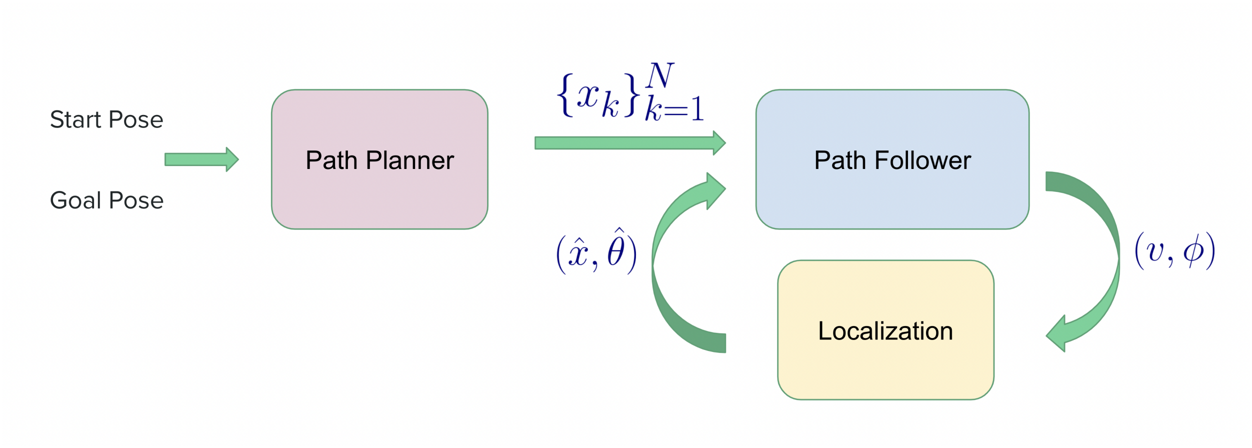


Figure 1: High level summary of the pipeline used for this lab. We receive a desired start and end pose and feed it to the path planner, which outputs a set of points that can be interpolated to produce a path. The path follower will then work in conjunction with the Localization module to control the car's movement along the generated trajectory.

We also assume access to the Localization module from the previous lab, which takes in odometry data and outputs an estimate of the car's pose in the frame of M. At a high level, our process (Figure 1) will consist of the following steps.

1. Receive $\mathbf{p}_{start}$ and $\mathbf{p}_{end}$ from the user.
2. Use the path planner to output a set of points $S = \{x_k\}_{k=1}^N$ representing the trajectory.
3. Interpolate (e.g. linearly) between points in S to produce path P.
4. Feed in P to the Path Follower to output driving controls v and ϕ (speed and the steering angle) to minimize error to P.
5. Receive feedback from Localization and repeat the control process.

We now discuss the implementation of each module.

2.2 Path Planning (RS)

With path planning, we are given a start and end position alongside an occupancy grid, which tells us where the walls are in our nearby environment, and our goal is to construct a path from that start position to the end. We also wanted this to return the shortest possible path and to be easily calculable, for during deployment, we cannot afford to spend too much time calculating an optimal path. Our overall strategy was as follows:

1. Convert the start and goal position from real-world coordinates to indices in the occupancy grid.
2. Use A*, an algorithm which can find the shortest path from a start position to a goal, on the occupancy grid to find a shortest path avoiding walls through the occupancy grid.
3. Convert that shortest path back to real-world coordinates.

We would then and pass that path on to our Pure Pursuit module, and our racecar would start following it.

2.2.1 Pros and Cons to Other Motion Planning Algorithms

We can lump motion planning algorithms into two categories: search-based and sampling-based. In search-based algorithms, we use a graph representation of the world and construct our path through said graph. Possible algorithms for such include breadth-first search (BFS), which finds the shortest paths to other vertices in a graph with unweighted edges, Dijkstra's, which finds the shortest path from a start point to all other points in weighted graphs, and A*, which can be used to find the shortest path from a start point to an end point.
One downside of search-based algorithms is that they can be computationally expensive. The runtime of BFS is $O(V + E)$, where V represents the number of vertices and E represents the number of edges, Dijkstra's is $O(V \cdot \log V + E)$, and A* is $O(E)$, so we can see that as the graphs get larger, the search-based algorithms get more expensive and thereby more infeasible practically. Thus, we have sampling-based algorithms as well. Instead of using an entire graph representation of the world, it randomly adds points to a set S, and we continue adding points until we are able to form a path to our goal with the points in S. With this randomness, sampling-based algorithm paths will not be optimal and could be dramatically longer than the shortest path, but they're also much more easily computable, giving the tradeoff between sampling-based and search-based algorithms.
We are given a graph representation of the world via the occupancy grid supplied. Thus, we don't need to make that representation. We only need to be able to convert from real-world coordinates to indices in the occupancy grid, which isn't that computationally expensive. Moreover, the occupancy grids aren't that big at the scale of 1000 by 1000. This makes search-based algorithms a good option and the one that we chose to pursue.
We chose specifically to use A* because it has the shortest asymptotic runtime. As will be explained later, it has the downside of requiring lots of memory to store potential paths in a queue, but we aren't concerned with memory here. Time is the only issue, so A* suits our needs.

2.2.2 How A* Works Through the Graph with A* (RS)

At a high-level, A* works by building up multiple potential paths. Every time we analyze a path that doesn't end in our goal, we add a new path composed by sticking a neighbor of the last point onto the path, and we do this for each neighbor of the last point. We continue as such until we get a path that ends in our goal.

However, if we analyzed paths in an arbitrary order though, we could end up wasting time analyzing and building paths that aren't heading towards the goal. Thus, to ensure we don't over-analyze sub-optimal paths, we analyze the paths from shortest to largest in terms of the distance underestimate. We define our distance underestimate, d_u, as follows

$$d_e = \text{current distance travelled in path} + d(\text{last point, goal}),$$

where $d(x, y)$ represents the Euclidean distance from a point x to y. By theorem, if you analyze paths in order of a distance underestimate, A* will return the optimal path. With that stated, we can now detail technical outline of A*.

- Initialize a queue with just our start position in it.
- Pop off the path of the queue with the lowest distance underestimate.
- If the last point of the path is our goal, we're done. Otherwise, add a path to the queue for each neighbor of the last point in the popped path. Note that a neighbor cannot be a point that is a wall in the occupancy grid.
- Repeat step 3 until done.

As for what we used as the neighbors to a point in the occupancy grid, we used the eight points neighboring a central square as depicted in the figure below.

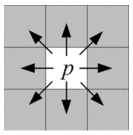


Figure 2: A diagram representing the neighbors to a given point.

2.2.3 A* Optimizations

This implementation of A* could still falter from analyzing multiple paths that go through some point, x, with the distances to x being longer in some paths than others. We wouldn't want to analyze the longer ones, for they're certainly longer than the optimal which would use the shorter path. Thus, we include a visited set as well and add this restriction to the addition of paths to the queue: we only add the path including a neighbor n if n hasn't been visited yet.

We also use the heapq data structure, for heapqs are able to find the lowest value for the distance underestimate much faster than other data structures. In terms of Big-Oh notation, a heapq is $O(\log n)$ vs $O(n)$, for a Python list, making the heapq structure a significantly better choice.

2.2.4 Graph Construction (RS)

Our implementation of A* will give us the shortest path, but when deployed onto our racecar, it might cut corners too tight that our racecar cannot handle, and we could potentially crash into a wall. Thus, we would want to give ourselves protection against that. Thus, we dilated the walls throughout the occupancy grid. Specifically, if a point has a neighbor that is a wall, it became a wall in the dilated grid. We then did then again so as to dilate by 2. Now, we could safely cut as many corners as we want in our planned path, for regardless of how tight we cut corners, we have that inbuilt cushion of size 2 to protect us from crashes.

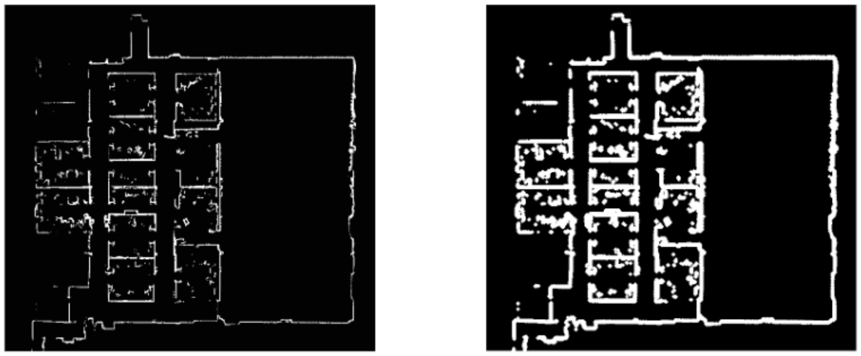


Figure 3: A diagram representing the original occupancy grid on the left and the dilated one on the right. White represents a wall.

2.2.5 Path Compression

Our algorithm for A* returns all points along the path, but that makes for a long path of minutely changing trajectories. This can be hard for our racecar to follow. Thus, we would like to have a shorter path encoding the same information that we could pass along. To do this, we start from our starting position, find the last point in our path that we can reach in a straight-line trajectory without hitting a wall, x, and we add x to a new path. We then repeat until we hit our goal position. As depicted in the figure below, this reduces a long path of over 30 points to one of just 4.

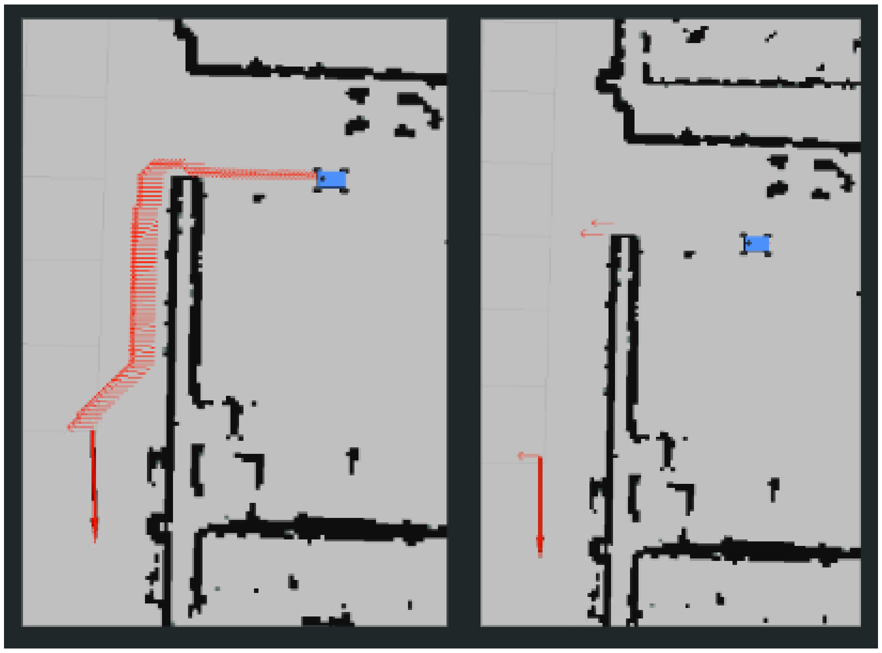


Figure 4: A diagram showing how we use path compression to encode the same path with many fewer points, which are depicted as arrows. Left depicts our original path, whereas right is our compressed path

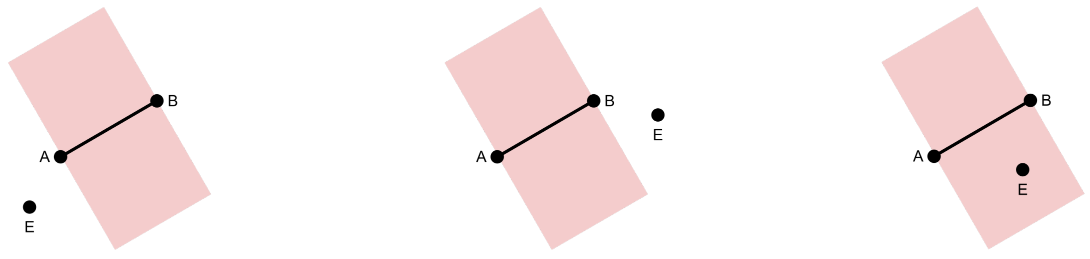
2.3 Path Following

Given the optimal path as a series of line segments, we implemented a path follower based on the pure pursuit algorithm. Our path follower can be modularized into three steps:

1. Find the closest line segment on the path.
2. Locate the intersection point with that segment using the lookahead distance.
3. Pursue the intersection point using Pure Pursuit.

2.3.1 Identifying The Nearest Line Segment (JS)

Our goal is to find the line segment that is closest to the car's current position. For every line segment $\overline{AB}$ on the optimal path, we want to find the shortest distance from the car's current position E to $\overline{AB}$. However, there are three distinct cases: 1) E is closest to A, 2) E is closest to B, and 3) E is within the range of $\overline{AB}$.



(a) E is closest to A (b) E is closest to B (c) E is within the range of $\overline{AB}$

Figure 5: E represents the car's current position and $\overline{AB}$ represents a line segment on the optimal path. The red area represents the range of $\overline{AB}$.

Case 1: E is closest to A
This can be confirmed by checking that:

$$\overrightarrow{AB} \cdot \overrightarrow{AE} < 0 \tag{1}$$

which means E lies in the opposite direction of $\overline{AB}$. Therefore, the nearest point from E is A, and the distance is given by $\left|\overrightarrow{EA}\right|$.

Case 2: E is closest to B
This can be confirmed by checking that:

$$\overrightarrow{AB} \cdot \overrightarrow{BE} > 0 \tag{2}$$

which means E lies in the same direction as $\overline{AB}$. Therefore, the nearest point from E is B, and the distance is given by $\left|\overrightarrow{EB}\right|$.

Case 3: E is within the range of $\overline{AB}$
If the dot product is 0, then E is perpendicular to $\overline{AB}$, and the distance between E and $\overline{AB}$ is given by:

$$\left|\overrightarrow{EF}\right| = \left|\frac{(\overrightarrow{AB} \times \overrightarrow{AE})}{\left|\overrightarrow{AB}\right|}\right| \tag{3}$$

Now, we have the distance from the car's current position to every line segment in the path, and we take the line segment that has the shortest distance. This closest segment was then used as the starting index for iterating through the rest of the segments when finding intersection points, as outlined in section 2.3.2.

2.3.2 Locating a Point on the Path to Pursue(OM + JS)

In order to determine a viable point on a given path for our racecar to pursue at any given time, we construct a circle around the car and find the intersection between this circle and the line segment. The circle is given a radius based on a chosen parameter l_d, the look-ahead distance. This parameter requires tuning based on the speed of the racecar and the environment it is traveling in. The selection of this parameter will be described in detail in section 2.3.3.

We can fully define any point on a line segment $\mathbf{p}_{line}$ by adding some fraction t of the vector between the start and end points to the start point.

$$\begin{gathered} \mathbf{v} = \mathbf{p}_{end} - \mathbf{p}_{start} \\ \mathbf{p}_{line} = \mathbf{p}_{start} + t \cdot \mathbf{v} \end{gathered} \tag{4}$$

We can define p_{car} as the coordinates of the racecar's current location. Any point x on the circle must be a distance l_d from p_{car}:

$$|\mathbf{x} - \mathbf{p}_{car}| = l_d \tag{5}$$

Therefore, the line intersects the circle if there is a point on the circle $\mathbf{x}$ such that $x = p_{line}$. Substituting p_{line} in for x in equation 5 and using the definition for p_{line} in equation 4, we can then write:

$$\begin{gathered} |\mathbf{p}_{line} - \mathbf{p}_{car}| = l_d \\ |\mathbf{p}_{start} + t\mathbf{v} - \mathbf{p}_{car}| = l_d \end{gathered} \tag{6}$$

We can then square both sides of the equation, and use the identity that $\mathbf{A}$, $|\mathbf{A}|^2 = \mathbf{A} \cdot \mathbf{A}$ for any vector $\mathbf{A}$ to achieve:

$$\begin{aligned} l_d^2 &= (\mathbf{p}_{start} + t\mathbf{v} - \mathbf{p}_{car}) \cdot (\mathbf{p}_{start} + t\mathbf{v} - \mathbf{p}_{car}) \\ \implies l_d^2 &= t^2(\mathbf{v} \cdot \mathbf{v}) + 2t(\mathbf{v} \cdot (\mathbf{p}_{start} - \mathbf{p}_{car})) + \mathbf{p}_{start} \cdot \mathbf{p}_{start} + \mathbf{p}_{car} \cdot \mathbf{p}_{car} - 2(\mathbf{p}_{start} \cdot \mathbf{p}_{end}) \end{aligned} \tag{7}$$

Moving r^2 to the lefthand side of the equation allows us to use the quadratic formula to solve for the two solutions for t, t_1 and t_2, that satisfy equation 7.

$$\begin{aligned} a &= \mathbf{v} \cdot \mathbf{v} \\ b &= 2\mathbf{v} \cdot (\mathbf{p}_{start} - \mathbf{p}_{car}) \\ c &= \mathbf{p}_{start} \cdot \mathbf{p}_{start} + \mathbf{p}_{car} \cdot \mathbf{p}_{car} - 2(\mathbf{p}_{start} \cdot \mathbf{p}_{end}) \\ t &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \end{aligned} \tag{8}$$

If the discriminant $\sqrt{b^2 - 4ac} < 0$, then there are no intersections with the line segment. Furthermore, we impose the restriction that if a solution $t_i \notin [0, 1]$ it must be rejected, as t is again a fraction of the line segment. We can then have up to two points that represent the intersections between the line segment and the racecar's look-ahead circle, which can be solved for by plugging t_1 and t_2 into equation 4.

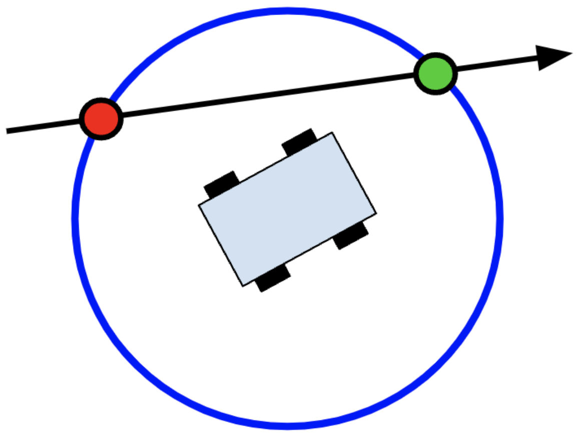


Figure 6: A diagram representing how a goal point is chosen in the case of two intersections with a single line segment on the path. The arrow represents the line segment. The green point is the chosen point, while the red point is the rejected point.

If there are two solutions t_1, t_2 corresponding to points $\mathbf{p}_{line,1}$, $\mathbf{p}_{line,2}$ for a given line segment, we need to choose which point we set as our intersection point to pursue. To do this, we decided to choose the point that minimizes the car's need to turn. This is equivalent to finding the point that is best aligned with the car's current orientation θ_{curr}. For each point, the inner product between the unit vector from the car's current position to the point and the unit vector of the car's orientation is found as:

$$\mathbf{r}_d = \mathbf{p}_{line} - \mathbf{p}_{car} \tag{9}$$

$$Q_{inner} = \hat{\mathbf{r}}_d \cdot \hat{\theta}_{curr} \tag{10}$$

We take the point that has the greater inner product Q_{inner}.

Now, we have one solution for each line segment on the path. However, it's still possible that there are multiple solutions across multiple line segments as shown in figure 7. In this case, we again need to choose which solution we set as our intersection point to pursue. We chose to pursue the point on the line segment that is further along the path, because we want to get closer to our trajectory end point. This is achieved by starting our search for the intersection point at the closest line segment, then iterating through the rest of the segments as given in the original path trajectory.

2.3.3 Path Following Using Pure Pursuit on Nearest Path Segment (OM)

Pure pursuit is a controls algorithm used to calculate the necessary steering angle of the front wheels of a ground vehicle in order to follow a semi-circular path towards a goal point. The intention behind a semi-circular path, versus one that more quickly reduces the angle from the path, is to maintain small changes in the steering angle rather than drastic ones. This should help the car with over-correcting past a desired path, but is ultimately dependent on the chosen l_d. Our implementation uses this algorithm on every time step in order to pursue a point on the path that is l_d away.

This goal point is the above-described intersection point between the look-ahead radius of the car and the path. In the case that there is no intersection, meaning the car is not close to any segment on the path, then the car will stay stationary. If there the radius only intersects with one segment on the line, then we can easily choose that point as the goal point to pursue. However, if multiple intersections occur with multiple segments on the path, as shown in figure 7, we will choose to follow the point that is further down the path to ensure forward progress.

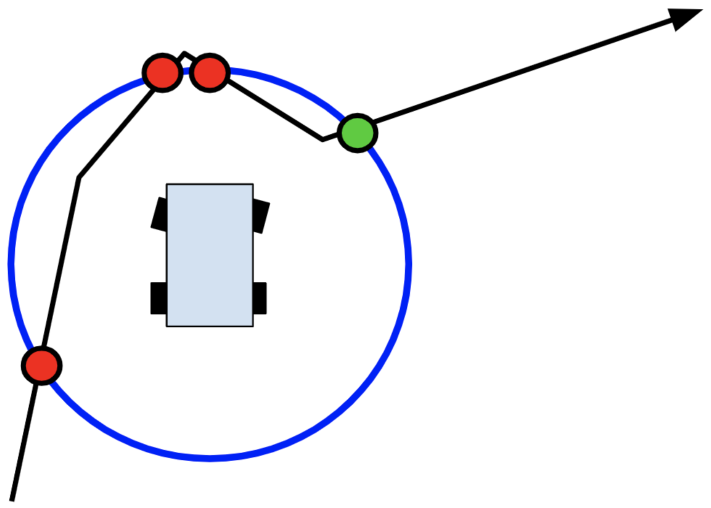


Figure 7: A diagram representing how a goal point is chosen in the case of intersections with multiple line segments on the path. The direction of the path is indicated by the arrow on the line. The chosen point, which is the one furthest down the path, is represented in green, while the rejected points are shown in red.

The algorithm uses equation 11 to calculate the racecar's steering angle, δ, to achieve a circular path to that point using the geometric values shown in figure 8.

$$\delta = \arctan \frac{2L \sin \alpha}{l_d} \tag{11}$$

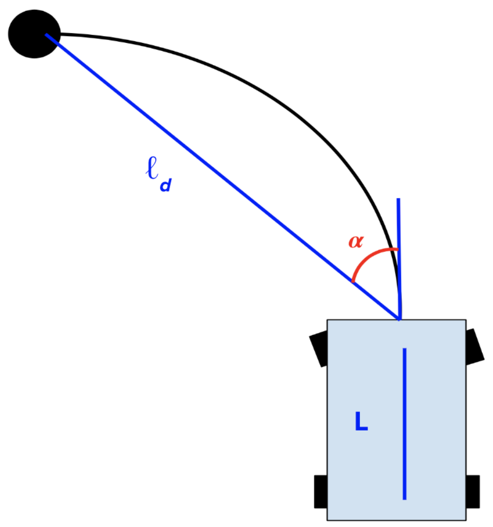


Figure 8: A simplified diagram visualizing the geometric values used in the calculation of δ. l_d is the chosen look-ahead distance, L is the wheel-base length of the racecar, and α is the angle between the racecar's current orientation and the line of shortest distance connecting the car to the point.

2.4 Deployment in Real Environment (OM)

Similar to the wall following algorithm from lab 3, we found that the physical racecar would frequently over-correct when attempting to pursue the desired path with the same parameters as simulation. In this case, the only tunable parameter for our implementation is the pursuit algorithm's look-ahead distance.

Given equation 11 from section 2.3.3, one can see that the steering angle δ is inversely related to the look-ahead distance l_d. This means that for a larger l_d, δ will be smaller, and therefore the car will take a longer pursuit route to a given point. Using this information and the results of brief qualitative testing, we deduced that we must increase l_d from .75 m to 1.4 m to reduce the likelihood of over-correcting past the planned path.

Furthermore, in order to ensure the safety of the racecar and reduce damage to its components when turning around corners and obstacles, we increased the expansion value of the occupancy grid in order to increase the distance between the path and the obstacles. This allows the racecar more room for error when navigating sharp turns.

3 Experimental Evaluation

3.1 Testing in Simulation (AY)

After synthesizing our modules together, we investigate its qualitative and quantitive performance in simulation. We find that we are able to navigate both long and complex paths successfully (Figure 11).

To evaluate quantitative performance, we examine four test scenarios in more detail: (a) a short straight path, (b) a single corner, (c) a complex route, and (d) a long corner. We find that not only is the planning time efficient, but our pure pursuit controls the car to be no more than 3 centimeters away from the planned path on average. These results are summarized in Table 1 and Figure 10, and each test case is displayed in Figure 9. The distance measurement is computed as the orthogonal distance to the closest line segment. We notice that peaks in the errors occur at corners or wall irregularities, which is to be expected. However, they diminish quickly which indicates that our control is able to adapt readily in these settings. Additionally, the low number of points used indicates that the path compression is working well to reduce the number of redundant points while increasing the smoothness in the trajectory.

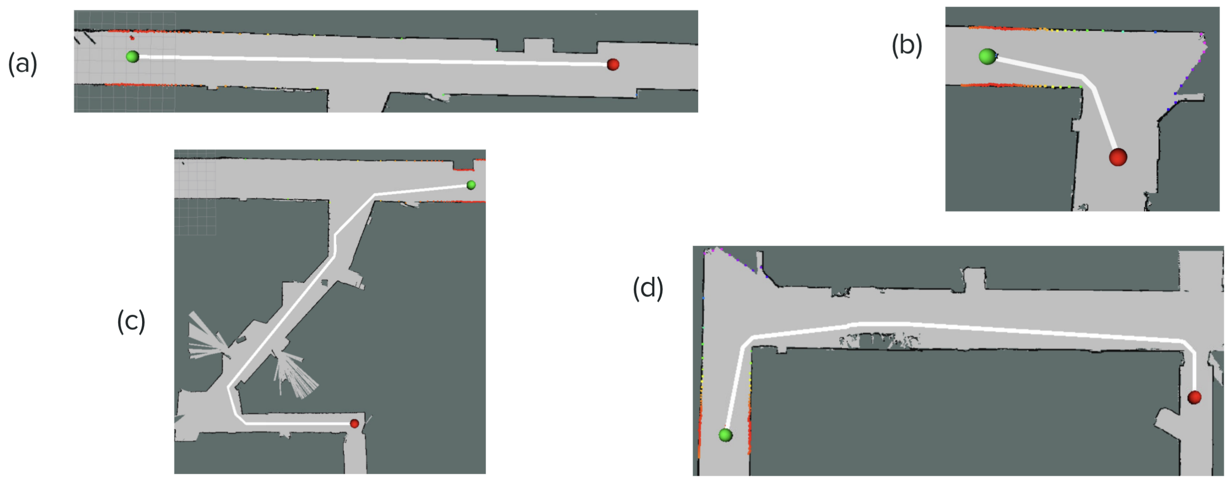


Figure 9: We examine four encompassing test scenarios for evaluating the quantitative performance of our pure pursuit module. These include (a) a short-straight path, (b) a single corner, (c) a double corner, and (d) a complex path.

Table 1: Planning Time for Various Path Scenarios

Test	Distance (m)	Num Points	**Plan Time (s)**
Short-Straight	5.95	2	**0.016**
Single Corner	11.12	11	**0.320**
Double Corner	50.86	21	**5.596**
Complex	54.43	9	**5.560**

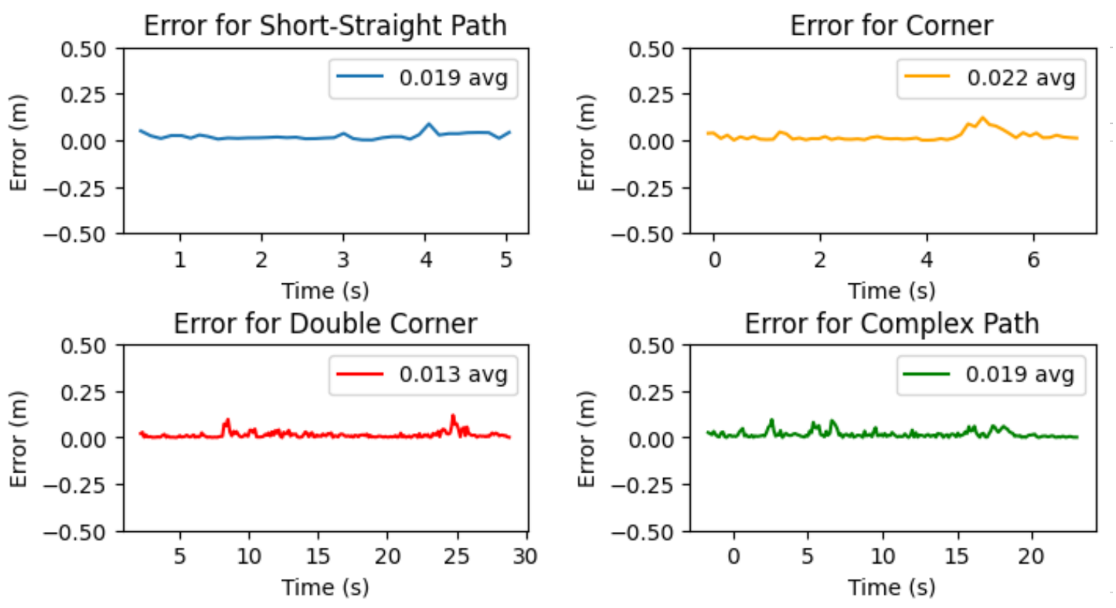


Figure 10: Distance error for the pure pursuit module given the planned path in simulation. Our implementation maintains an average error of around 2 centimeters.

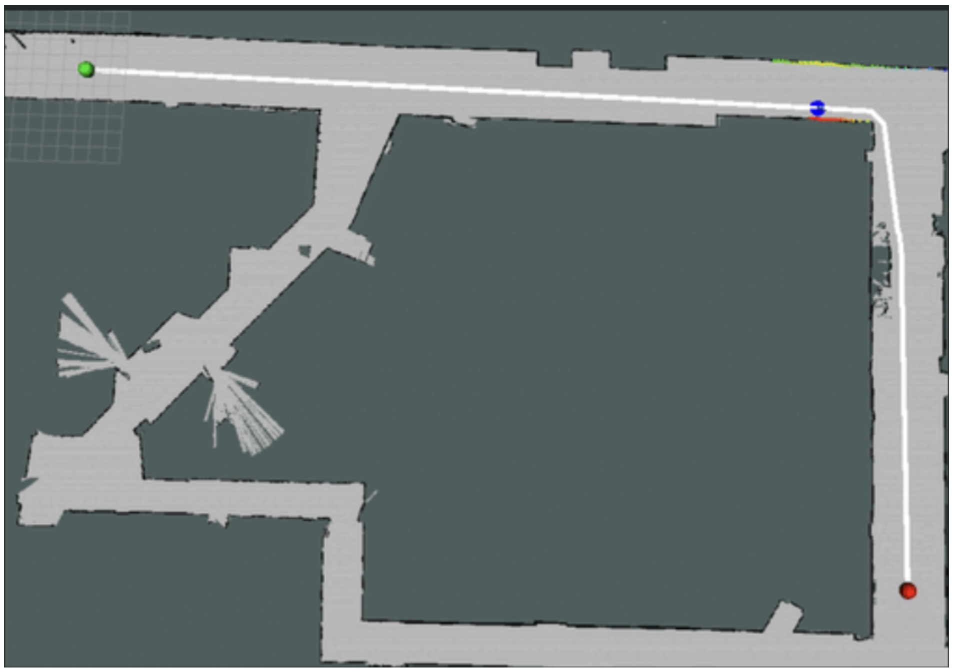
We noticed that a big factor playing into the accuracy of the pure pursuit was the performance of the localization module. With poor localization, especially in the real world where noise is more pronounced, the pure pursuit will have a certain amount of systematic error. Using the staff solution for localization (as opposed to our own) helped in both cases, although the effect was much greater in the real world and will be discussed in the following section.

3.2 Testing in Real Environment (OM)

The first step in validating the path planning and path following systems on the physical racecar was a qualitative evaluation of the both systems working in tandem. This entailed placing the racecar at a location in the physical environment and setting a corresponding estimate pose on the simulated map, then setting a goal pose on the map to generate a path using the path planning algorithm, and finally allowing the car to traverse this path using the pursuit algorithm. The results of this qualitative test can be seen in figure 12, in which the car is seen traversing the path both in the real world and in the corresponding simulated view. With the system validated qualitatively, the next step is to numerically test our implementation on the physical racecar. To do so, we needed to validate the trajectory follower algorithm exclusively, as path planning is still done entirely on the simulated map, and, therefore, its effectiveness should not change. This allowed us to channel the entirety of our focus into testing the racecar's path-following ability.

We drew upon the simulation test suite to construct a suite containing a straight path test and a single corner path test. By reusing tests that were performed in simulation—and doing so in equivalent locations on the basement map—we are able to eliminate confounding variables and accurately contrast the algorithm's performance in a real environment with that in simulation. The results of these tests can be seen in figure 13. We again tracked the estimated deviation of the car from the planned path.

For the straight path test, the difference in average error between simulation and the physical racecar is only 8 millimeters. However, despite the corner test on the racecar still having very low error at approximately 8 centimeters, this is a jump of around 6 centimeters up from the simulated corner test. This increase in error can likely be attributed to localization error, as localization is less accurate in a real environment, especially while turning a corner. Another factor could be the need for further tuning of the pursuit algorithm's


(a) Successful navigation around long corner

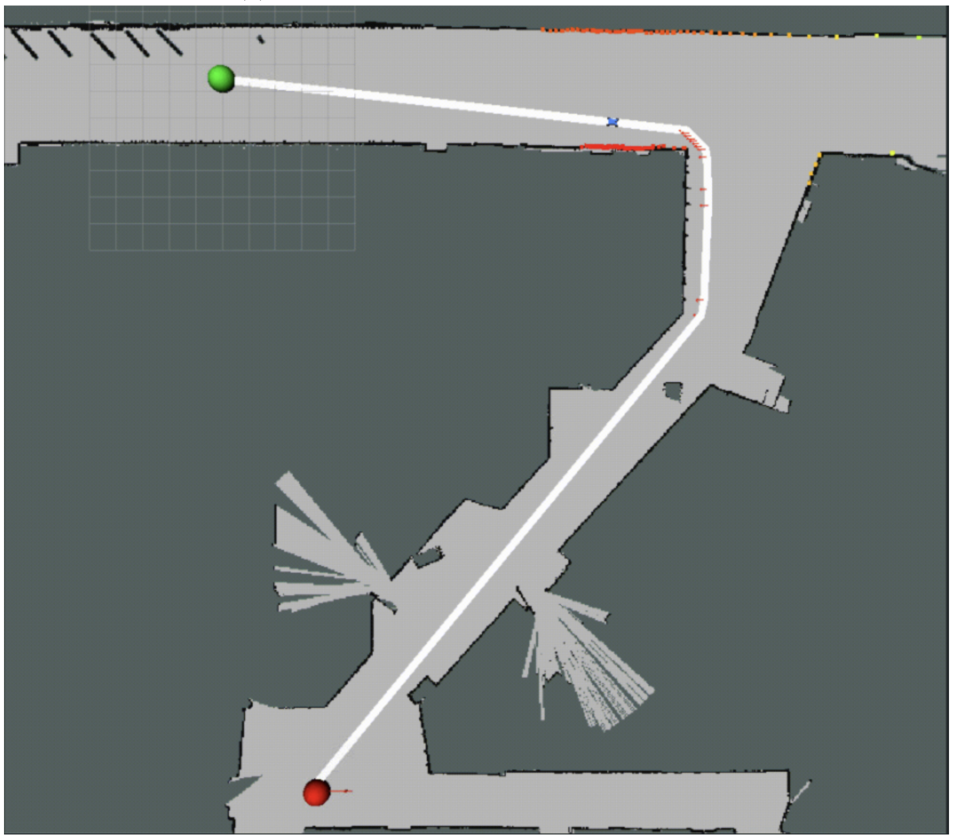
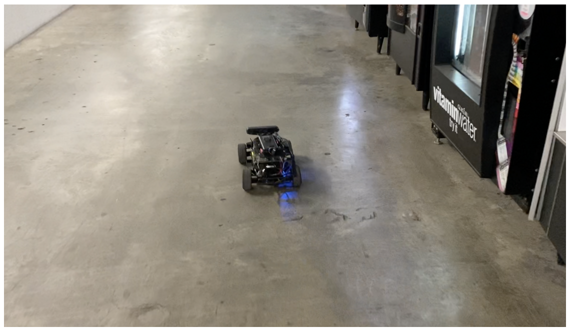

(b) Traversing complex region

Figure 11: We visualize the result of our synthesized path planning and pure pursuit algorithms in simulation. The racecar is able to safely and effectively navigate these challenging routes.


(a) Real life view of car traversing path in real environment

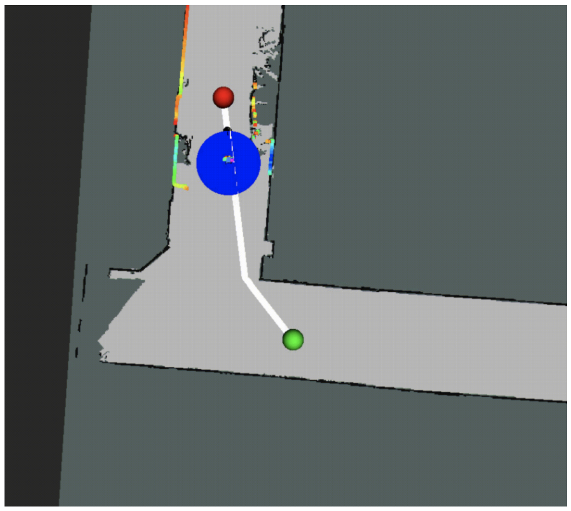

(b) Simulated visualization of car traversing path in real environment

Figure 12: The visualized result of our synthesized path planning and pure pursuit algorithms in a real environment for a path around a corner in the Building 32 basement. The car was able to plan the path and safely traverse the path from start to end with no collisions, indicating a qualitative success.

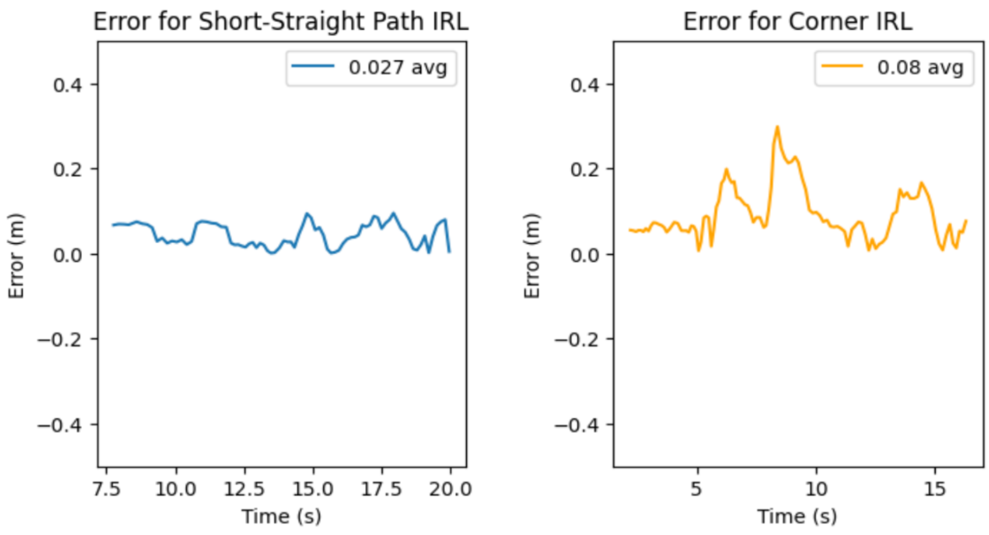


Figure 13: Two graphs visualizing the error between the car's estimated position using localization and planned path.

lookahead distance, as this may help reduce the spike in error as the car turns the corner.

3.3 Bonus: Fastest Implementation in Real World (SY)

Our team successfully demonstrated the real-world performance of our trajectory planning and tracking system, resulting in an impressive completion time of 23.55 seconds on a marked segment of the Stata basement loop. Below are photographs of our racecar at the start and end points of the race (Figure 14, Figure 15), and the trajectory (Figure 16) we gained in the race.

Figure 14: Racecar at the starting point of the race track.

Figure 15: Racecar at the finish line of the race track.

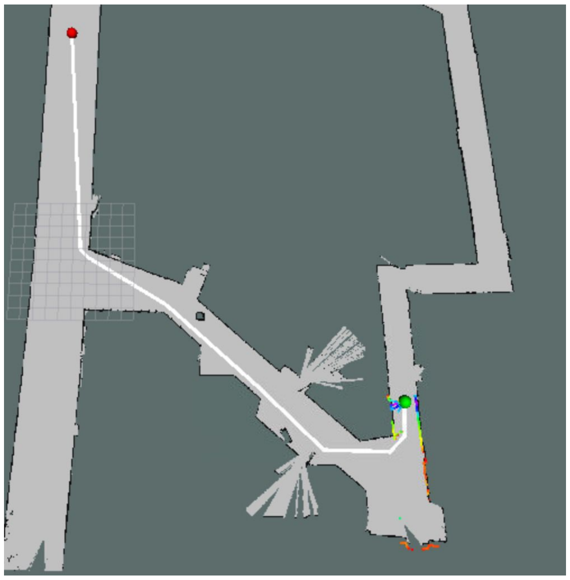


Figure 16: Racecar trajectory planned out in the RViZ from start to end.

This result highlights the effectiveness of our approach in trajectory planning and tracking. Our system was able to plan and execute trajectories efficiently, maintaining a high level of accuracy throughout the race. The top three teams with the fastest real-world implementations were awarded bonus points, and we are confident that our performance will place us among the top contenders.

3.4 Limitations (SY)

Although our implementation of the A* algorithm achieved satisfactory results, it has several limitations. We discuss these limitations and compare them to the advantages of sample-based planning methods in the following sections.

Unscanned areas introduce gaps that challenge planning. Dilation of obstacles helps avoid getting stuck in gaps, but determining the proper dilation amount is difficult. Insufficient dilation may still allow the racecar to enter gaps, while excessive dilation may cause the algorithm to overlook narrow passages, leading to inefficient paths or missed traversal opportunities.

The path planning using A* algorithm may sometimes result in sharp trajectories rather than smooth ones, which can pose challenges for the racecar's motion control. To address this issue, we can incorporate the racecar's dynamics into the planning process, taking into account its kinematic and dynamic constraints. By doing so, we can design paths with limited curvature, ensuring that the racecar follows a smoother and more efficient trajectory.

Another limitations of our A* algorithm is the inability to efficiently adapt to new start and end points. This is due to the algorithm's dependence on a fixed graph representation of the search space, which necessitates a complete recomputation whenever start and end points change. This can result in a significant increase in computation time, especially in large or complex environments. On the other hand, sample-based planning methods offer a more efficient solution by reusing the existing graph for searches, allowing for faster exploration and adaptation without the need for extensive resampling.

In addition to the aforementioned limitations, our search-based agorithm also struggles in dynamic environments where obstacles may appear or move unexpectedly. Sample-based planning methods, on the other hand, are more adept at handling these situations by incorporating the environment information into the sampling process. This advantage enables sample-based planners to update planned paths in real-time, providing a more robust navigation performance in changing environments.

4 Conclusion (MW)

We have successfully achieved our goal of creating a truly autonomous car by implementing both the planning and control phases of autonomous operation. The planning phase involved calculating the safest and shortest path to our destination using a search-based or sampling-based motion planning method on a known occupancy grid map. The control phase utilized the pure pursuit algorithm, which combines the particle filter and pure pursuit control to enable our car to follow the predefined trajectory on the map. Through our implementation, we have demonstrated the ability of our car to navigate autonomously and precisely follow the planned path, ensuring a smooth and secure ride to our destination.

5 Lessons Learned

[**AY**] On the technical front, I learned about A* search and that often, it is best to try out simple methods first. As mentioned in our briefing, we started out with an approach for computing neighbors that was great in theory, but actually applying it caused hours of headache from attempting to debug it. In the end, we resorted to a simpler pixel-based implementation that was theoretically sower, but ended up giving great results nonetheless. In the future, I would implement the simpler method first, and then follow up by modifying it. For CI, I realized that although modular work is better, it is a mistake to overlook the importance of helping everyone understand each step of the pipeline; those that worked on planning should also have a good understanding of the pure pursuit, and vice versa. That way, during integration, simple issues can be addressed quickly instead of being blocked.

[JS] Technical: I learned more about how pure pursuit works. From a big picture perspective, pure pursuit seemed much simpler than PID control: drive towards a point. However, implementation turned out to be more complicated than I expected. Finding the exact point to follow required some mathematical gymnastics using vectors and dot products. In addition, there were a couple of different edge cases we had to handle, such as two intersection points in a single line segment and multiple intersection points across multiple line segments. Communication: I learned that modularization is a really powerful tool. I conceptually understand how we generated a trajectory using A*, but I don't know much of the technical details. However, I don't have to. In order to implement pure pursuit, all I needed to know was the output of A*: a path as a series of line segments. This was enough to finish pure pursuit, and synthesize our work as a final step like magic.

[RS] From a technical perspective, I learned about new path planning algorithms and learned the respective pros and cons to each. I also learned how some of these algorithms are certainly easier to implement than others, for we tried to implement an algorithm using the ideas behind a visibility graph for more efficient path planning, but that process was riddled with bugs that we couldn't solve. This was in contrast to A*, which was quite simple to implement.
This leads well into what I learned on the communications side: Start with a simple idea and build out complexity from there. I knew that the visibility graph idea would be more efficient than A*, but I did not anticipate the difficulty with implementation. If I had instead started with A* first and then worked my way up to the other idea, I don't think I would have ran into as many errors, for I'd have already encountered similar errors in the simpler case of A*. Thus, moving forward, I'll start with simpler ideas and iteratively increase the complexity for my efficiency. That way, it'll be clear where things are going wrong, and I'll more easily be able to debug.

[SY] Technically, I gained a deeper understanding of search-based planning algorithms, including their implementation, optimization, and various techniques to enhance their performance. Furthermore, we developed the ability to combine different steps of various implementations and integrate them into a cohesive system, demonstrating practical engineering prowess. Another significant aspect of my learning experience is exploring and comparing the characteristics, performances, and weaknesses of both search-based and sample-based planning algorithms. This enabling me to make informed decisions when selecting an algorithm for a particular task.
At CI aspect, I recognized the importance of efficient parallel task completion by dividing responsibilities among team members. This approach ensured that each team member could focus on their area of expertise, resulting in a more streamlined workflow and timely project completion. In addition, I learned to balance teamwork with individual preparation. By allocating time for self-preparation, we can be better equipped to contribute effectively to the team, the together teamwork like co-programming also will be more effective.

[MW] Technical: This lab was particularly enjoyable because it required us to apply our prior knowledge of localization to achieve path planning. It was fascinating to see how our previous coding skills could be utilized to accomplish this task successfully. I was particularly proud of our performance in the pure pursuit section of the lab, as we were able to get our individual sections functioning well early on.
Communication: However, we could have improved by initiating integration testing a little earlier in the process. During the lab, I also learned more about the A* search algorithm and other strategies to optimize search algorithms. Witnessing the car drive autonomously was incredibly rewarding and left me feeling fulfilled by the experience.
[OM] On the technical side, I was able to practically expose myself to path-planning algorithms for the first time and research the tradeoffs between sample-based planning and search-based planning. Furthermore, I expanded my knowledge on control algorithms from solely PID control to now include pure pursuit, which has a more geometric, grounded derivation.
In terms of communications lessons, I realized that sometimes there is a tradeoff between the efficiency

parallelized worflow and lack of understanding. I focused almost entirely on the controls portion of this lab, and was therefore left a bit behind in regards to the planning portion. Luckily, through asking questions and investigating the code, I was able to catch up. However, this was an unnecessary time sink that should be avoidable in the future. I should make sure to stay more involved in other portions of the lab, even if I'm not directly contributing to the code.