Lab #6 Report: Path Planning

Team #18

Cindy Wang
Eric Gonzalez
Herbert Turner
Jose Lavariega
Reinaldo Figueroa

6.141 RSS

April 21, 2022

1 Introduction (Eric)

In this lab, the group researched path planning algorithms and how to design a controller to follow these paths. In robotics, it is often necessary to follow the most optimal path within an environment in order to maximize efficiency. After all, a system does not want to waste its energy or lose precious time in pursuit of its goals. Therefore, the problems researched in this lab highlight two important topics in robotics: planning, and control.
Before explaining these two topics, we first make note that this lab also makes use of another important aspect in robotics: localization. In the prior lab, we implemented a means for a robot to localize itself within a static, known map. While this lab does not focus on localization, it is important to note that the robot does localize itself using our algorithm. This information is vital for the planning and control steps that were implemented for this lab.
Using the robot's estimated location, the team can implement planning and control algorithms. The path planning algorithm is given an end goal location, takes the robot's estimated pose as the start location, and uses a provided occupancy grid to avoid obstacles. Solving for optimal paths is a common problem in Computer Science, and so we drew from the wide repertoire of algorithms for inspiration. The team decided upon the Rapidly-exploring Random Tree (RRT) sampling based algorithm. Such an algorithm sacrifices the guarantee of an optimal solution for algorithmic speed. After all, a robotic system cannot lose too much time to computation. We discuss our choice further in this report. Overall, we are successful in generating paths between the start and end locations.
Once paths are generated, our controller takes over to guide the robot along the path. As done in prior labs, we implemented a pure pursuit controller. Despite having written pure pursuit controllers before, we faced a new challenge in this lab. Here, the challenge was to derive which points to drive towards as the vehicle traveled along the line. The lab report explores the choices made as well as the details behind our approach. Overall, it proved successful as well.
The pieces that comprise this lab can be tested independently. For example, the path planning algorithms can generate paths and be visualized in simulation. As for the controller, one can manually create any path, optimal or not, and have the controller follow it in simulation. Below, we discuss how each segment performs, and then address how it all comes together in simulation and in the vehicle itself.

2 Technical Approach

2.1 Path Planning Algorithms (Cindy)

The first objective in the implementation was to plan a trajectory for the car to follow in a known occupancy grid map from the car's current pose to a goal pose using either a search-based or sampling-based motion planning method. We implemented the search-based method using the A* method and the sample-based method using Rapidly-Exploring Random Tree (RRT) method. From these path planning algorithms, we opted to return a trajectory in the form of a list of poses. These poses when connected form the path, and the pure pursuit controller then follows this path.

2.1.1 Real World Coordinates to Pixel Map Transformation (Cindy)

We are given an occupancy grid map that consists of a 1 dimensional array, indexed by row major order, where each element represents the probability that an obstacle lies there in the map, from 0-100 and -1 if it is unknown. The occupancy grid also includes a map resolution which represents the meters/cell ratio. Since we are given start and end positions as poses in real world coordinates, we needed to implement a transform from the real world coordinates to an index within the occupancy grid.
After transforming to the occupancy grid pixel indices, then we use RRT algorithm to obtain a path in terms of nodes/map pixels. We then need to transform this path back into real world coordinates for the pure pursuit controller to use.

2.1.2 A* Algorithm (Herbie)

A* is a graph search algorithm that will tries to find the shortest path between two specified nodes, the start and end nodes. Each node of the graph must store three values: f,g, and h. g is the "actual cost" to get from the start node. It can be calculated as f of the previous node + the cost of the edge to the current node. h is the "estimated cost" to reach the end node from the current node. This is calculated based on the heuristic you choose for your implementation of the algorithm. Lastly, f is the "total cost" of the node and is simply the summation of g and h. Given these values, A* performs the following steps:

1. Instantiate some form of priority queue with start node and empty set for "closed nodes"
2. Pop top of queue and consider that the current node
3. Check if it's the end node and break if so
4. Check if its position in closed set and return to step 2 if so
5. Add the current node to the closed set
6. Loop through all neighboring nodes:
   (a) Calculate f, g, h of the node
   (b) If the node is not in the closed set and it's not in the priority queue or its value in the priority queue is greater than f, add it to priority queue with a priority of
7. Return to step 2

A notable property of A* is that with an **admissible** heuristic, A* is both **optimal** and **complete**. And admissible heuristic is any function that underestimates the "actual cost" from any node to the goal node. Optimal means that any path found by A* is guaranteed to be the "best" path according to the cost metric used by the graph's edges. Complete guarantees that if there is at least one path between two nodes, A* will find it. Combined with the optimality property, this means we are guaranteed to find the optimal path between any two nodes in a graph if such a path exists.

To allow A* to path plan properly and efficiently on the map, we dynamically create a graph as we search. To start, we create a node for our starting position on map with no parent and a value of 0 for f, g, and h and add it to our priority queue. Then, when we pop that node and subsequent nodes from the priority queue, we generate nodes for all 8 surrounding grid points with an edge to the current node weighted by the euclidean distance between the grid points. We then calculate our node values using euclidean distance as a heuristic since it's guaranteed to be the minimum distance between any two points and continue with the steps of the algorithm described above. One other note is that we used a three times eroded version of the stata basement map when running our A* algorithm since taking the optimal path often means it will hug the walls to closely for our comfort and car safety.

2.1.3 Rapidly-Exploring Random Tree (RRT) Algorithm (Cindy)

The sample based planning algorithm that we implemented was Rapidly-Exploring Random Tree (RRT). As an overview RRT works best on high dimensional spaces such as an occupancy grid map, and it works by randomly filling a space using a tree until a path is found to the goal point.

We represent each cell within the grid as a potential node, where it is a node if we have added it to the tree and not a node if it has not been traversed yet.

We begin the tree with a single node at the starting point and RRT expands the tree with each iteration of a loop, with some maximum number of iterations. For each iteration, we generate a random point, p_i, within the occupancy grid. We then find the closest node, n_i^c in our tree to this point by iterating through all of our current nodes and taking the minimum distance from the node to the random point. We then interpolate between the closest node, n_i^c, and the random point, p_i, to create a node, n_i^n that is 1 step size away from the closest node. This new node will store the closest node n_i^c as its parent. By storing the parent of each node, we can trace back to reconstruct the path from the goal node to the starting node. Note that the step size is the same as the $path_resolution$ that is described in the following tunable parameters section.

For each iteration, we also check that the new node that we are creating is within some distance of the goal node; if it is close enough, then we break out of the loop.

After exiting the loop, we can reconstruct the path from the goal node to the starting node, if it existed. We simply do this by using a while loop to continue iterating on the parent of the previous node until we reach back to the starting node. We can then return this list of nodes, which represent the cells within the occupancy grid.

2.1.4 Tunable parameters and Obstacle detection method for RRT (Rey)

Tunable Parameters in RRT implementation (Rey)

In our RRT implementation there are multiple parameters that can be changed and that will have an effect on its performance. Two of these parameters are:

1. path_resolution - Parameter that describes how long is the distance we move towards the randomly generated node inside our map.
2. goal_sample_rate - Parameter that dictates the chance at which we choose our ending location as our random node.

By changing path_resolution, we decide if we want to move longer or shorter distances towards the randomly generated node. It is not always a good idea to increase this parameter. For example, in places where there are a lot of obstacles such as walls, the higher this value is the higher the chance we will step or go over walls. This will force us to continue generating other nodes and also check too many times if there are obstacles in our path. Another disadvantage of increasing this parameter is that the higher it is the more distant each node in the final path will be. Although our pure pursuit can follow rough trajectories, in reality we would want a smother path which results in more continuous nodes in our resultant trajectory.

The goal_sample_rate will define how likely it will be to choose our ending position instead of a randomly generated node. This will allow the algorithm to continuously check if the path towards our goal is clear and move in that direction. Increasing this parameter might unnecessarily check if there is a clear path towards our goal, especially if it is early during the path finding process. Therefore, goal_sample_rate is usually set to a low value. In our case, there is a 5% chance that we will select our ending position as our target node. However, it makes sense to have a higher goal_sample_in open spaces since there will probably be a clear path towards our desired location.

Obstacle detection used in RRT implementation (Rey)

Whenever we want to move towards the node randomly generated by RRT, we will want to check if there are any obstacles in between the current and destination node. In order to do this, we will want to find the angle with respect to the world frame orientation that the line between these two nodes represent. Once we find the angle, we will move one step size (defined inside the obstacle finding function and which is fairly small) towards that direction, and check if the point this new location represents in the occupancy grid map is an obstacle. If it is, we stop and consider these two nodes are not able to be connected with a line. Otherwise we continue until we get to the final point or find an obstacle along the way. If we reach the destination node without finding any obstacles, these two nodes are now connected and considered as a viable path. This method of obstacle detection might not be an optimal solution, but it performed really well during testing and simulation. In addition, using an eroded version of the map is always recommended to avoid paths being created through holes in the map due to missing scan data.

2.1.5 Evaluation of Performance(Herbie)

In this lab, we explored the search based algorithm, A*, along with the sample based algorithm, RRT. We observed that A* will produce an optimal path, optimal meaning that it is the shortest path possible from the start position to the goal position. We observed that RRT, on the other hand, is non-deterministic due to the random generation of positions to create the tree, and this makes it such that the path generated is not guaranteed to be the shortest path possible and typically doesn't. Below you can see how inefficient RRT is when compared to A* on both an 8x8 and 22x22 grid. However, RRT does have an advantage in that it tends to find a suboptimal path to the goal state

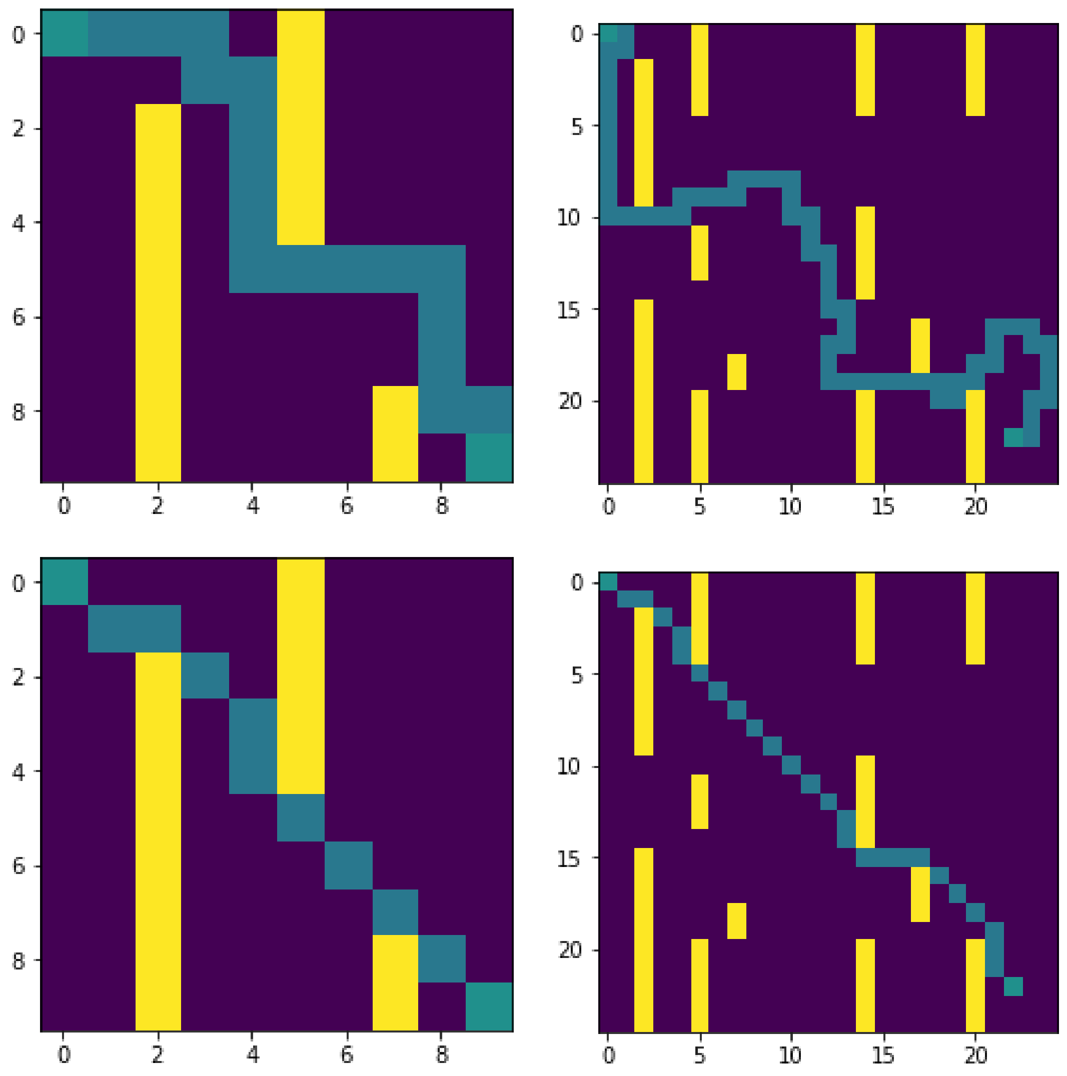

Figure 1: Top Row: RRT; Bottom Row: A*

extremely quickly, while A* struggles to perform in real-time with high grid resolution. To evaluate the tradeoff between the two algorithms, we examined both the efficiency and speed of the paths found. Efficiency was measured by calculating the total distance of a path based on the summation of the euclidean distance between each of its waypoints. Speed was evaluated using timeit magic function in ipython. We examined 5 different start and end node combinations on the stata basement map to evaluate performance on the following scenarios: a short straight-away, a long straight-away, a corner, multiple corners, finding the shorter of two long possible paths. These were labeled tests 1-5 respectively. Below you can find the tables of our results.
As you can see while A* does provide more efficient paths it does so at the expense of real-time computation.

Table 1: Distance(Map Units) benchmark

Algorithm	Test 1	Test 2	Test 3	Test 4	Test 5
RRT(max_iter=10000)	526.88	696.63	1281.84	1865.10	Failed
A*	393.07	557.38	1049.63	1365.85	1154.80

Table 2: Time(s) benchmark

Algorithm	Test 1	Test 2	Test 3	Test 4	Test 5
RRT(max_iter=10000)	.06	.07	.34	5.47	2.71
A*	16.23	18.57	87.06	73.57	74.99

2.2 Pure-Pursuit Controller (Jose)

This is the third lab where we have implemented a Pure Pursuit Controller, which has for now become a familiar control paradigm for us. The challenge with the first one was to start to frame the pure pursuit problem as a set of tunable parameters, but we ended up using a linearized pure pursuit that still relied on PD gains. In the cone follower lab we ended up implementing a more general version of the pure pursuit, where the point to follow was always defined as a cone. The challenge with this lab was to find the point to follow, given that it is constantly changing. This is the first lab where we had to determine which point to follow as a ROS callback, and the challenge was to find such a point.

The generalized equation for a Pure Pursuit controller is the following:

$$\delta = \arctan(\frac{2L_{self}\sin\theta_{pt}}{L_{look}})$$

Where δ is the angle at which the wheels are commanded to be positioned, L_{self} is a measurement on the car from driven wheel axle to steering wheel axle, θ_{pt} is the relative angle from the car to the follow-point, and L_{look} is the lookahead distance, a tunable parameter.

Finding the Follow Point (Jose)
To determine the follow point, we first make some simplifying assumptions about our path planning and trajectory following stack.

1. There is always a loaded trajectory in the racecar
2. The car starts reasonably close to the trajectory
3. the trajectory is made of at least three points

From there, our algorithm works as follows:

We first find the distances to all the segments created by the waypoints in our trajectories. That is to find the distance from our car to the segment composed of waypoints 1 and 2, waypoints 2 and 3, 3 and 4, and so on. This is done through finding the distance to the projection of a point on the line.

Once we have the distances to all possible segments, we choose the segment that is closest, and evaluate intersection points with a circle of radius of our lookahead distance to the line segment. This is framed as a quadratic equation and so we assume that we will only get 0, 1 or 2 solutions. Given the following tuple: (Q, P_1, r, V) we want to construct some distance from a point to the tuple. This quadratic equation is then:

$$(v \cdot v)t^2 + 2(v \cdot (p_1 - q))t + (p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q - r^2) = 0$$

Which gives intersection solutions of the form:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

.

Of course, we care only when we have a positive discriminant. Our solution is then to grab the point that is furthest along the line, in the case where we have two solutions: $p_1 + t_{furthest}v$.

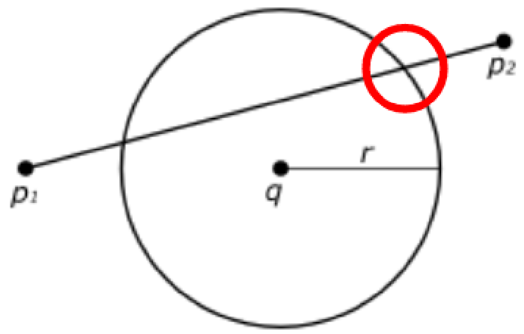

Figure 2: Chosen intersection point in two solution case. p_2 represents further along trajectory.

This obtained intersection point is described in figure 2.
Once we have obtained the intersection point. We transform this point over to coordinates in the reference frame of the car, such that the point is represented in relative coordinates. Then we can use the pure pursuit controller formulation from our cone follower lab to follow the defined point.

Controller Evaluation and Additional Ideas (Jose)
Pure pursuit is a control scheme where the only tunable parameter within the context of RSS is the lookahead distance. The point-to-follow is a separate parameter, but that is described in the above section. We iterated on multiple control schemes, to follow a determined loaded trajectory against stata basement. The selection of each of these candidate iterations were chosen based on a couple of guiding principles:

1. Choose a controller that will safely finish the racetrack.
2. Choose a controller that minimizes time around the trajectory.
3. Take heuristics whenever they make sense.

Given the context, we came up with three candidate controllers to test on running a trajectory: The first controller would be pure pursuit with a static lookahead distance, the second would be pure pursuit with a dynamic lookahead distance, and the third would be a hybrid controller that uses pure pursuit and a wall follower controller on straight-aways, where localization can be a bit more problematic to deal with. In the Wall following sections for the third controller, the speed is increased. The dynamic lookahead distance in the second controller is chosen proportional to the car speed, so the car speed also changes depending on the section of the track the car is in.

In figure 3 we show the performance of these controllers with respect to error against the overall predefined trajectory. Here the error is defined as the distance from the closest line segment, which in this case means the perpendicular distance from the trajectory.

Note that the occasional spikes represent areas where the car was going through a corner, while the larger spikes represent areas where the car localization failed and the car had to go off of the map.In this case the car was reset to a position where it could keep following the trajectory.

From the above we can see that pure pursuit with a dynamic lookahead distance drifted the least from the actual followed path. At 2.5 m/s, the car completed the base trajectory in 63 seconds. At 5m/s, the car completed the trajectory with 42 seconds. When activating the wall follower during long straightaways, the car was overall faster, at 45 and 49 second completion times. However, the car ended up not being as safe when it was run in conjunction with the wall follower. This is because the wall follower was perhaps activated too early, and there were still sections of the track that were confusing for the wall follower at the beginning of the straightaways. Perhaps a possible ammendment could be to activate the wall follower controller at a later time with some tolerance after passing through

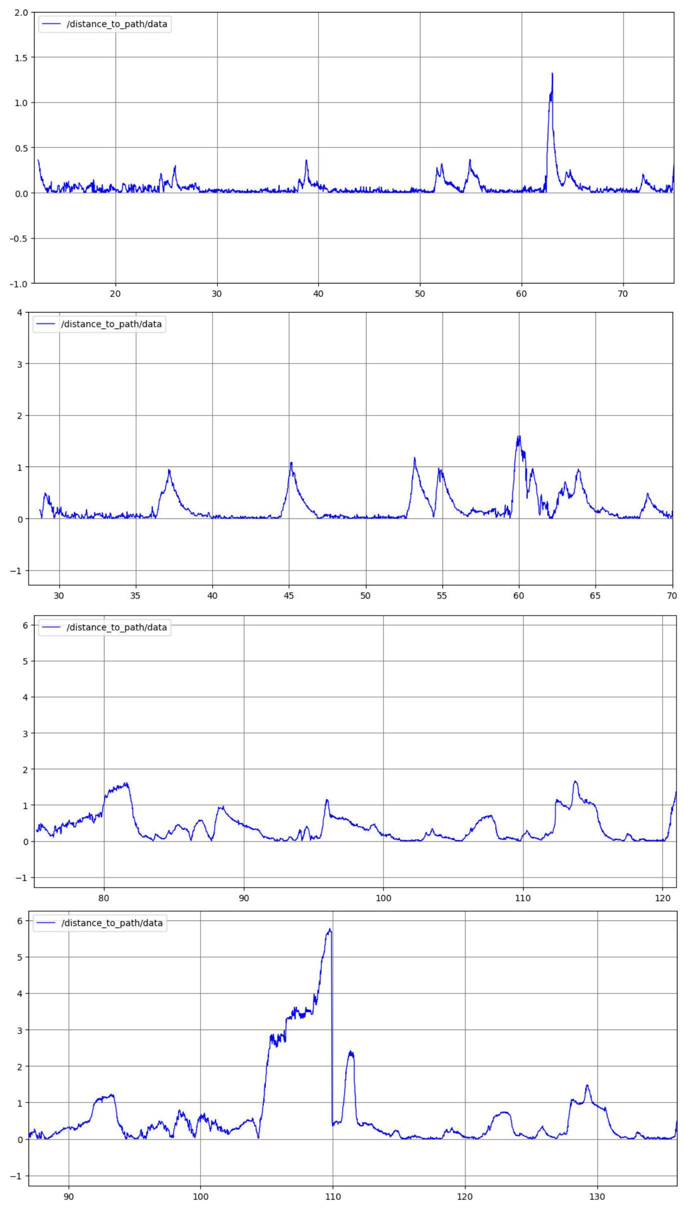

Figure 3: Error across different trajectory following controllers. Horizontal axis is time, vertical axis is absolute error in meters. (distance). From top to bottom: 1) Only with pure puruit at 2.5m/s , 2) pure pursuit running at 5 m/s, with a dynamic lookahead distance depending on speed, 3) pure pursuit with wall following, 4) pure pursuit with wall following, a second trial.

a corner.

Which motivates our last controller selection as simply pure pursuit wtith a dynamic lookahead distance depending on speed, **where the set speed is 5 m/s.**

3 Integration on Car (Eric)

3.1 Path Planning

There is only one significant difference between testing in simulation and testing on the vehicle for path planning. In simulation, the vehicle can determine its actual location in the map by listening to the appropriate rostopic. From there, this pose can be used as the start location for the path algorithm.

On the other hand, the actual vehicle does not have this luxury. Instead, the car must also run its localization algorithm to determine its location within the map. The vehicle listens to the rostopic containing the estimated pose and uses it as the start of the path planning algorithm. This can also be tested in simulation, so long as the rostopic for the estimated pose is used instead.
Aside from how the algorithm receives a start goal, little else changes between simulation and the physical vehicle/

3.2 Pure Pursuit Controller

The pure pursuit controller relies on the path it is given to decide upon the control action. Therefore, the controller uses a few tools to allow for testing.
First, we use a tool to manually construct paths within simulation, and place the vehicle at the starting location. From there, the controller is able to generate the appropriate instructions for the vehicle to follow.
A similar tool used to test the controller is a tool to load pre-defined paths into simulation. These paths can be generated manually, or could be provided by outside resources as well. Again, once a path is defined, the pure pursuit controller can act upon it.
In order to switch from simulation to working on the physical vehicle, one small change occurs. The path planning files must be ran in order to generate a path for the vehicle to follow. All three sources of paths publish to the same path topic, so no change must occur in the architecture. Therefore, simply running the path planning files would provide the vehicle the necessary paths needed to operate.

4 Conclusion (Eric)

The lab was successful in implementing a path planning algorithm and a controller to follow the generated paths. As shown by data in the report, the controller was largely successful in following paths it was given. While there is noise, this was to be expected, as the controller is not robust to handling corners. The controller is also reliant on accurate localization, which may guess incorrect locations and result in a failure to follow a viable path. Despite these issues, we can see the controller still performed well. In future iterations, the team can investigate different kinds of controllers, or modify parameters in the current controller. We could even reconsider how paths are generated, should we find that smoother paths significantly improve controller performance.
Beyond this lab, the ability to intelligently construct paths and use them to control the vehicle is incredibly important. There are many situations where a robot needs to find the optimal path between two locations in an environment. In addition, it needs to do so quickly and safely. In conjunction with localization, we are confident in our ability to tackle different environments and have the robot traverse it efficiently. Should there be important obstacles to note, we can even move beyond these topics and add an intelligent neural network. With this step, our robot would be able to assign meaning to certain obstacles and react accordingly. With everything built for the vehicle so far, we look forward to layering on even more tools.

5 Lessons Learned

5.1 Cindy

In this lab, I learned about different search algorithms and how sample based path planning can be better than a search based algorithm, even though it may produce a less optimal path. Because RRT very quickly generates random positions, it is able to quickly expand the tree until it reaches the goal position; however this tree may not necessarily have the best path. I also learned about the trickiness of working with transformations in the simulation from the pixel coordinate frame to real world coordinate frame. While things may appear to be looking okay in rviz, it is important to have a few benchmarks and sanity checks.

5.2 Eric

In this lab, I spent some time investigating search based algorithms, as well as ensuring that all components could operate together both in simulation and on the vehicle. Initially, I thought it would be helpful to try and implement a BFS or Dijkstra's algorithm, but quickly figured out the team needed to re-prioritize. While solving for optimal paths would have been nice, we found it took too much time, and so we dedicated more time towards creating a working sample based algorithm. Afterward, I checked that all components made sense together. For example, I integrated the localization component from last lab, and ensured path planning and the controller worked well while running simultaneously.

5.3 Herbie

I learned a lot about the theory and implementation of both sampling and search-based path planning algorithms. I also explore some trajectory smoothing techniques to help make paths more realistic. I discovered the importance of properly profiling code in this lab. A bottleneck that had me stuck on my A* implementation for almost a week could have been solved in just a few minutes if I had made sure I had properly evaluated where the bottleneck was instead of assuming based off just a few data points.

5.4 Rey

In this lab I learned a lot about search-based and sample-based path planning. Specifically, I worked on the implementation of the Rapidly-Exploring Random Tree(RRT) algorithm. Cindy and I worked on the implementation of RRT and we made sure we understood every detail of how the algorithm works in order for us to develop it from scratch. We learned the importance of knowing how to test our code with small test cases first, and later on with bigger ones to make sure it performs as expected. Visualization of is key when testing this kind of algorithms. We were also able to spot a lot of places in which our algorithm could be improved to increase its computational efficiency. Overall, I think that the best first step as a team during this class has been deciding how to split the work and see how we can support each other on our tasks; of course, always making sure everyone is comfortable with what they have to do.

5.5 Jose

I learned on yet again how important is for all modules to be working independently from the very start. There are some lingering issues with my controller in the areas in between the start and end trajectories where it is not sure on which waypoint to follow (especially near the end) and it starts to go in circles. However the entire car does complete a lap if placed closer to the start waypoint. It is incredibly important that everyone have their nodes working early for us to better integrate.