Lab 6 Report: Path Planning

Team 16

Jared Boyer
Daven Howard
Amy Ni
Niko Ramirez
Kayla Villa

1. Introduction - Jared

Path planning is a crucial capability for autonomous vehicles to traverse its environment. Given a start point and a goal point within a known map, path planning allows us to generate an optimal trajectory connecting these two points. These trajectories can be optimized in different ways for different systems, but for our system, we wanted to generate a path that was both fast and safe. Specifically, we wanted to find the shortest path possible between the start and goal points, provide a smooth trajectory for the racecar to follow, avoid obstacles, and keep a safe distance from walls during sharp turns.

While comparing path planning algorithms, we found that A* would be most suitable for our desired system. A* will always return an optimal solution given a high enough map resolution, meaning A* would give us a fast path. Most other algorithms trade off this optimality for run-time efficiency, which is something that we prioritized less. Although run-time speed is always important, our path would only need to be planned once per scenario, meaning we could afford a much slower algorithm if it meant a faster path.

Beyond path optimality, we wanted path safety. In order to create a short, fast path while maintaining safe distance from walls, we decided to employ pre-processing techniques on the map like buffering. By thickening up the occupied radius of obstacles in the initial map, which is an operation called *dilation*, our racecar has a little extra breathing room when turning corners.

After planning a path between two points in an environment, it's equally important that the racecar can accurately follow this path. Building path following capabilities involves implementing an algorithm called Pure Pursuit, which is ubiquitous in modern autonomous vehicles. Luckily for us, we implemented a generalized pure pursuit library in previous labs, which allowed us to hit the ground running for path following test and evaluation.

By integrating path planning, path following, and the localization capabilities we built previously, this full stack navigation system can successfully navigate and traverse the stata basement autonomously. However, integrating these modules and testing on the car was more involved than initially expected. One of the primary problems we encountered during this process was finding the right balance between the speed and safety of our vehicle. The fast algorithm, path smoothing, and pure pursuit behavior all worked together to speed up performance as much as possible, but at the cost of cutting corners and running into walls. In contrast, map buffering helped provide safe behavior when the car took sharp turns, but often limited our performance in thin corridors and spatially-constrained areas. Balancing these effects was a difficult tradeoff to find, but by tuning the parameters of map buffering, path smoothing, pure pursuit, and the racecar speed, we arrived at a system that can quickly and safely traverse its environment.

Moving forward, we aim to further tweak our parameters to get an optimal performance

and we're shooting for a top score in the stata basement race.

2. Technical Approach

2.1 Path Planning

The first step for our racecar to successfully reach a goal location is to plan a path to that point. In this section, we will compare the strengths and weaknesses of various path planning algorithms, discuss why we ultimately chose the A*algorithm, and detail our implementation of A* to fit our specific problem.

2.1.1 Comparison of Path Planning Algorithms- Kayla

Path planning algorithms can be divided into two main categories: search-based and sample-based. In different contexts, one of these may be better suited for solving the problem than the other.

Search-based path planning algorithms work in two main steps. First, the map that the robot will be traveling through is discretized into a graph structure. A graph consists of a set of vertices, which represent points on the map, and a set of edges, which represent possible paths between points that the robot can take. These edges may be given weights to represent the cost of traversing this path. Once the graph is made, the next step is to search over the graph to find a path. The search algorithms depth-first search and breadth-first search differ in how they select the nodes to inspect next, with depth-first extending a single path to as close as it can get to the goal, and breadth-first inspecting all nodes that are the same level away from the start before moving on to nodes at the next level away. Dijkstra's search algorithm is an extension of breadth first search in which the edges are given weights, and the next node to inspect is selected based on its weighted distance from the start. Dijkstra's is guaranteed to find the shortest path for a weighted graph. If we are able to estimate the distance from each node to the end goal, then we can use the informed search algorithm A*. A* is resolution complete, meaning that if the discretization has a high enough resolution, an optimal path will be found, if one exists. More detail about this algorithm is given below in section 2.1.2.

Search-based algorithms may not be feasible for robots with a large number of degrees of freedom, which would produce a very large graph of possible states. As the degrees of freedom increases, causing the graph to grow, the runtime and memory usage of these algorithms will grow exponentially.

Sample-based search algorithms, such as Probabilistic Roadmap (PRM) and Rapidly-exploring Random Tree (RRT) try to solve these drawbacks. Search-based algorithms do not require the maps to be discretized, but some algorithms do discretize the space. These algorithms randomly sample the configuration space of the map and connect the samples with

trajectories. PRM randomly samples *N* points from the map, and connects each pair with a trajectory. It then checks if these trajectories result in a collision, and deletes them if so. This collision check can be implemented through various collision-checking algorithms. The result is a graph-like structure (roadmap) that may contain many paths from the start to the end goal. PRM is considered a multi-query algorithm because as long as the robot's environment doesn't change, the same roadmap can be used multiple times to find paths between different start and end goals. If the environment is changing, however, PRM will have to be run again. PRM is probabilistically complete, meaning that the probability that a solution is found, if one exists, approaches one as the number of samples increases. The runtime is difficult to characterize because of the random sampling, but it can be shown that a solution can be found "quickly" with high probability in environments that have good "visibility".

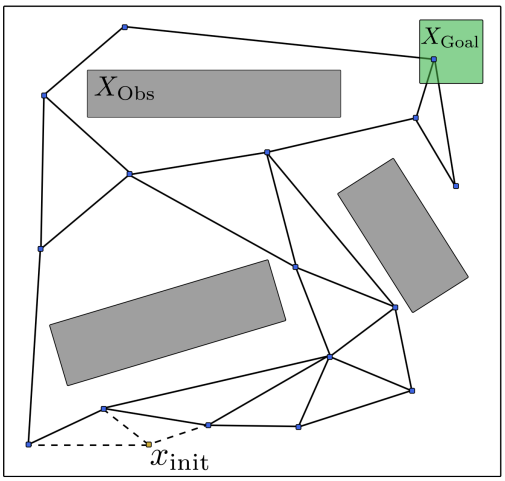

Figure 1. Example result of Probabilistic Roadmap sample-based algorithm.

The other main sample-based algorithm, RRT, is compatible for dynamically-changing environments. The algorithm samples one point at a time and if the trajectory connecting to that point doesn't result in a collision, the point is added to the exploration tree. More and more samples are added to the growing exploration tree until the goal is reached or time expires. The result is a tree-like structure. RRT is probabilistically complete and is a single-query algorithm because anytime you want to find a new path for new start and end points, the algorithm must be run again to generate a new tree. RRT can be extended to RRT* which re-connects nodes to update the connections with less cost, returning an optimal solution.

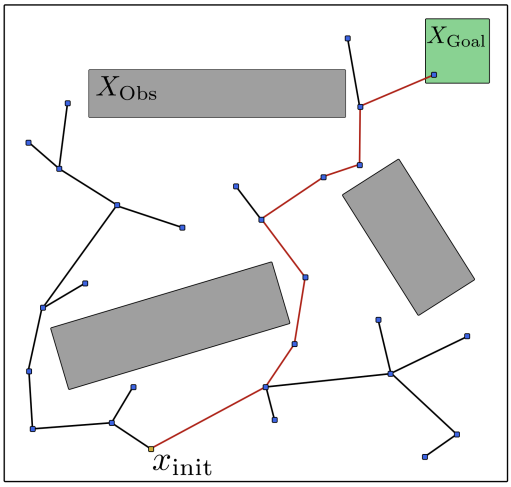

Figure 2. Example result of Rapidly-exploring Random Tree sample-based algorithm

Sample based algorithms can be faster than search-based algorithms, but may produce inefficient and odd-looking paths.

We considered these strengths and weaknesses of the various algorithms when deciding on which one to implement for our car, ultimately choosing the search-based algorithm A*. We chose to use a search-based algorithm because although sample-based algorithms can be faster, our path has to be planned only once, so we were okay with a longer runtime. We decided on A* because it is resolution complete and returns an optimal path under a specified heuristic. We also had previous experience working with A*, which was another consideration we made in our decision. More details about our implementation choices of A* are provided below in section 2.1.3.

2.1.2 A* Algorithm Overview

The A* search algorithm is an informed, best-first algorithm that can find the shortest path from a start node to an end node within a weighted graph containing nodes and traversable edges each labeled with a cost. Figure 1 shows a sample weighted graph where A* can be performed. The algorithm is an extension of the famous Dijkstra's search algorithm with a few modifications. Dijkstra's algorithm searches through the graph by keeping track of all the nodes it has visited as well as the cost it takes to traverse from the start node to each visited node. At each iteration, the algorithm finds the node with the least cost and extends that node by adding all the node's neighboring nodes to the list of visited nodes, where their cost will be recorded based on the cost of the edge that led to the node. In addition, each visited node will also have a parent node, which tells the algorithm from which "parent" the node came from in the graph. The process then repeats until the algorithm visits the end node for the first time, after which the shortest path can be constructed by backtracking to the start node via the

parent node attribute of each node.

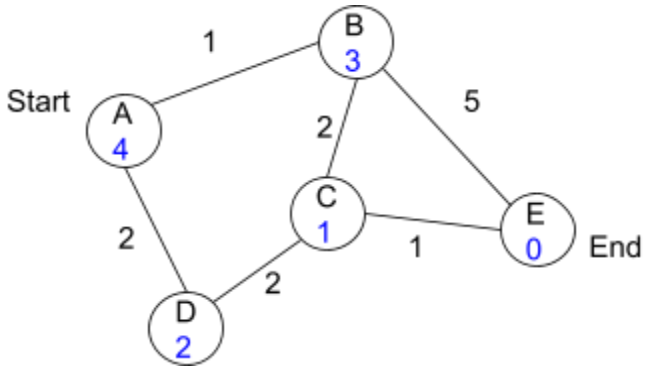

Figure 3. Example of a weighted graph

Compared to Dijkstra's algorithm, the A* algorithm uses a heuristic cost in addition to the regular edge costs. The heuristic of a node is the estimated cost it takes to traverse from the node to the end node. In Figure 1, the blue number underneath the name of a node is the estimated heuristic. An iteration in a generic A* algorithm is identical to that of Dijkstra's, but the cost of a visited node is now the sum of the actual cost from start to the node plus the heuristic cost of said node. Whenever the algorithm picks the least cost node to extend at the start of an iteration, it takes into account the heuristic and picks the node with the best chance of being on the shortest path to the end. The addition of a heuristic guides the search towards the general direction of the end node, while Dijkstra's algorithm tends to spread out in all directions.

2.1.3 Path Planning Implementation - Amy

To implement the A* search algorithm as a path planning tool, we first need to find a graph representation of the map. The map we are given is a list of integers representing each cell of the map, where the integer represents the percent probability an obstacle exists within the cell. After turning the list into a 2D grid, we can use each cell as a node of the A* search graph. To find the start and end node, the algorithm takes the predefined start and end positions relative to the map frame and uses the xy-coordinates to find the exact cell indices within the grid representation. Given the start and end nodes, we perform A* on the grid map where each cell has eight neighbors each referring to a neighbor cell in the grid. The cost of traversing from one cell to a neighboring cell is the Euclidean distance between the cells. The heuristic we chose is also the Euclidean distance between a cell and the end node. This is sufficient because our racecar has a relatively high degree of freedom under the assumption that the goal node is far enough away. In addition, we define each cell that contains an obstacle to have a cost of infinity so A* will never explore occupied spaces. After A* returns a shortest

path described as a list of nodes, we turn the nodes' according indices in the grid back into their actual coordinates on the map as a trajectory. These coordinates can then be fed into the pure pursuit module to allow the racecar to follow the path.

As we were implementing the path planning algorithm, we encountered several issues that required some modification to our approach. One of which was that the A* algorithm tends to find paths with sharp turns and jagged trajectories due to our use of a grid representation as the map. In order to smooth out the trajectory so our path following module can achieve a better performance, we effectively low-pass filtered our trajectory waypoints. Each point is updated to be the average position of its twenty closest points, smoothing out sharp corners without affecting straight line trajectories.

Lastly, we had to ensure that the path we found is safe for the racecar to follow, meaning it will not clip any walls along the way. Paths found with A* tend to get close to obstacles, especially during turns. In addition, our pure pursuit system that guides the racecar along the trajectory naturally cuts corners, possibly driving the racecar into obstacles. To avoid such collisions, we modified our grid map such that cells within a certain distance of occupied cells are also marked as occupied and inaccessible to A*. Figure 4 shows a visual demonstration of the map before and after applying this modification. This technique effectively creates a buffer along obstacles so the paths found by A* keep some distance from them, allowing the racecar space to drive and turn. The final buffer distance we chose was around 0.5 meters, which is larger than the width of the racecar, allowing tight turns.

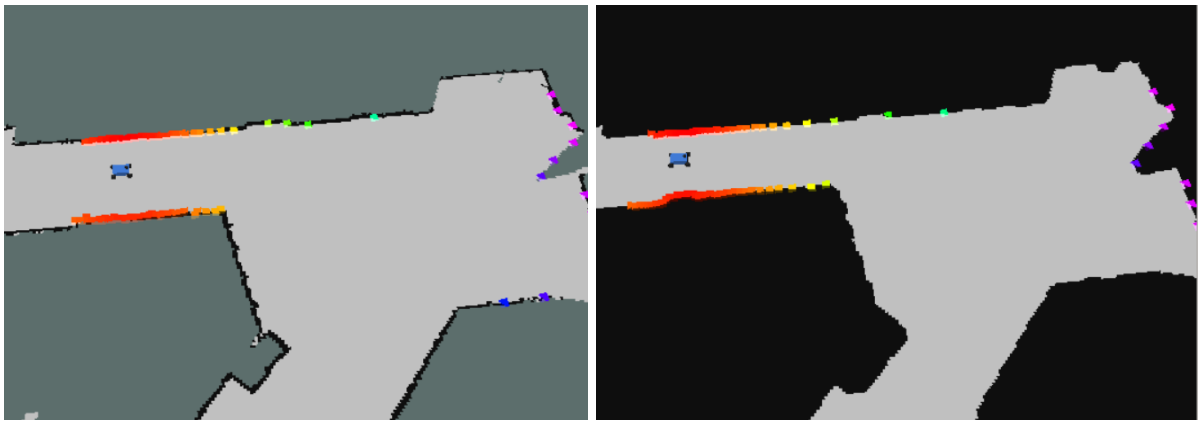

Figure 4. Visualization of map without buffering (left) vs with buffering (right)

2.2 Pure Pursuit- Daven

In order to follow the trajectory provided from our path planning, we used the pure pursuit algorithm. Having already implemented this algorithm in past labs, we only needed to modify it such that it would be compatible with the trajectory representation given by the path

planner. As a reminder, here is an overview of how pure pursuit works. When the racecar is given a trajectory, it attempts to find the furthest point along that trajectory which intersects the circle around the racecar with a radius determined by a given lookahead distance. Once this point is found the algorithm computes the turning angle needed for the car to reach that goal point.

The way the car calculates the next goal point to drive toward is by looping through all of the line segments of the trajectory and performing calculations to determine if a point of the line segment intersects the lookahead distance circle. In order to increase performance, we added bookkeeping such that the trajectory segments checked are only those which are ahead of the car. Once a point was found, we needed to convert the coordinates from the map's reference frame to the car's reference frame which was done by multiplying transform matrices. Upon completion of these calculations we used the equations in *Figure 5* to determine a steering angle based on the lookahead distance and the length of the wheelbase. Our pure pursuit algorithm was called every time odometry was received since it provided the new location of the racecar. This odometry was either received from ground truth in the simulation, or our monte carlo localization algorithm when used on the racecar in the real world.

$$R = \frac{l_d}{2 * sin\alpha} \qquad \delta = \arctan\left(\frac{L}{R}\right)$$

Figure 5: Equations to calculate steering angle

When modifying pure pursuit for this lab we needed to make a few key changes. The first was editing the algorithm so that it would follow a finite trajectory and then stop at the end. We did this by comparing the distance from the trajectory's end point to the actual racecar position and setting the racecar's velocity to zero when the distance was less than a small threshold. In order to have the racecar start following the trajectory even if the first point was not within the racecar's lookahead distance, we initialized the first goal point as the first trajectory point when a new trajectory was received. This allows for the steering angle to be calculated so the racecar can drive directly towards the start of the trajectory until it is able to update the goal point when the lookahead circle begins intersecting the trajectory. The last change we made to pure pursuit was changing the lookahead distance and velocity to be dynamically based on the steering angle. The philosophy behind this is a larger steering angle means a sharper turn which we would want to approach more slowly so we decrease velocity. We also want to prevent corner cutting as much as possible since this is a side effect of the pure pursuit algorithm. By decreasing the lookahead distance as well we are able to follow the trajectory more closely around sharp corners. The functions calculating the new velocity and lookahead distance based on steering angle were implemented simply at first but can be tuned

to improve the balance between performance on straights and sharp corners.

2.3 Integration

Combining the path planning and path following capabilities was a fairly straightforward task. It required re-plumbing rostopics and tuning the parameters for lookahead distance, path smoothing, map buffering, and racecar speed to arrive at optimal performance. Additionally, we leveraged our safety controller that was built in previous labs to ensure safety in the case of failure. Ultimately, we arrived at the following parameters for our racecar's nominal performance. In order to maximize our speed for applications such as the extra credit race, these would need to be tweaked accordingly. Also, we arrived at different parameters for performance in the real world versus simulation. The following table shows the parameters used on the real car.

Lookahead Distance	Speed	Map Dilation Radius
1.0 m	1.0 m/s	0.5 m

Figure 6. Racecar path planning and pure pursuit parameters

3. Evaluation

3.1 Path Planning Evaluation- Kayla

To evaluate our path planning algorithm, we used a simulated map of the Stata basement in Rviz. We selected a start position for our car and an end goal position, and then ran our path planning algorithm. Our first evaluation was a qualitative assessment of whether the planned path looked like it was the optimal path from start to end point.

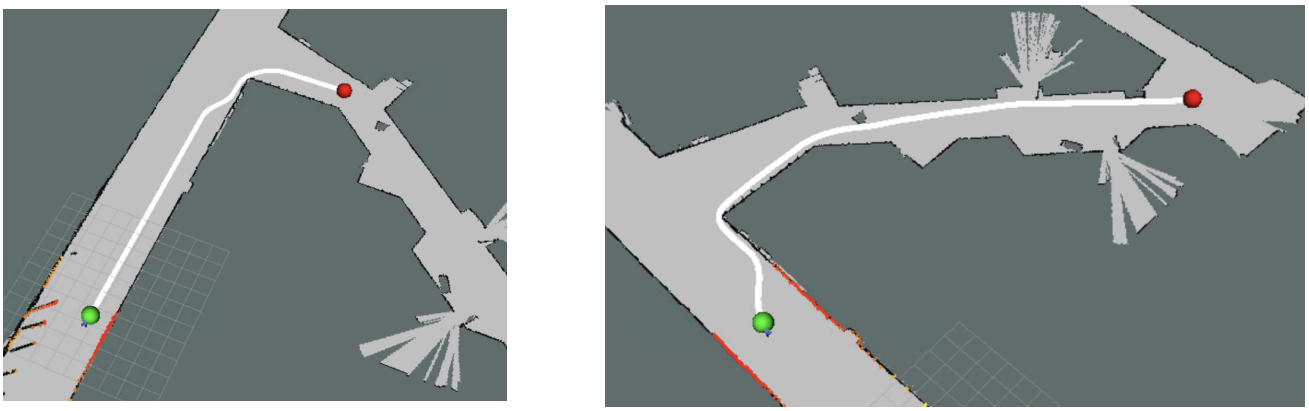

Figure 7. Two examples of our planned path in simulation.

For a quantitative evaluation of our path planning algorithm, we analyzed the runtime of our algorithm given various start and end positions. We found that the runtime was dependent on the distance to the end goal as well as how direct the path to the goal is (i.e. if there are many turns necessary to reach the goal or if it is a more direct path). For a short path the runtime was 0.13 seconds, for a medium path the runtime was 4.01 seconds, and for a long path the runtime was 29.49 seconds. This is a relatively slow runtime, especially for long paths. We expect that the runtime would suffer a lot if the car has to traverse through a map with lots of twists and turns. Runtime is an area we would like to improve on for our path planning algorithm and we have discussed various ways in which we could do this. For example, if we decrease the resolution of the map, there would be less nodes in the graph, resulting in a faster search. This change would have to be carefully balanced with map buffering, in order to make sure that a lower resolution doesn't result in a planned path that collides with a wall.

Figure 8. Short, medium, and long length paths used to evaluate runtime of A.*

3.2 Pure Pursuit Evaluation- Niko

In order to evaluate our Pure Pursuit model, we ran both quantitative and qualitative tests in simulation. Since we already implemented pure pursuit in previous labs, our evaluation was completely focused on optimizing it. As a result, the general approach we took was to iteratively improve our implementation by using tests in simulation to tune the necessary params. Firstly, we ran a qualitative test in simulation to ensure that the changes we made in order to restructure Pure Pursuit was working properly. This consisted of running the racecar on the stata basement map with a preloaded trajectory and then seeing how accurately it would follow the path. During these qualitative tests, we placed a specific focus on the ability of our model to make sharp turns and follow the path with minimal oscillations. This was a rather straightforward process given that we extensively tested pure pursuit in previous labs.

After checking that our racecar was able to follow paths in RViz, we then ran our model against the gradescope tests to have a quantitative check. This gradescope test initializes our racecar at a starting location and launches our pure pursuit and localization models. It enabled us to have a standard for measuring the ability for our car to follow a path. Specifically, it would give us the percentage of the given path we were able to follow before we exceeded a time threshold, entered occluded space on the map, or drove further than a present distance from the path.

The first few iterations revealed flaws in certain edge cases for our model. Specifically, the figure below shows a problem with our initialization setup. For our initialization, we would set the previous goal to a value before a trajectory was loaded. As a result, we would start pursuing a path to this last goal point even if the trajectory wasn't set. This led to our racecar just circling for the first few seconds as shown below.

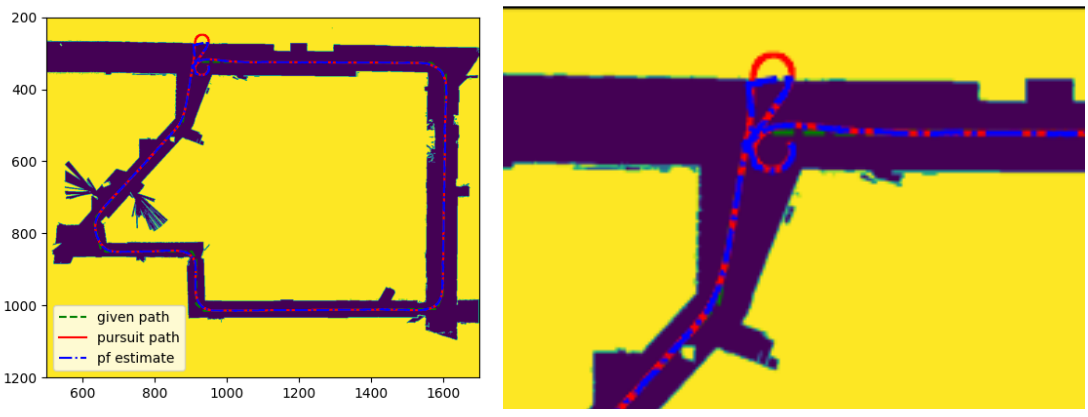

Figure 9. Initialization Problem Example.

Once we determined this was the cause of the issue, we were able to fix this edge case with a simple boolean check. After a couple more tests to tune the lookahead distance, we were able to achieve 95.06% accuracy when following the given planned trajectory. The path is

shown below.

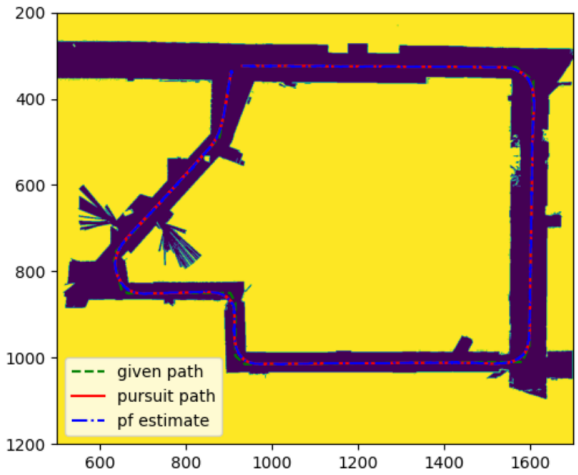

Figure 10. 95.06% accuracy for path following.

In addition to these findings, more quantitative analysis, parameter tuning, and evaluation occurred during the integration phase.

3.3 Integrated Performance Evaluation- Jared

In order to evaluate the performance of the system as a whole, we first conducted tests in simulation, gathering quantitative data on the racecar's tracking performance, and then conducted tests on the real car. For testing the real car, we primarily relied on qualitative evaluation, such as observing the car's driving behavior and noting any collisions (*Note: our safety controller stopped the car prior to crashing)*. If the car wasn't driving straight we would adjust the pure pursuit gains, and if the car was too close to the walls we could increase the map buffer.

Regarding testing in simulation, our car's full navigation stack performed very well. In the integrated gradescope test, which evaluates the system's path planning, localization, and path following in one, we scored a 98.77% following accuracy.

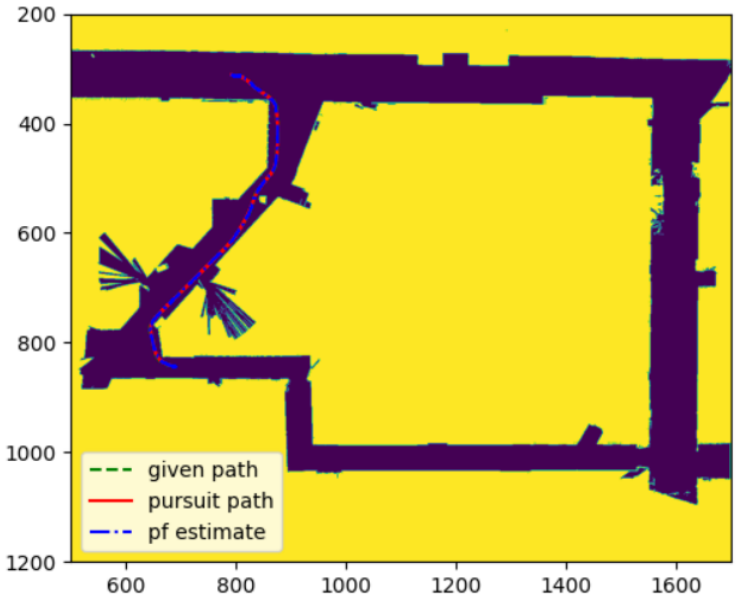

Figure 11. Integrated Gradescope Test

In order to get a better picture of how our integrated system performs, we also conducted some of our own tests in simulation and measured the cross-track error from the path at each point in time. Using the same test scenario, as shown in Figure 12., we first recorded the cross track error when using ground truth odometry for our pure pursuit algorithm, and then we recorded another scenario where we use our MCL odometry instead. When using ground truth data, the racecar followed the planned path with an average error of around 5 centimeters. Considering our final implementation keeps a 50 centimeter buffer from the wall (more than what is shown in Figure 12.), this error margin is completely acceptable. In our localization test case, the observed cross-track error is around twice as high. This is to be expected, as this test case combines the uncertainty of MCL with the uncertainty of our pure pursuit implementation. Although the error and noise present with localization is much higher, it still performs well within our error budget with an average cross-track error of around 8 centimeters.

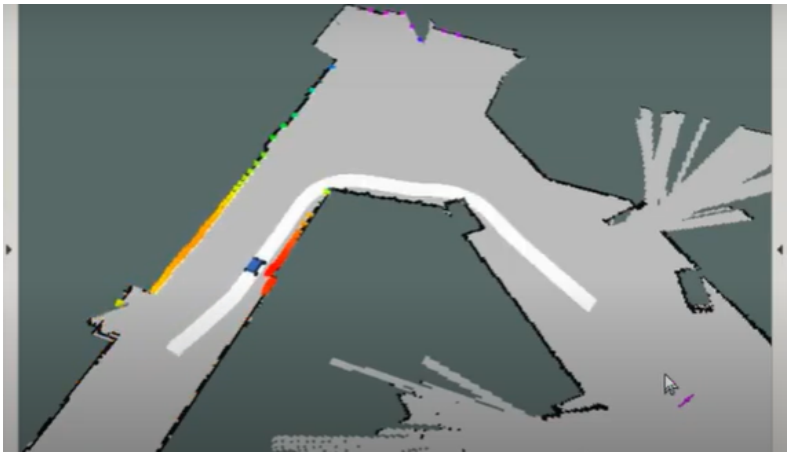

Figure 12. Simulation Test Scenario

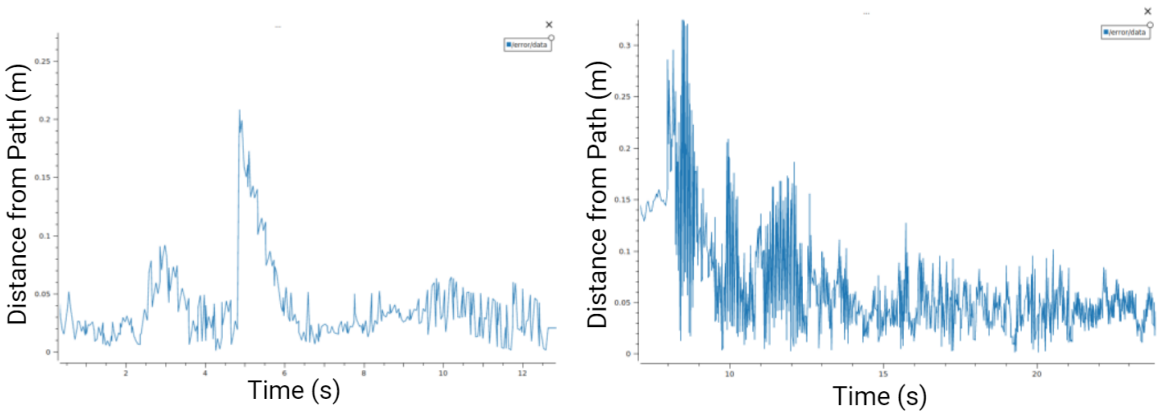

Figure 13. Cross-track Error in Ground Truth (Left) and Localization (Right) Scenarios

After we analyzed the performance of our path planner and pure pursuit implementation in simulation, we conducted a few qualitative tests on the real racecar to validate these results. Initial tests showed a mismatch between the parameters found in simulation versus the optimal parameters in reality- that is, the lookahead distance needed to be significantly increased to avoid unstable driving. After tuning parameters, we conducted a few final tests, asking the racecar to navigate different portions of the stata basement and noting its performance. Figure 14. below illustrates one of the test cases we selected, where the racecar navigates around a turn. Feel free to access the video of this test case at this link.

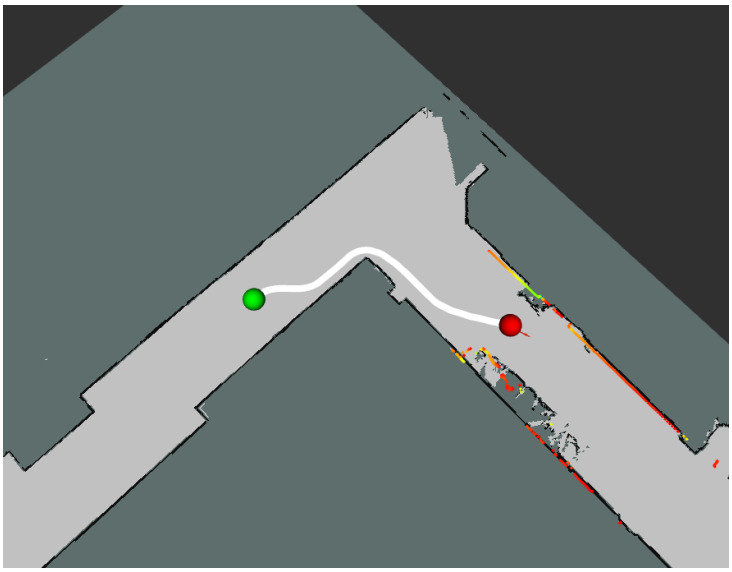

Figure 14. RViz View of Final Test Case

After multiple iterations of tests and navigating through different areas, we identified some areas of improvement. Although the racecar avoided collisions and followed the desired path accurately, we found that it came very close to collisions when navigating around turns. A*, path smoothing, and pure pursuit all try to cut corners as much as possible, so we found that our map buffer was still too small for optimal performance. Moving forward, we'd like to further tune these gains to prioritize safety around corners, potentially by increasing the map buffer twofold, which would allow us to attempt higher speed maneuvers.

4. Conclusion- Daven

Ultimately, we were successful at having our racecar autonomously plan and follow a path to a given goal location. Both in simulation and the real world we were able to generate a path with our A* algorithm and have our pure pursuit algorithm accurately follow the generated path. While we had other choices for algorithms such as RRT, we implemented A* and were able to get it to work well for our intended purposes. Our pure pursuit algorithm was also very successful at following the trajectory with a high degree of accuracy as evidenced in our evaluation.

Given our success, there are areas which can be improved in future iterations. While the A* algorithm was successful, as the length of the path increases the time taken to calculate the trajectory increases greatly. Making this path planning algorithm more efficient can be a focus in the future as well as looking into other algorithms such as RRT which are more performative with time constraints. As for pure pursuit, an improvement which can be made is the way the dynamic velocity and lookahead distance is calculated. With more analysis on the behavior of

the car as it approaches sharp corners, better functions of the steering angle can be derived to determine these values.

Now that our racecar is able to both plan and follow a path, as well as localize itself in its environment, we have almost all the workings of an autonomous racecar. As we move into the final challenge we will continue to refine all of our previous algorithms and add new functionalities so that our racecar can excel when it premiers at Johnson Track.

5. Lessons Learned

5.1 Kayla

Technical: In this lab, I enjoyed the opportunity to learn how to use search-based algorithms that I had learned in previous classes in a new problem setting: path planning for a car. I learned about the various strategies to turn a continuous map environment into a discretized graph that can be searched over, as well as the various strengths and weaknesses of the different algorithms to perform path planning. It was an informative experience to discuss with our group the advantages and drawbacks of each algorithm with respect to our unique problem because it gave us the opportunity to engage with what we learned about the algorithms to select the one that would be best for us.

Communication: I learned the importance of discussing the specs of a function that you are writing with other people who are writing functions that provide input to your function or use the output of your function. This is an important first step because it allows for separate people to work on separate pieces of code at the same time, parallelizing the process. If the specs are not properly discussed up front, then a lot of time may be wasted later when trying to integrate the code together.

5.2 Daven

Technical: In this lab I worked mainly on the pure pursuit algorithm and learned a lot about how it works. While we had an initial working implementation from previous labs, the different structure of this lab forced us to restructure our previous pure pursuit so it worked with our new path planning algorithm. This allowed me to really understand what was going on behind the scenes of pure pursuit and appreciate how effective it is at following a trajectory. It also gave me the opportunity to increase the efficiency slightly of certain aspects since I had a better understanding of the whole algorithm.

Communication: Discussing the lab at a high level and making technical decisions early on was key in this lab. Since we had to make a choice of which path planning algorithm to use it forced us as a team to talk about the tradeoffs of each algorithm in order to choose the one which best

worked for our situation. This helped in the long run as we were able to make our decision without spending extra time implementing multiple algorithms and use the saved time for bug fixing and integration.

5.3 Niko

Technical: For this lab, I learned the actual intricacies of pure pursuit and how to handle edge cases for trajectory tracking. I haven't had a chance to deal with the previous iterations of pure pursuit from previous labs until this one. As a result, I was able to dive into how it works and how to improve our current implementation. In terms of edge cases, I learned how to utilize testing to debug. We ran into a lot of issues with the racecar starting and I was able to use simulated tests to determine it was an initialization issue.

Communication: My communication takeaway from this week is the importance of setting goals and making decisions early. Given how open ended this lab was, we had many ideas for path planning and possible pure pursuit upgrades. However, we realized that time was a major factor, so we set goals early for what we considered the best path forward (no pun intended). This allowed us to focus on implementation early.

5.4 Jared

Technical: One of the most frustrating things I encountered in this lab was trying to tune parameters for the system based on test cases. By going from one iteration to another and making slight tweaks, better performance isn't guaranteed, and it's often very easy to forget the conclusions you drew previously. For our system, there was a very delicate balance between map buffering, path smoothing, and look ahead distance, and tuning these was difficult. In the future, it would be more useful if I approached this in a more organized way, like standardizing test cases, getting quantitative results, and recording observations.

Communication: When writing software as a team, it's easy to run into conflict when there are discrepancies in the ways inputs and outputs are defined in different sections of code. When working on A*, we spent a good chunk of time explicitly defining how we wanted to represent our map, graph object, and trajectory. This allowed us to work in parallel without running into conflicts, which was very helpful.

5.5 Amy

Technical: So far, I believe the integration step of this lab took the shortest amount of time compared to the previous labs. The fact that we discussed beforehand what each section needs to do to make sure everything fits together at the end made integration seem simple and trivial, while in other labs the integration step has always accounted for a large portion of the time we

spent working. This experience taught me how important it is to plan ahead when we split up the code just so we don’t need to make major changes later to accommodate each other’s code.

Communication: Similar to the technical lessons, making sure each member of the team understands exactly what they will have to work with was a big theme in terms of splitting up the work. In addition, a lot of the work had to be done within the weekdays for this lab, so it was very important for each of us to understand each other's workload outside of this class and when exactly each member can come in to work on the lab.