Lab 6 Report: Path Planning

Team 7

Bradley Albright
Ethan Ardolino
Jiahai Feng
Tahmid Jamal
Tian Lin

Robotics Science and Systems

April 16, 2022

1 Introduction [Tian]

Previously, we created a package that localizes the racecar using its pose (rotation and position data) and LiDAR data. In this iteration of the lab, we are using the racecar's localization information to aid in implementing a path planning and following mechanism.

We programmed two algorithms for path planning, a search-based and a sampling-based implementation. We integrated the search-based method A* in simulation and then on our racecar. We also implemented an RRT* algorithm. These two algorithms were the basis for our path planning.

To ensure the racecar follows the path created by the two algorithms, we also needed to create a pure pursuit controller that ensures the racecar travels on that path. The controller locates the nearest point from the racecar to the path and uses the point to determine the appropriate steering angle to get to the path. We tested the controller on the simulated racecar before transferring the controller to an actual racecar.

The localization data allows us to determine where the robot is on the path and the combination of our path planning and pure pursuit controller ensures our racecar can find its way from the starting destination to the endpoint.

2 Technical Approach

Given a starting location and an ending location the robot must plan a path in order to traverse the distance in between while avoiding obstacles. There are many algorithms for path planning that each have trade offs in terms of optimality and computation time on different graphs. We will take a look a two different algorithms, A* and RRT*, to compare their effectiveness in planning a path for the robot. The robot then takes these paths and accordingly traverses them by making use of a pure pursuit controller. These combined steps allows our robot to navigate to any connected area on a known map.

2.1 Transforming the Occupancy Grid into a searchable Graph

In order to determine which points in the real world are obstacles and which are free space that the robot can drive through, our algorithms make use of the provided occupancy map of the stata basement. This gives us a grid of coordinates that correspond to the probability that a space is occupied.

2.1.1 Transforms between the Occupancy Grid Space and Map Space [Bradley]

Points on the map must be translated, rotated, and scaled into the occupancy grid space for us to know if the space is actually occupied. A reverse transform is also needed to determine where the path found in the grid is in the real world. The following equations depict these transforms:

- Real World Point to Occupancy Grid

$$p_{uv} = (p_{xy} - o)R^{-1}/(s) \tag{1}$$

- Occupancy Grid to Real World Point

$$p_{xy} = (p_{uv} * s)\dot{R} + o \tag{2}$$

Where

- o: Origin of the map corresponding to (0,0) of the occupancy grid
- R: Rotation matrix from real coordinates to occupancy grid
- s: Real world scaling factor between grid points.

To decrease computation time the grid is subsampled by taking every third column and row decreasing the size by a factor of 9 which subsequently increases the scaling between the occupancy grid and the map coordinates by a factor of three.

2.1.2 Dilating the Occupancy Grid [Bradley]

In order to find the shortest path between two points our algorithms will cut corners and drive closer to walls than is safe. Taking into account that our robot may not drive directly on the path provided, planning a path close to walls will inevitably cause a crash.
To avoid this situation. The width of the walls are increased through dilation depicted in figure 4. A visualization of the effect on the occupancy grid is shown in figure 4.

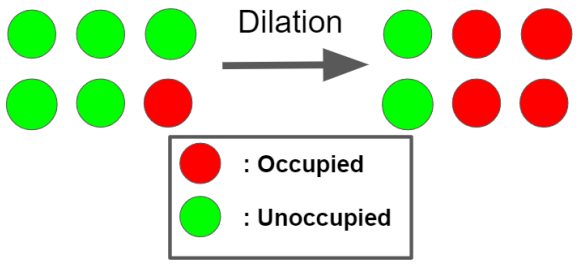


Figure 1: The neighboring unoccupied nodes which are adjacent to occupied nodes are flipped to occupied after dilation.

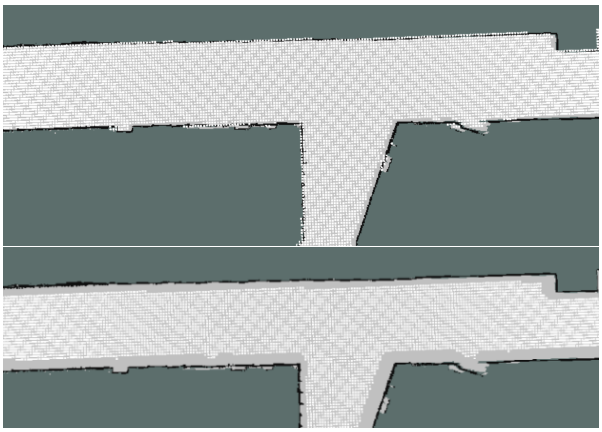


Figure 2: The upper picture shows the undilated image with the white squares showing all unoccupied spaces. The lower image shows the dilated image where more space has been given to the walls on all side to avoid collisions.

2.2 A* [Bradley]

Given a start point, end point, and a graph of nodes that connect those two points the A* algorithm finds a minimum cost path of nodes between the start and end points. Similar to other search algorithms, like breadth first search, the A* starts at the start node and gradually explores the grid until it finds a path to the end node. Unlike breadth first search A* explores the most promising nodes first by the concept of heuristics.

A heuristic is a estimate of how far away the end node is from the current node. The choice of a good heuristic can vastly decrease the amount of time it takes for A* to find a path to the end node. In this situation we made use of a heuristic which uses the straight line distance from the current node to the end node despite any obstacles in the way. The algorithm maintains a queue of nodes that it will explore, initiated with only the start node. A* 'pops' a node from the queue to explore if it has the lowest combination of the minimum distance to the node and the heuristic to the end node. The algorithm finishes when the end node is popped from the queue. As long as the heuristic never overestimates the actual distance from the current node to the end node the algorithm guarantees an optimal solution since any sub-optimal solution will never be popped from the queue before the path containing the optimal solution.

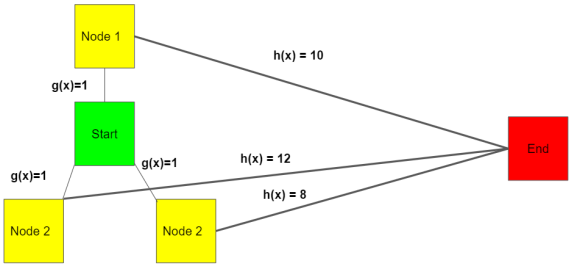


Figure 3: The picture depicts an incomplete graph with three intermediate nodes connected to the start node and a heuristic to the end point. The A* algorithm will explore node 2 first even though they are all the same distance away from the start point because that point is closer to the end goal than all the other nodes.

2.2.1 Computation time vs distance

To evaluate the computation time of the A* algorithm the search was timed with different distances between the start and end node. For the sake of consistency all points were considered in a straight line since the run time changes drastically based on the effect. Table 1 shows these values: The A* algorithms guarantees

Distance(m)	Computation Time(s)
10.949	0.248
25.03	0.392
68.77	0.745
80.412	1.0755

Table 1: Distance between the start and end points versus how long the A* algorithm took to compute the optimal path.

optimally but can take longer than needed when the graph nodes are particularly dense. The run time is also not particularly tunable unless we continue to sub-sample our graph which is not always possible. To find possible solutions to these problems we also investigated the RRT* algorithm.

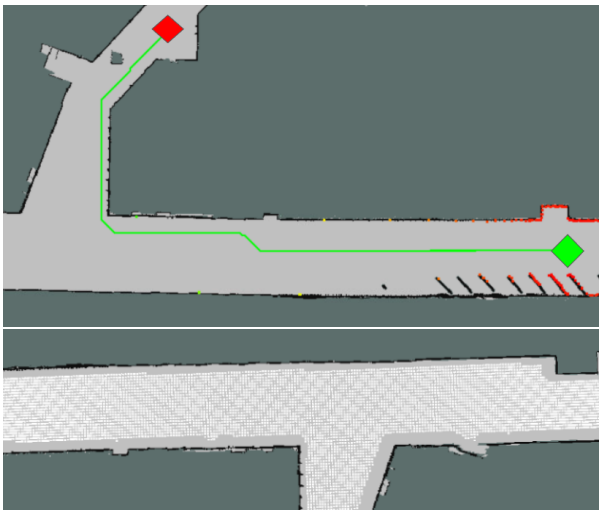


Figure 4: The green path connects points along the trajectory laid out on by the A* algorithm outlining a minimum distance path between the green start point and the red end point.

2.3 RRT* (Tahmid)

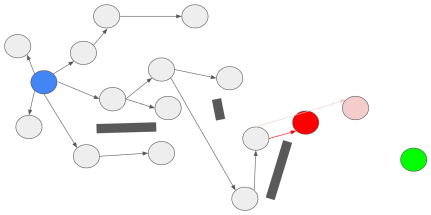
The RRT* algorithm is a search algorithm that builds a tree that fills the space of possible locations we can reach, efficiently. This tree is referred to as a Rapidly-Exploring Random Tree. This algorithm guarantees that given enough time, we will eventually reach the optimal or lowest cost path to the target. However, even if the algorithm has not converged to the optimal path, the path it returns will usually satisfice.

The RRT* begins by initializing the tree with a node representative of the starting location. At a given time, every path on this tree will represent our understanding of what the lowest cost path to the goal location (represented by some node) is. The following outlines the essential steps of the RRT* algorithm.

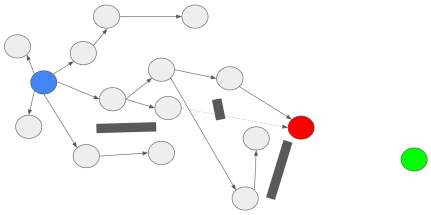
1. Uniformly sample a random location from the map with probability p. With probability $1 - p$, sample the goal location
2. Identify the closest node on the tree to the sampled location. If the sampled location is greater than R_{grow} (our branching distance), then decrease the distance such that the we have a new sampled location that is R_{grow} in the same direction.
3. Check if the sampled location has an obstacle in between it and the nearest node. If it does, return to 1.)
4. Create a new node for the sampled location and connect it to the node that would result in the lowest cost path to this new node.
5. Rewire or relax all edges for nodes within R_{wire} of the new node. This means if the the path through the new node to another node within R_{wire} has lower cost than what we believe to to be the lowest cost, we will change the paths and costs to reflect this new information.
6. Repeat $1. - 5.$ until we sample a node within ϵ of the goal node and find a collision free path. If we don't, then return the path that reaches closest to the goal.

The RRT* star algorithm is efficient in that it does not enumerate the entire state space and can find a satisfactory path relatively fast, based on the parameters $p, R_{\text{grow}}, R_{\text{wire}}$, and ϵ. However, it does perform a lot more costly operations such as floating point operations because it calculates distances and finding the minimum distance and cost to arbitrary locations.

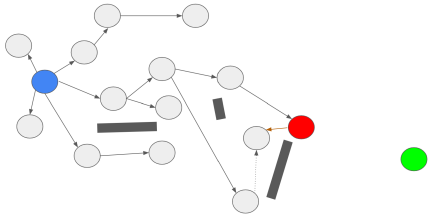
The RRT* implemented for this lab works and shows very near optimal paths in sufficient time for user created tests. However, due to issues with implementing it to the rest of path_planning.py file, we have yet to implement the RRT* algorithm on the robot. However, we expect to do so soon as the issue is likely resulting from a small misunderstanding.
An interesting side note is the obstacle detection algorithm used. For the purposes of this lab, we used a naive algorithm which paired together 2 linspaces between the starting pairs of x and y coordinates and rounded them to find a discretized line between 2 points. It then checked whether any of the points on this line were obstacles. This algorithm could be improved by ignoring floating point operations through use of Bresenham's line algorithm. It could also be made a lot faster by using raycasting for obstacle detection.



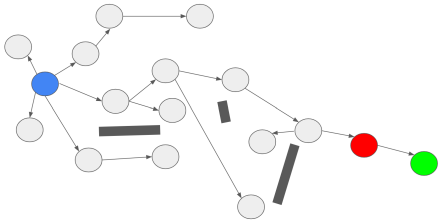
(a) Sample a random node and bring it near an existing node



(b) Identify the collision free path to the node that would have the lowest cost



(c) Relax all edges to vertices within a certain radius of the new node



(d) Repeat the process until we find a node sufficiently close to the goal

Figure 5:
fig:rrt

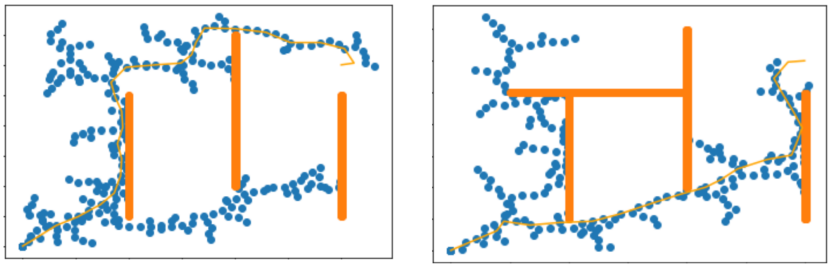


Figure 6: A quick look at a few tests that the RRT* passed successfully. The optimal path would be one that touch corners of the obstacles and the RRT* does almost exactly that.

2.4 Smoothing Trajectory Paths (Ethan)

After computing our planned path, there is one underlying issue with the trajectories achieved by both of our algorithms– they do not consider the driving mechanics of the racecar. Both approaches use a euclidean distance heuristic, so their paths is a large,linear piece-wise function. To combat this, we modified the trajectories with a smoothing algorithm. We first tried by forming spline curves.

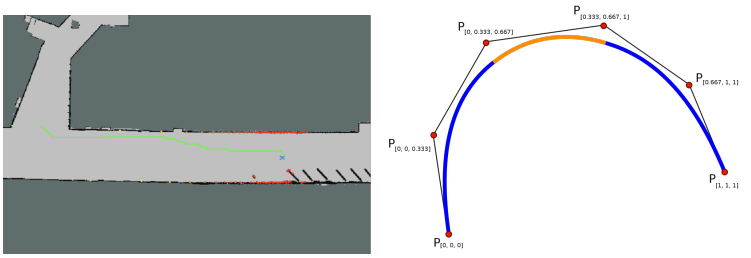


Figure 7: A visualization of our planned path from our A* algorithm (right). To the left is an example of a spline curve (a variation of a cubic interpolation). The blue and yellow lines represent a combination of three spline curves that are being fitted to the six datapoints.

Spline curves are a method of interpolation that generally use cubic functions. The cost function used to determine the curve utilizes a set of weights that allow trading off between smoothness and closeness to the data points.

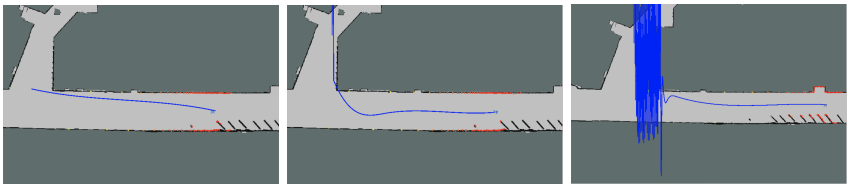


Figure 8: Here are the results of our spline curve implementation. As more complex trajectories were introduced, this method of smoothing began to perform poorly.

Although initially promising, smoothing via spline curves ended up being insufficient. As can be seen in the left picture, it successfully smoothed the path when given a basic trajectory. However, when the computed paths became more intricate, the smoothing computed a path through the wall. Furthermore, when a path was not defined by a function, the interpolation completely broke down and resulted in huge oscillations (as can be seen on the right).
We then adjusted to an algorithm I will denote "iterative" smoothing. Iterative smoothing used a gradient-descent like approach, continually updating the values of a data point until the overall $\Delta position$ was below a threshold value of $1 * 10^{-6}$. The cost function to "iteratively" smooth was weighted on the difference between a point's original coordinates as well as on the next and previous data points.
The alpha and beta values were eventually tuned to $\alpha = 0.1, \beta = 0.8$, signifying the smoothed trajectory was more focused on a point being close to it's neighbor rather than its initially computed coordinates.

$$p_i = p_i + \alpha\,(p_{orig} - p_i) + \beta\,[\,(p_{prev} - p_i) + (p_{next} - p_i)\,]$$

Figure 9: The cost function used in "iterative" smoothing. For each ith data point: p_orig = the original coordinates ; p_i = the current coordinates, p_prev = the original coordinates of the previous point, and p_next = the original coordinates of the next point.

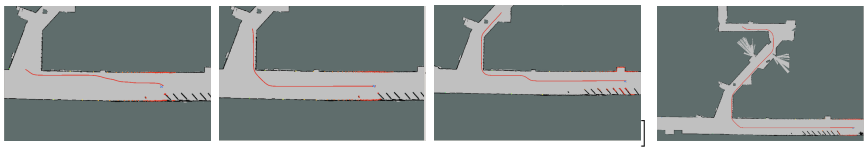


Figure 10: The results from the "iterative" smoothing method. As compared to Figure 5, in all situations the trajectory is both smooth and a valid path. Even in an extremely long trajectory (right), this algorithm proved effective.
As shown above, this new solution successfully smoothed the trajectories from our previously computed shortest paths.

2.5 Pure Pursuit (Jiahai)

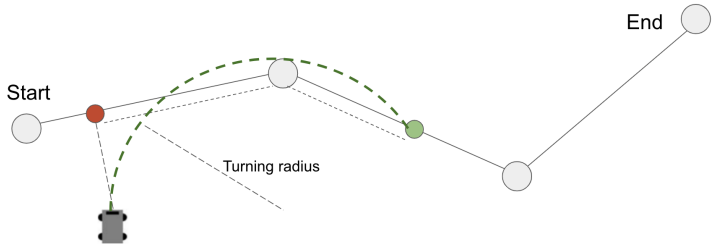


Figure 11: Schematic showing how pure pursuit works. **1)** The closest point on the trajectory to the robot is identified. (Red point). **2)** Traverse a distance equal to the lookahead distance forward along the trajectory to obtain the target waypoint. (Green point). **3)** Find the unique circle that is tangent to the car and passes through the target waypoint. **4)** The radius of the circle is the turning radius, and dividing the wheelbase length by the turning radius gives the steering angle.

With the trajectory obtained from the path planner, we implement a simple pure pursuit algorithm that follows this trajectory. First, we localize the robot using the particle filter that we built for the previous lab, by subscribing to the relevant topic. Then, having localized the robot relative to the trajectory, we perform the following steps (Fig. 11)

1. Identify the closest point on the trajectory to the robot
2. Traverse a distance along the trajectory equal to the lookahead distance. This point is the target waypoint.
3. Find the unique circle tangent to the car and passing through the target waypoint
4. Find the radius of the circle. This radius is the turning radius. Divide the wheelbase length by the turning radius to get the steering angle in radians.

The final step is a well-known result. In practice, we also exploited the wheelbase length to tune how tightly the robot makes turns. Increasing the value of the wheelbase length used in the calculation makes turns wider and hence less likely to cut corners.
The whole process (steps 1 - 4) happens at a rate of 10Hz.

3 Experimental Evaluation

This lab contains several components working in tandem. First, we evaluated the A* path planner and the pure pursuit trajectory follower independently, and then we evaluate the integrated end-to-end performance.

3.1 A* path planner (Jiahai)

The A* planner is tasked to find a path between two points in the Stata basement map (Fig. 12). The planner finds a path that is reasonably close to the staff solution, is consistently a safe distance from obstacles, and does not pass through any walls.

3.2 Pure pursuit follower (Jiahai)

In the same Stata basement map, a fixed trajectory is given this time to the pure pursuit follower (Fig. 13). The ground truth racecar trajectory (in red) and the estimated particle filter trajectory (in blue) both closely mirror the given target trajectory (in green) for the first roughly three quarters of the trajectory. However, in the bottom left corner of the map, the particle filter trajectory diverges wildly from the ground truth trajectory, indicating a catastrophic failure in the particle filter localization.

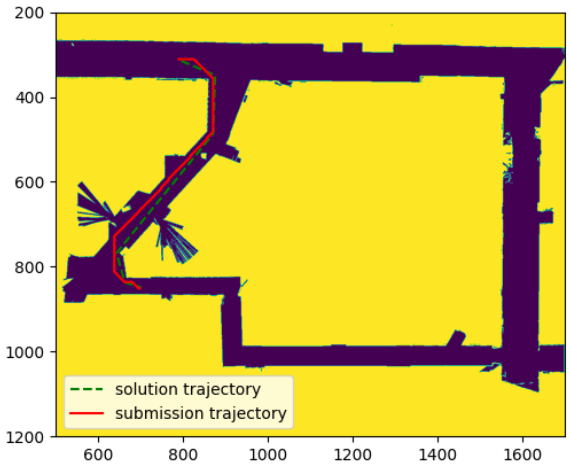


Figure 12: Path that our implementation produced (red) and path that the staff solution produced (green) when tasked to find a path between two points on the map. Our trajectory is reasonably close to the staff trajectory, and does not pass through any occupied points.

We are unable to ascertain its cause. Switching the particle filter solution to the staff solution did not eliminate this problem. The ground truth path also does not enter any occupied area before the massive divergence began. We are also unable to replicate this issue in our own simulations.
Nonetheless, the pure pursuit algorithm works robustly, since the estimated path, which is what the pure pursuit algorithm sees, still follows the given path. This gives us sufficient confidence to move on with the lab.

3.3 Integrated test (Jiahai)

In this test, we combine the two previous tests. We provide the path planner with the start and end point, which produces a target trajectory that is passed to the pure pursuit follower, and the resultant ground truth trajectory of the racecar is evaluated (Fig. 14). Our trajectory follows the solution trajectory for 97.42% of the path.

4 Conclusion [Tian]

At the conclusion of the lab, we path planned with two algorithms A*, a search-based method, and RRT*, a sampling-based method. After ensuring we can determine the best path from the starting point to the endpoint, we created

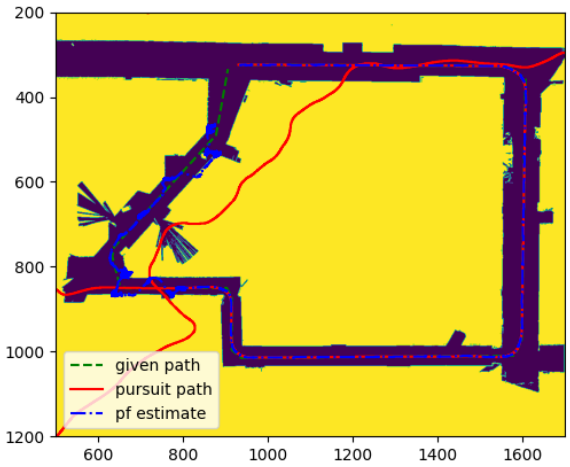


Figure 13: Pure pursuit algorithm when given a fixed trajectory to follow (in green). The blue trajectory is the estimated trajectory from particle filter, and red is the ground truth racecar trajectory. This reveals relatively good performance, but the particle filter fails catastrophically at the bottom left corner.

a controller to allow the racecar to follow the planned path. The pure pursuit controller was tested in both simulation and the physical racecar. With these steps, the robot can successfully travel from its current starting point in the environment to another chosen point in the environment. However, this environment/map has to be provided to the robot. One caveat is we have to manually set the destination in our program since our visualization software is often unreliable in transmitting information on our desired endpoint to the robot.

We successfully completed this phase of the lab. In our next phase, we will combine everything we created in all the labs thus far to react to challenges on the Johnson race track, mini-city, and rainbow road. We may use our path planning and pure pursuit controller to help us accomplish these challenges. However, the challenges deal more with reacting to the environment so it is unclear if we will need to plan a path.

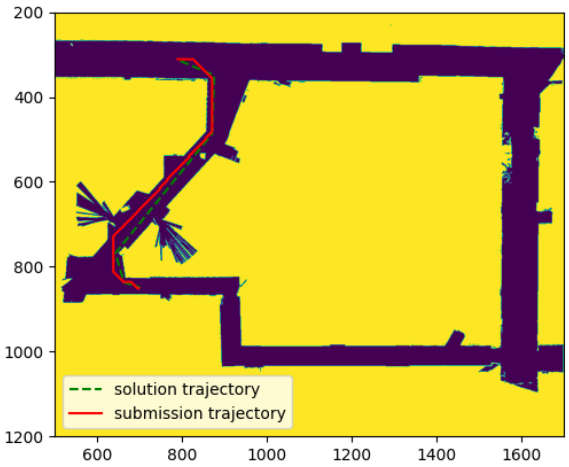


Figure 14: The integrated test with both path planning and trajectory following. In green is the solution trajectory, and in red is our ground truth trajectory, which tracks the solution trajectory closely.

5 Lessons Learned

Presents individually authored self-reflections on technical, communication, and collaboration lessons you have learned in the course of this lab.

5.1 Bradley's Reflection

Technically I focused on creating visual representations of my code for this lab which paid dividends in confirming the correctness of transforms and debugging errors in my code. When transferring to the robot RVIZ was very buggy and we eventually had to abandon the visualization instead of continuing to waste time trying to get it to work. Therefore it taught me the lesson that while visualizations are good, other tests such as unit tests are necessary for debugging when making changes to robotics code.

In terms of Communication I could feel myself getting a little terse with what I was saying to my teammates both near the deadline of getting the technical side done and getting our presentation done. I believe that I can be a more effective communicator by relaxing more in those situations. I also recognized that I should be giving more feedback to my teammates about their parts in the presentation. Ethan asked me for some advice and I realized that I zoned out during his portion while we were practicing. I realized that that practice

period is a prime time to get invaluable notes from teammates.

Collaboratively I realized that I jumped in too quickly to the technical side of the work without thinking enough or talking enough about what format we'd like the input and output of our functions to be. Since I took too large a jump into coding too early I had to rewrite massive portions of my code multiple times and even then using some of my functions were unintuitive. This lead to some setbacks in terms of testing Tahmid's RRT* code that I could've avoided with more communication and documentation.

5.2 Ethan's Reflection

In this lab, my technical takeaways were the importance of having a deep understanding of the math behind the algorithms you implement. I attempted to implement spline curves without this foundation, and upon later inspection this method was not compatible with the problem at hand.
With regards to communication, I felt that my briefing and report had a large focus on the overall "journey" from start to finish, not solely what ended up working in the end. Also, I feel my slide titles and captions were the most effective in lab 6.
In terms of collaboration, we took what we had said in the past about working together and did so excellently this week. Although work was delegated, we had a much better idea of others' sections so that it came together seamlessly.

5.3 Jiahai's Reflection

My technical takeaways are on learning to work with systems that I do not understand fully. I cannot always understand systems fully: in the case of the autograder, the internals are blackboxed to me, and in the case of RVIZ, the ROS remote set up we have in the class simply did not allow RVIZ to work properly with the racecar. In both situations I was blind to critical aspects of the system that would have let me troubleshoot. I learnt that it was important to consider several hypotheses and work on them concurrently instead of jumping to conclusions.
In terms of communication and collaboration, I realize that sometimes I get caught up with solving technical problems, and I do not think about how to help my teammates become more effective, by identifying any blockers they have or by providing context for parts of the lab that they might not be familiar with.

5.4 Tahmid's Reflection

A technical takeaway I've taken would be implementing psuedo-code from a research paper. I was mostly reading the paper on RRT* that was provided to us by the lab, with the help of a few online videos to simplify the process. However, a lot of details were lost in the abstraction and simplification that the

online videos provided, which lead to my initial versions of RRT* being actually RRT as the rewiring was not being done correctly; this took up a lot of my time which I could have used to address a future issue. However, a key contributor to my problems with implementing these methods was the fact that I would try to optimize my implementation as I was writing it, rather than writing a poorly optimized version and improving it. I think this process actually gave me a better understanding of the situations and settings where RRT* actually prevails as a superior search algorithm as I understood which areas I simply could not optimize as much, such as obstacle collision.
My communication for this lab could have used a lot of improvement. I struggled with implementing my final working and optimized RRT* algorithm with Bradley's template for path_planning.py. I assumed it would simply involve me plugging in my RRT* class. However, I ran into many issues where I would be detecting obstacles everywhere, and Bradley ran into this issue too. It has to do something with how the graph being read by the planner is different from what we think it is. If I worked this through with Bradley, we would have been able to implement the RRT* for this lab.

5.5 Tian's Reflection

My main technical takeaway is to read over the specifications of the lab closely. When I was working to debug with Bradley, I realized that I actually wasn't very familiar with which topics the subscribers and publishers were supposed to connect to which caused a lot of naming errors.

My main communication and collaboration takeaway are to get involved early during the task delegation so that I can have a clear goal to work towards for the rest of the lab. I felt like I was picking up pieces of things to do here and there because I wasn't clear on what my task is. In our final project, I will try clear up my goals in the beginning.