Lab 6 Report: Path Planning

Team 17

Devin McCabe
Anish Ravichandran
Arnold Su
Anirudh Valiveru

6.4200 - Robotics: Science and Systems

April 27, 2023

1 Introduction - Arnold Su

Motivates and contextualizes this lab's goals (i.e., identifies **what** you have designed in this lab and **how** that fits among the other RSS labs or how it contributes to developing an autonomous system). Presents an overview of the **purpose** and **specifications** of this lab. Provides a short and informal summary of the **technical problem** and introduces a bird's-eye view of your technical solution.

The main objectives of this lab is three-fold: path-planning, path-following, and integration. All three modules are centered around trajectories, which we will define as a path with start and end points. The goal of path-planning is to be able to produce a trajectory that avoids any obstacles on a given map for any start and end points. Path-following, on the other hand, should be able to take any trajectory and produce driving commands such that the robot will follow that trajectory. The final objective is to integrate path-planning and path-following. This will abstract away the produced trajectory, and so as a final result, when the robot is given a map and any two points signifying the start (initial pose) and end (goal point), the car will move to the goal point while avoiding any obstacles on the map.

For path-planning, we present two different algorithms: A* and rapidly-exploring random trees (RRT). A* is a variant of what we call *search-based planning*. In search-based planning, the map is discretized into a graph. So we can now apply graph-search algorithms which are very well studied and practiced in computer science. A* finds the shortest path on a graph by mapping each traversable

edge to a number. Then, A* will greedily chooses to travel on the best edge (smallest number). This mapping is where a lot of variability comes in as it encapsulates both distances along edges and heuristics, which are extra functionalities meant to improve performance while maintaining optimality. RRT, on the other hand, is a *sampling-based planning* method. Unlike search-based methods, sampling-based methods are able to function on continuous spaces. They are generally faster and perform better with scale than search-based methods. However, sampling-based methods do not provide the same optimality guarantees as search-based methods. RRT is a sampling-based method that samples random points on the map, and determines whether a path to that point from the "current" point is collision free. If any points are collision free, the algorithm will add the trajectory to that point, travel to the point and make it the "current" point, and repeat the sampling process. Finally, for path-following we implemented a pure-pursuit algorithm. This algorithm served as an easy interface between a trajectory and the car, as it was intuitive to calculate a look-ahead point, the point used to guide the car in pure pursuit, given set lines on a map (our trajectory).

As the final lab of the semester, this lab brought together many of the lessons and functionalities of our past labs. For one, the localization module implemented as part of lab 5 was a crucial part of path-following, as the robot would be unable to follow a path if it did not where it was. Moreover, we had to utilize our gained skills in software development to adapt and optimize our algorithms in addition to drawing upon our experience in writing controllers for the car. With path-planning in hand, the robot has effectively fully autonomous motion, and is one major step closer to being a fully autonomous system.

2 Technical Approach - Anirudh Valiveru, Anish Ravichandran

Designing a system that can drive involves the interface of two very different technical modules: the path planner and the controller. As a part of this lab, we decided to design two distinct strategies to plan a path between the robot's current position and a goal position specified in RViz: a search-based deterministic approach with A* and a sampling-based stochastic approach through RRT*. Both approaches have their pros and cons; while the search-based planner is guaranteed to reach an optimal solution given enough time, the sampling-based planner is able to search faster in more complex environments with multiple parameters. Additionally, search-based planners require the discretization of free space, leading to "close-enough" solutions that are not necessarily the shortest, while sample-based planners are only limited by the resolution of the space's representation.

The paths that our algorithm generates are represented as an ordered list of

points in the global map frame. Once a path is found, we use the pure pursuit control algorithm to compute the steering angles necessary to follow this line, turning a fixed lookahead distance ahead of the car along the path. The mathematics and tradeoffs between parameters in pure pursuit in comparison with other control methods will be detailed in section 2.2.

2.1 Path Planning - Anirudh Valiveru

2.1.1 Search-Based Planning: A* Search - Anirudh Valiveru

The basic approach to search-based path planning leverages algorithms that are used to find the shortest path between two nodes in a discrete path, such as breadth-first search (BFS), Dijkstra's Algorithm, or A* Search. In order to do so, the first step necessary is to convert the continuous 2D map plane into a discrete graph representation with explicit weighted transitions between the nodes, representing their distances. Because of this, it follows that the optimality of the search-based planning method depends directly on the granularity of the discretization method chosen.

Our group discretized the search space by rounding the coordinates of the start and end points, and then only transitioning between points with integer coordinate values. At each point, then, there are up to eight possible neighbors that the car can visit, representing the eight adjacent points.

Some other options for discretization include using points that are closer to each other or searching through configuration space (including the robot's rotation), instead of solely focusing on the position of the robot. However, we realized that using points that are closer together increases the runtime of graph search. We also ultimately decided against searching through configuration space because the trajectories required for pure pursuit only require the positions of each point to make the algorithm work.

Once the nodes and node transitions are defined, the next step is to implement a graph search algorithm that will give us the shortest point between two points on our graph. Since the length of transitions is not uniform, we need to implement a weighted graph algorithm, such as Dijkstra's Algorithm, to perform this task. Our team implemented A* search, which is an extension of Dijkstra's Algorithm. While Dijkstra's algorithm searches in all directions equally, picking the next neighbor to explore solely based on the lowest edge weight, A* search adds a second "heuristic term" to the exploration priority, encouraging the exploration of nodes that are closer to the end goal before exploring other nodes. The heuristic that we added to all of the priorities is the Euclidean distance from the neighbor to the end goal; if the Euclidean distance is smaller, then it has a higher priority in exploration than the other nodes.

The choice of heuristic is another design choice that we had to make as we

developed the algorithm. We chose to use Euclidean distance instead of the Manhattan distance or distance on the Dubins path for the sake of simplicity; any admissible heuristic that accomplishes the same objective will perform reasonably.

If implemented naively, discretization with A* search will provide us a reasonable path that connects the two points. However, some of the paths generated by A* will hug a wall as they traverse the space, which becomes a problem because while the path does not cross into the wall, the car is not a point mass. To avoid this, our team applied a binary dilation to our occupancy map, increasing the width of each wall to make sure that all paths are a considerable distance from the wall.

The choice of discretization strategy, heuristic, and map modification are the core decisions that we made in our design. While there are many different tradeoffs in these choices with respect to performance and simplicity, we found that our choices provided a good baseline upon which our pure pursuit controller could be built and tested.

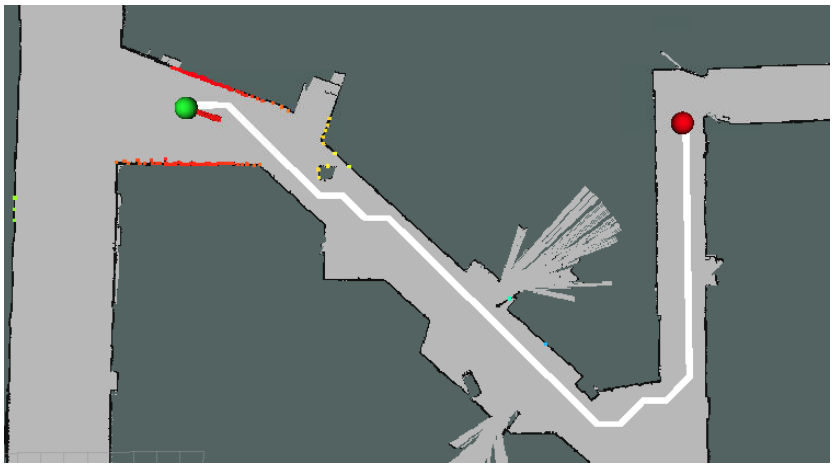


Figure 1: A* path planning result

2.1.2 Sampling-Based Planning: Rapidly Exploring Random Trees - Anish Ravichandran

Once our search-based path planning algorithm performed satisfactorily, we moved on a sampling-based approach to determine how these two methodologies compare. We chose to use Rapidly Exploring Random Trees (RRT) as a result, more specifically RRT*, an optimized version of RRT that is guaranteed

to provide the shortest path from the start to goal along the tree, with the tradeoff of computation and performance time. Because of the size of our problem and low level of rigor, we accepted this tradeoff because having the car drive along the shortest path was more essential than computational or time efficiency.

Here is the pseudocode of RRT* that we implemented:

```
define RRT*(start_point, end_point, map) {
  initialize tree;
  for n iterations {
    generate random_point;
    tree.add(random_point);
    find nearest_point;
    create new_point between random_point, nearest_point;
    if in_free_space(new_point) {
        new_point.cost = nearest_point.cost +
                    distance(new_point, nearest_point);
    } else {
        continue;
    }
    for all points in tree {
        if distance(new_point, point) > max_distance {
            point.cost = new_point.cost +
                    distance(point, new_point);
        }
    }
    tree.add(new_point);
    if new_point = end_point {
        break;
    }
  }

}
```

Though our RRT* implementation ran without errors and was efficient, we had issues getting it to avoid hitting walls. This is a point we want to address in the future.

2.2 Pure Pursuit Control - Anirudh Valiveru

The paths that our algorithm generates are represented as a list of points in the global map frame. Once a path is found, we use the pure pursuit control algorithm to compute the steering angles necessary to follow this line, turning a fixed lookahead distance ahead of the car along the path.

Once our path is found, there are six steps that we take to calculate the wheel's steering angle at each time step. These steps include: localize the car, find the

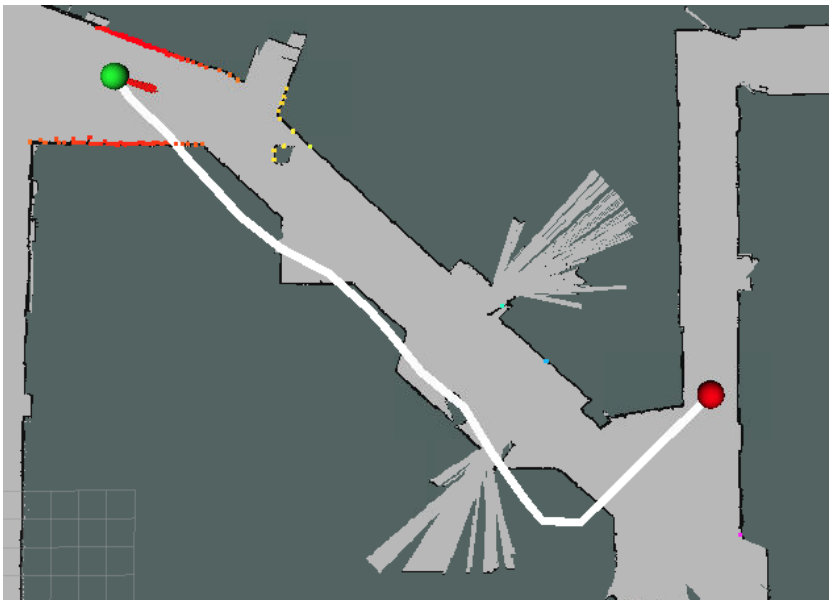


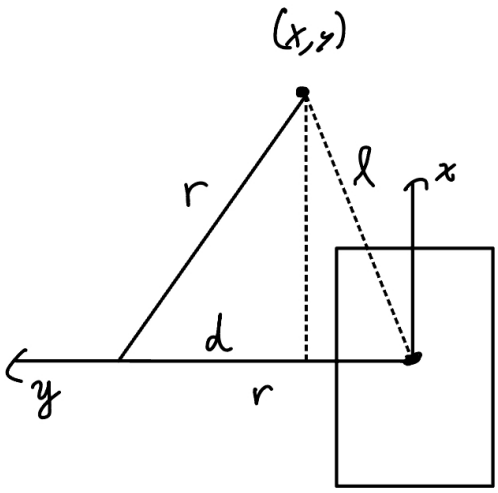
Figure 2: RRT* path planning result

closest point on the path to the car, iterate through the path to find a point on the path that is a set distance away, transform the point to vehicle coordinates, compute the vehicle's turning angle, and drive the vehicle using Ackermann steering.

The closest point on the car is found by computing the closest point to the car along each line segment on the path simultaneously using numpy. After this, we find the point l away by iterating through each segment following the minimum segment, computing the intersection between the line and the circle.

Once this point is found and converted to vehicle coordinates using a rotation and translation based on the vehicle's pose, we can use a derived formula to compute the final steering angle: $\gamma = \frac{2y}{l^2}$. A diagram of the vehicle and relevant equations are detailed in the figure below:

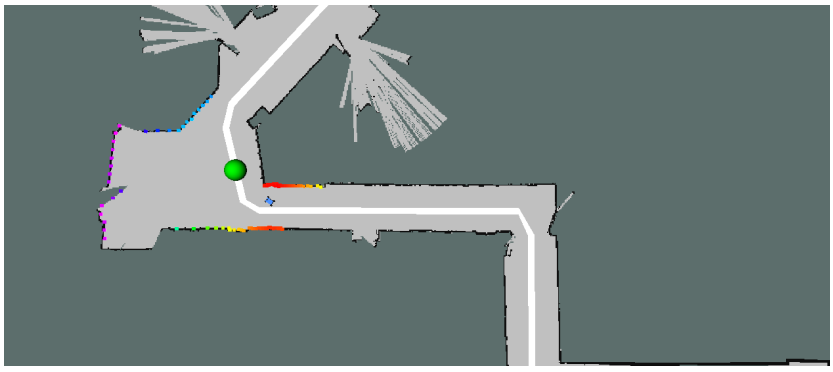
The two main parameters set in pure pursuit control are the lookahead distance and the vehicle's speed. As a general trend, a shorter lookahead distance allows the vehicle to turn with more agility but may also lead to unstable behavior at high enough speeds.



$y + d = r$

$x^2 + y^2 = l^2$

$\Rightarrow \quad r = \frac{l^2}{2y}, \quad y = \frac{2y}{l^2}$



3 Experimental Evaluation - Devin McCabe

The pure pursuit controller was evaluated by calculating the error of the actual followed path while following the loaded trajectory. The error was defined as the distance of the car's pose from the trajectory path. It was calculated by taking the magnitude of the vector perpendicular to the path to the current pose The error is calculated every time a new lookahead point is determined. For each test, the error was plotted as the car followed a trajectory going in a loop around the Stata basement. For a range of different lookahead distances and velocities, the error term was measured while the car completed the loop. The results from these tests help determine the relationship between the lookahead distances and velocities and the performance of our pure pursuit algorithm.

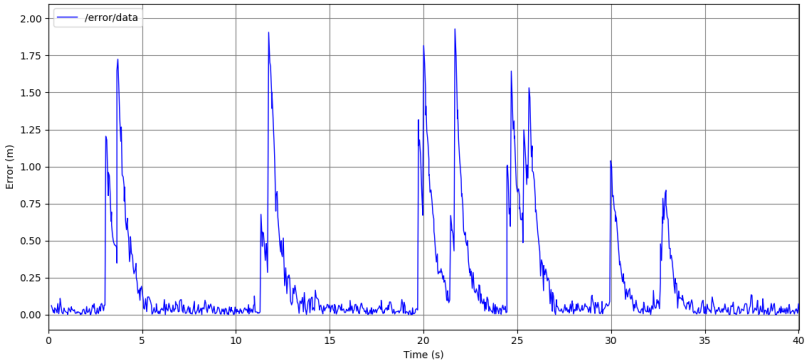


Figure 3: Error for a lookahead distance of 2 meters and a velocity of 4 m/s

From our testing, a lookahead distance of 2 meters performed with a number of favorable characteristics. The steady-state error stayed below 0.1 meters while the error while going around corners was always less than the 2-meter lookahead value. The error around corners was not enough to make it crash on tight corners and the car still followed the path relatively close. The settling time when going around a corner was approximately 2 seconds.
While the error for a lookahead distance of 0.5 meters has a lower average error. While going around corners, the error stayed low and did not spike as it had for the larger lookahead distances. The error stayed below 0.5 meters while going around the corners, while for the 2-meter lookahead distance, it stayed below 2 meters. The settling time was greatly reduced as well, taking approximately half a second to return to steady-state. Overall the error had significantly more variation and spikes. This most likely came as a result of the short lookahead distance causing the car to overshoot and oscillate approaching the path. This small lookahead distance also caused the car to get lost around tight corners, but this only happened infrequently.

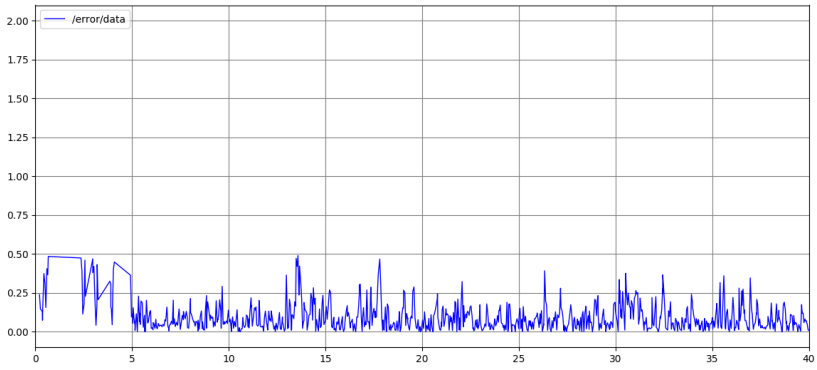


Figure 4: Error for a lookahead distance of 0.5 meters and a velocity of 4 m/s

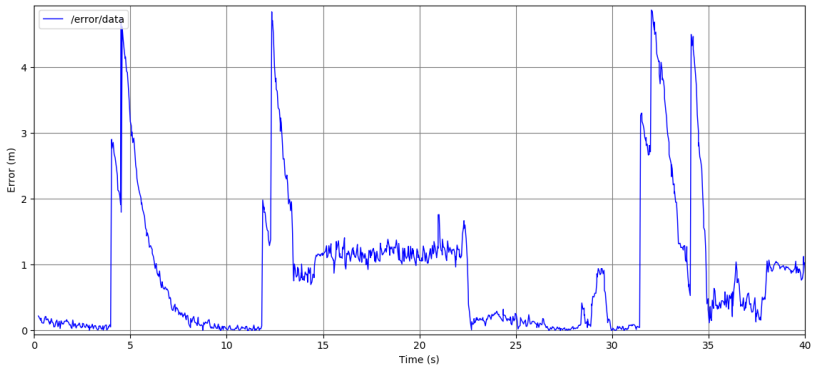


Figure 5: Error for a lookahead distance of 0.5 meters and a velocity of 4 m/s

The error for a 5-meter lookahead distance never exceeded 5 meters. This supports a pattern that the error stays below the lookahead distance set for that test. One issue with large lookahead distances is that they cut the corners and will often crash into the walls. This makes using a lookahead distance of 5 or greater a poor choice for our pure pursuit algorithm. The large lookahead distance decreases the steady-state error for straight sections. Setting the velocity and lookahead parameters is a balance of having optimal steady-state conditions and closeness to the trajectory going around corners. For our target goals of both following the trajectory close and making it to the end of the trajectory, a lookahead distance of 2 meters.

In the future, it would be good to put in place a variable lookahead distance that changed depending on the curvature of the trajectory. For corners and tight curves, a low lookahead distance performed well. For straight portions, a large lookahead distance decreases error. A function that has an output inversely proportional to curvature or one that decreases with high curvature would improve our pure pursuit controller.

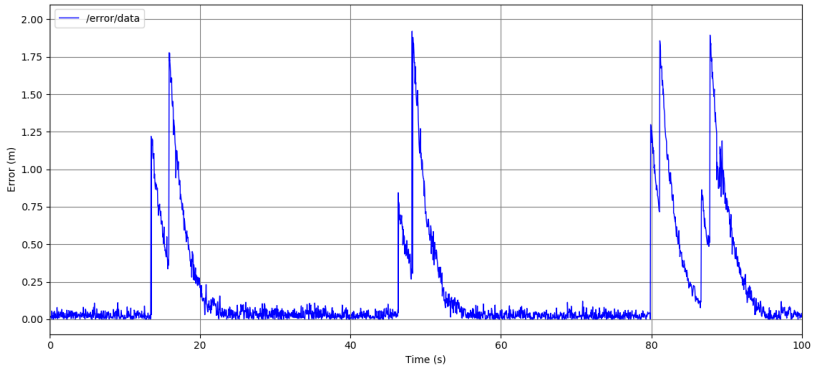


Figure 6: Error for a lookahead distance of 2 meters and a velocity of 1 m/s

In addition to testing lookahead distances, a range of velocities was tested for their performance. Overall, low velocities had lower overshoot and error when in steady-state. The velocity did not impact the error due to going around corners as this resulted from the lookahead distance and the pure pursuit function itself. Velocities up to 7 m/s did not significantly impact the error of our pure pursuit controller.

4 Conclusion - Arnold Su

Summarizes what you have achieved in this design phase, and notes any work that has yet to be done to complete this phase successfully, before moving on to the next. May make a nod to the next design phase.

In the end we have achieved two well-performing path-planning algorithms, while our pure-pursuit path-following algorithm has much room for improvement. As this lab shows, there was significantly more difficulty and roadblocks when it comes to implementing functionality that interacts with the robot directly, compared to modules that are purely software. However, despite it's flaws, the path-following module has adequate enough functionality such that when integrated with path-planning, the car was able to follow paths in both

simulation and real life.

The biggest priority moving forward is to improve and optimize the path-following functionality. The algorithm we currently have implemented is quite general and works in some simple settings. However, we want our car to be handle various settings and optimize for each one. For instance, having different cases for handling straight lines versus making turns. Along straight lines, we can afford to go faster and have less sensitive steering, while making sharp turns requires the car to be much slower and make sharp adjustments. While there can always be things to improve in path-planning, we feel path-following is where there is the most potential for growth, and will be better actuated into real-life results and functionality. In other words, path-following is what actually communicates with the car, so even if path-planning is perfect, the path-following will bottleneck that quality if it is not adequate.

From a non-technical perspective, this lab was a real test of handling the unexpected and severe road-blocks. It was also a test of collaborative coding. The pure-pursuit algorithm brought upon an unprecedented number of bugs and problems compared to previous labs, and there was a point where every single group member was contributing and messing around with the code. This may have incidentally cause more harm than good, as not everyone was on the same track and different strategies got muddled and confused together. We learned that for the future, taking the time to write out and think about the whole algorithm before implementing would be highly beneficial, as well as remaining vigilant on communicating our availability and commitments.

5 Lessons Learned

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

Arnold Su - From a technical standpoint, I mostly worked with attempting to implement the pure-pursuit algorithm. Here I struggled a lot with the algorithm, and was having a lot of difficulty debugging (especially since the module was unable to print anything for some reason). For that reason, I learned to think outside the box and find other ways to debug and deconstruct the code. For instance, I got a lot more familiar with RViz and publishing Markers (like to see lookahead points) to figure out what is going on in the background. Additionally, I got a lot more familiar with echoing topics and publishing in general since as an alternative to printing, I would just publish to a topic and rostopic echo it. From a communication standpoint, for our briefing we were in an unusual (and a little uncomfortable) position of having faulty algorithms, and not much "good" content. After discussing this with Dylan and the teaching staff, I learned that part of robotics is this process of failing, and that we must learn how to evaluate and present on failure as a guide for moving forward. Finally,

for collaboration I learned the importance of communicating beforehand when trying to implement algorithms/code together. Additionally, I looked ahead in my calendar/schedule and learned to effectively communicate my expected available bandwidth in the future.

Anish Ravichandran - I learned a lot from this lab both technically and socially. On the technical side, I feel like I finally understand how ROS works and how to properly code with ROS. By implementing proper debugging practices instead of print statements, it was easier to see how data was being passed through the various moving parts, giving me a better understanding of our system. On the social side of the lab, our team hit major roadblocks working on pure pursuit, and it really became a "too many chefs in the kitchen" situation. The benefit of having many people working on a single section is the exchange of ideas and correcting of errors, but the downside is the possibility of confusion and the latency cause by having to parse and understand someone else's code. I learned that even though I have the bandwidth for a task and can communicate it properly (major improvement from my performance in the last few labs), I still need to work on asking for help when I'm stuck, because it leads to roadblocks down the road for everyone if I don't. I hope to improve this in the future.

Devin McCabe - I learned a lot from this lab about collaboratively coding. I did not have much experience working on the same file as someone else in the past. For the most part, I had only used git to download or pull other complete files from the repository. Working actively with the rest of the group on the same file was a challenge for me, and I became more familiar with using GitHub. I also spent a great deal of time trying to make our path planning work on the robot itself. I learned even more about troubleshooting and problem-solving throughout the process. It was very helpful to make use of my resources and know where to look or who to ask for help. While I was not able to make the trajectory planner or pure pursuit controller work on the robot, the experience working through this process taught me a lot about how the car worked and how to work through these types of problems.

Anirudh Valiveru - This lab was one of the most important learning experiences for me, in both a technical and a collaborative sense. My primary technical contributions were involved in implementing A* search and pure pursuit planning, the implementation of which taught me many things about ROS, numpy, and transforming between different coordinate frames to make calculations. I also learned a lot about using rostopic echo with rospy.loginfo, for example, to debug code and print things when a print statement would not necessarily work in the same way. Beyond this, however, I learned a lot of things about managing progress and how to handle situations where people are unable to work towards their goals due to last-minute commitments. I learned that it is often beneficial to blur the lines between responsibilities assigned previously, being more flexible about tasks in pursuit of a greater team goal. I learned how to handle difficult situations where our work may not be where we want it to be, and hope that we

can take these learnings and apply them to perform highly on our final project.