Lab 6 Report: Path Planning

Team 9

Nisarg Dharia
Zhenyang Chen
Meenakshi Singh
Kwadwo Yeboah-Asare Jr.

6.141/16.405 Robotics: Science and Systems

April 16, 2022

1 Introduction

Author: Meenakshi
Having a robust path planner is essential for autonomous systems to be able to navigate. Given a starting location A and a goal location B, an ideal path planner is able to determine if it is possible to navigate to the goal location. There are several metrics that are considered when evaluating the efficiency of generated path, such as the safety of the path, the total distance of the path, and the time it takes for the planner to generate a path.

In lab 6, we focus on the particular use case of planning paths in a known environment. Given a map, we want to be able to give the robot a start and goal destination, generate a path quickly and in real time and then follow the generated trajectory to reach that goal point. This process builds on some of the previous work we did in the localization and path-following labs. The racecar must be able to generate paths from its starting location, which we can determine using a particle-filter. We then use our pure pursuit controller to follow the generated path to target localization.

In this lab, we implemented two different algorithms to generate a path: search-based planning via A* and sample-based planning via Rapidly-Exploring Random Trees (RRT). Overall, we found that A* takes more time to generate a path as it explores more of the map, but conversely guarantees that it will return an optimal path if it exists. RRT on the other hand is faster in returning a path and is more efficient in larger state spaces, but may not return the optimal solution. Additionally, we found that the paths produced by A* were smoother than those produced by RRT and had a smaller path-following distance error. We tested the two algorithms for paths with short distances, medium length distances, and long distances at varying speeds and found that A* performs better for the use case of Lab 6, since we have a known map of Stata provided to us and it is possible to explore the whole map without excessive computational cost.

2 Technical Approach

2.1 Path Planning

The goal of path planning was to find a safe and efficient path to get the racecar from its current location to a goal position in a known map. To do this, we first processed the given map to make it easier to run path-finding algorithms on. Next, we converted our start and goal locations into the newly processed map's coordinate system. We then conducted either a search based algorithm known as A* or a sample based algorithm known as rapidly-exploring random trees (RRT) to find the desired path. Finally, we transformed the path coordinates back into the world frame for the racecar to follow.

2.1.1 Map Processing

Author: Nisarg
In order to implement a path planning algorithm, we first needed to process the map data into a format the algorithms were compatible with. To accomplish this, we converted the 1-dimensional array of map data into a 2-dimensional grid space. This allowed us to represent each location on the map as a set of (u, v) coordinates, which could then be used as vertices for the path planning algorithms.

Our next goal was to make the map safer, so that slight deviations in the car's position wouldn't cause it to come into contact with the walls. This was done by adding a small amount of padding to the occupied spaces of the map. For each coordinate on the map that was occupied, we set the values of all spaces within a distance d of the coordinate as occupied as well. As a result, the planned path would keep a barrier of distance d between itself and the real wall, allowing some buffer in case the car deviated slightly.

We also wanted to make sure that the racecar never wandered into any unexplored locations on the map. Thus, for any coordinate with a value of -1 to signify unknown, we converted it into a value of 100 to represent occupied. This ensured that the racecar never planned a path along a section of the map it didn't have enough information about.

Finally, in order to make search based algorithms less computationally expensive, we discretized our map to reduce the number of vertices we needed to explore. This was done by taking the average of each n x n square of points on the map, and checking whether it was greater than a specified occupancy threshold. If so, the square would be represented as occupied, and if not, it would be represented as open space. This was particularly useful given that the sample map of Stata basement contained 1,730 x 1,400 = 2,422,000 data points. Using this method of discretization allowed us to reduce the number of vertices by 1-2 orders of magnitude depending on the choice of n, with the new number

of vertices being

$$\|V\| = \frac{2,422,000}{n^2} \tag{1}$$

Note that for sampling based algorithms that don't require discretization or for maps with small passages where discretization may be limiting, setting n equal to 1 would simply leave the map in it's original level of granularity.

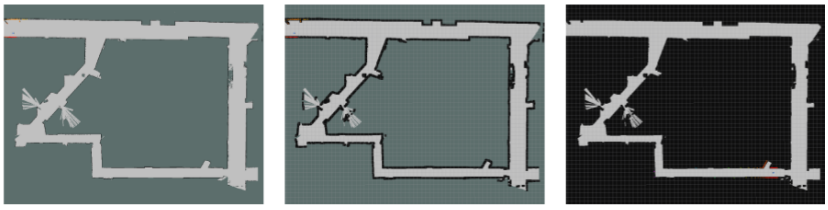

Figure 1: Illustration of the map before it was processed (left), after the walls were thickened by 0.25 meters (center), and after all unknown areas were marked as occupied (right)

2.1.2 Obtaining Start and Goal Locations

Author: Nisarg

Now that our map had been processed, we needed to convert the start and goal location from their (x, y) position in meters on the original map O to coordinates on the processed map P. To do this, we first took the orientation of O and used it to build the rotation matrix

$$R_O^P = \begin{bmatrix} \cos(\theta_O) & -\sin(\theta_O) \\ \sin(\theta_O) & \cos(\theta_O) \end{bmatrix} \tag{2}$$

From this, we then took the position we wished to convert (x, y), the rotation matrix above, and the translation of the map origin (x_O, y_O) to calculate the new position (x', y') in meters as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta_O) & -\sin(\theta_O) \\ \sin(\theta_O) & \cos(\theta_O) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_O \\ y_O \end{bmatrix} \tag{3}$$

To convert from meters to coordinates in the non-discretized map we simply divided (x', y') by the resolution size of the map. Similarly, to convert the resulting coordinates into the discretized map we divided the values again, this time by the discretization size n.

Lastly, the resulting coordinates are a transformation from (x, y) to (v, u) in the discretized map frame where v is the column and u is the row, so we can just swap the values to get a more standard (u, v) representation of our start or goal position in the processed map's coordinates.

2.1.3 Search Based Planning: A*

Author: Nisarg
Given the start and goal locations on the processed map, our first approach to path planning was to use a search based algorithm known as A*. The algorithm works by keeping track of the shortest path to each vertex, and exploring the paths in order of minimum distance traveled so far plus distance remaining to the goal. The pseudocode for the algorithm can be found below.

Algorithm 1 A* Algorithm

```
openSet ← Set(start)
cameFrom ← Map()
cost ← Map(start: 0, default: ∞)
score ← Map(start: dist(start, goal), default: ∞)
while openSet is not empty do
    current ← key of min(score)
    if current == goal then
        return reconstruct_path(cameFrom, current)
    end if
    openSet.remove(current)
    for each neighbor of current do
        tempCost ← cost[current] + dist(current, neighbor)
        if tempCost < cost[neighbor] then
            cameFrom[neighbor] ← current
            cost[neighbor] ← tempCost
            score[neighbor] ← tempCost + dist(neighbor, goal)
            if neighbor not in openSet then
                openSet.add(neighbor)
            end if
        end if
    end for
end while
return failure
```

For our implementation, we used each (u, v) coordinate in our discretized map as a vertex. Each vertex was neighbors with the 8 or fewer coordinates immediately adjacent to it, including diagonals. This made computations quicker and increased efficiency, but it came with the tradeoff that our paths could only extend in directions that were multiples of 45 degrees.

If a cell was unoccupied, the cost to reach is was simply the distance between two vertices. However, occupied cells had a cost to reach of ∞. Meanwhile, the heuristic was simply the Euclidean distance between the end of the path and the goal. Note that this choice of heuristic would never overestimate the remaining distance from the path end to the goal, and therefore ensured that any path returned from A* would be optimal.

Finally, it is important to note that A* is a resolution complete algorithm, which meant for a small enough discretization size n, the algorithm was guaranteed to find the optimal path. In our case, when working on the map of Stata basement with wall padding of 0.25 meters, this meant that for any discretization size of 6 or less, A* would always return the shortest path from start to goal when it existed, making the algorithm very reliable.

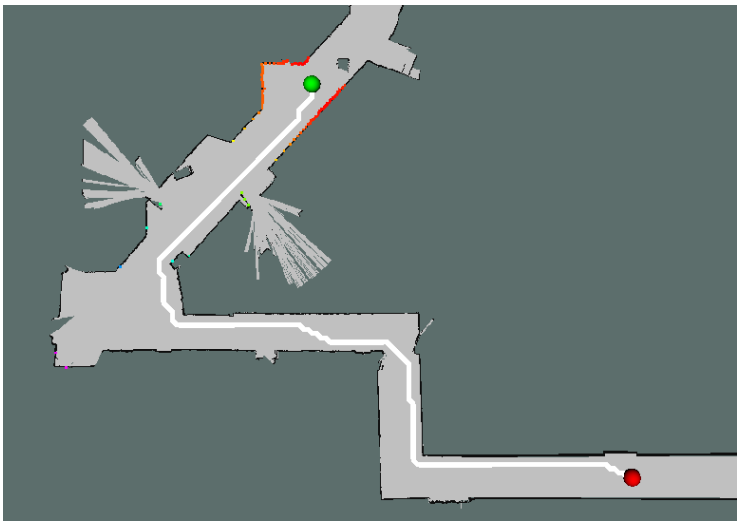

Figure 2: Example path planned by the A* algorithm in Stata basement, with a padding size of 0.25 meters. The green and red markers represent the start and goal respectively, while the white line represents the planned path.

2.1.4 Sample Based Planning: RRT

Author: Zhenyang
Though A* is an efficient algorithm which guarantees to find a optimal path given the map, it is hard to find a good heuristic distance function in high dimensional space, and solving A* in a very large state space can be a slow process (in worst case, the algorithm needs to go over the whole state space. To address the shortcomings of A*, we also implemented a sample based planning algorithm, rapidly-exploring random tree (RRT).

The basic idea of sample based planning is to randomly sample the points in state space. Based on the sampled points and space information (in our case, the map) we can try to find the feasible path with those sampled points. The property of randomly exploring makes sampled based algorithm possible to explore and know the big picture of environment faster than search-based algorithm,

especially in a large state space. The following code is the algorithm we used in our racecar.

Algorithm 2 Rapidly-exploring Random Tree Algorithm

```
tree ← ∅
start ← startcoordinate
goal ← targetcoordinate
K ← iterations
while N ≤ K do
    X_sample ← RandomSample
    L_nearest ← NearestLeaf(X_sample)
    X_new ← Steer(X_sample, L_nearest)
    if ObstacleFree(X_new, L_nearest) then
        tree ← X_new
        if X_new Closeto(goal) then
            return tree
        end if
    end if
end while
```

Here, we first randomly sample a feasible point in the map which can not be intercepted with obstacles. And then we will find the nearest leaf on the tree to the new sampled point. However, due to the randomness of choosing point, it is possible that we have a sample point far from the nearest leaf. If we simply connect the leaf and its parent together, it may result in frequency overlap of the path and obstacle and lower the efficiency of the algorithm. So we use "steer" function to determine how far the new leaf should be from the parent node. For simplicity, we fix the steering distance value which is tested in the simulation in the function. Finally, if nothing blocks the way of leaf and parent, we can add the new leaf to RRT tree. And return the path from goal to start point when we successfully sample a point near the target. Figure 3 demonstrates the path generated by RRT which is not as smooth as A*, but still a feasible path.

2.2 Pure Pursuit

There are two parts to a working pure pursuit controller. The first is setting the right speed and drive angles given a target location in the robots frame, and the second is actually finding what point on the path the robot should be targeting. This are completely separate tasks and can be designed as such.

2.2.1 Following a given target

Author: Kwadwo

To follow a target we need to use the relative position of the racecar and the target to set an arc for the car which will meet the target point. As the car gets closer to the path this relative vector changes and so does the cars desired

Figure 3: Example path planned by the RRT algorithm in Stata basement, with a padding size of 0.25 meters. The green and red markers represent the start and goal respectively, while the white line represents the planned path.

arc this allows the car to continually modify its trajectory based on where it is looking on the path. The distance between the car and the point on the path it see is known as the look ahead distance and this number is important for calculations. I will outline the particular math used to calculate the desired radius of the cars arc.

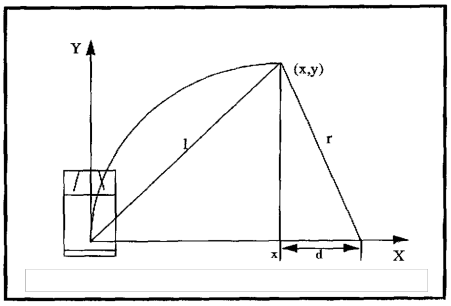

Figure 4: Geometry of pure pursuit problem

Figure 4 shows the geometry of the problem the car target point is place in the frame of the car and the horizontal distance of the car from the target point is used to calculate the radius of the arc required to meet that point.

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

l here is our look-ahead distance

$$x + d = r$$

r is the radius of our arc and d is the desired distance from the path, which in this case is 0
The next step is to solve for the radius of the circle that connects the car and the target point centered around the (x,0) in the car frame.

$$(t_r - x)^2 + y^2 = t_r^2$$

$$t_r$$

here is the turn radius, solvinng this gives us

$$t_r = \frac{l^2}{2x}$$

with this turn radius we solve for the desired drive angle using

$$angle = arctan(\frac{wheelbase}{t_r})$$

However for our robot specifically the axis are such that x maps unto -y and y maps unto x so we have to negate this angle to get the right value

$$angle = -arctan(\frac{wheelbase}{t_r})$$

This gives us the drive angle for a given target relative to the robot, here is a desmos of the process to play with and get a better understanding DriveAngleCalcs.com.

2.2.2 Finding target on the path

Author: Kwadwo
Given a target we now know how to follow it, the next step is to place some target on the path to follow. To achieve this we draw a circle around the car with radius of the lookahead distance we are using. This circle intersects with the path at some point and that becomes our target on the path to follow.
We then use the relative x position of the target in our pure pursuit. To find this intersection of the path line and circle we paremeterize the line between the start and end points of the line.

$$P = P1 + t(P2 - P1)$$

P1 being the start of the line and P2 being the end. This allows the variable T to represent some point of the. With this parametrization the line hits the circle when the distance between P and C the center of the circle is R.

$$P - C = R$$

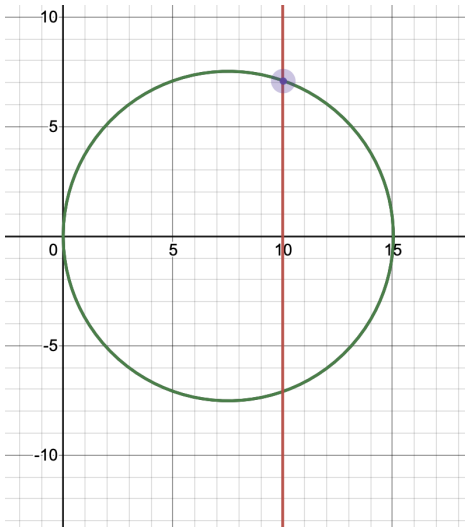

Figure 5: lookahead circle intersecting with path

We then solve for the magnitude of this vectors which gives us two values of t representing two intersections on the circle. Since the value of t goes from 0 to 1, 1 being the end of the line. We can pick the larger value and know the intersection is ahead of us on the path.
Code outline below.

```
P1 = point1
P2 = point2
Q = car_pose
r = self.lookahead
V = P2 - P1

a = np.dot(V,V)
b = 2*np.dot(V,P1-Q)
c = np.dot(P1,P1) + np.dot(Q,Q) - 2*np.dot(P1,Q) - r**2
disc = b**2 - 4 * a * c
if disc < 0:
    return None
sqrt_disc = np.sqrt(disc)
t1 = (-b + sqrt_disc) / (2 * a)
t2 = (-b - sqrt_disc) / (2 * a)
t = max(t1,t2)
target = P1 + t*V
```

This would work for one line segment and so we need to do this for whichever line segment we are closest to on the path. To find this we find the minimum

distance from every line segment on our path and use the segment with smallest distance as the path we are following.
Similar to the circle intersection we can parametrize the line between 0 and 1

$$P = P1 + t(P2 - P1)$$

with this we can solve for the value of P which which gives an angle of 90 degrees with the cars current location giving us the closest point to the line

$$P.(P3 - P1) = 0 \text{ P3 being the cars position}$$

This gives us a value of t which represents the closest point on the line to the car. However we want only closest points on the actual path and not just the extension of the line past the end and start point so we clip this t value between 0 and 1 and return it given us the closest point on the actual line. The minimum of these distances gives us the line segment to follow. Code outlined below.

```
path_points = self.path_points
last_but_index = path_points.shape[0]-2

P1 = path_points[:-1,:]
P2 = path_points[1:,:]
P3 = car_pose
LVEC = P2 - P1
norm = np.linalg.norm(LVEC,axis=1)
t = np.einsum('ij,ij->i',LVEC,P3-P1)/(norm**2)
# np.dot doesn't have an axis input so you have to do this wonky thing
t = np.clip(t,0,1)
# Limit measure of minimum distance to the actual line
min_points = P1 + t[:,np.newaxis]*LVEC
min_vecs = min_points - P3
min_dist = np.linalg.norm(min_vecs,axis=1)
start_ind = np.argmin(min_dist)
final_ind = start_ind + 1
```

There is one more step for the case of this problem. Because we are using a lookahead radius the right line segment to choose may not necessarily be the closest line segment. There may be situations where the path it is on is smaller than length than the look ahead radius so we should be following the next path. To fix this we pick the last point within our look ahead radius in the path ahead of us as the start of the point.
Last step is place the target relative to the car. To do this we use the relative angle between the car and the target point to place the target relative to the car.
We use the sin of this angle and the magnitude of the distance between the car and the target point to get the relative x which is what we need for our controller
This process finally gives us the x location of the target point in the car frame.

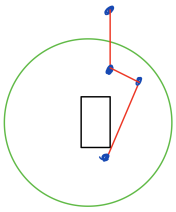

Figure 6: picture of possible problem

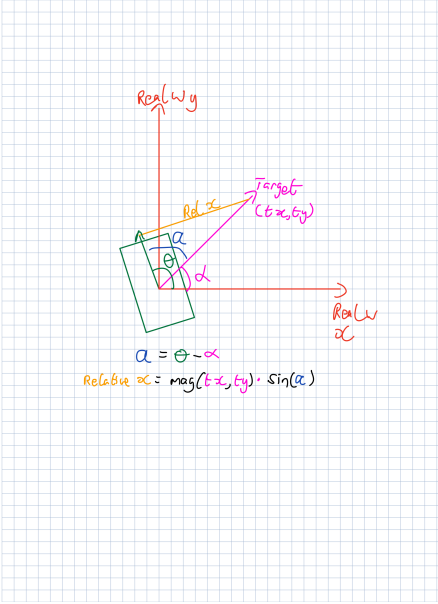

Figure 7: Placing the target in the frame of the car

3 Experimental Evaluation

3.1 A* Performance

Author: Nisarg
In order to test the performance of A*, we recorded the time it took to produce paths of various lengths given a variety of discretization sizes. For each path, we took 3 trials and recorded the data in figure 8.

As expected, when the map is not discretized at all (n=1), the time to plan long paths across Stata basement is prohibitively large for most applications. In comparison, a discretization size of 5 can still produce paths in under a second for some of the longest paths in Stata basement. Thus, as long as the paths are wide enough, an appropriately chosen discretization size can keep the runtime for A* reasonable. Additionally, the time to find a path with A* scales quadratically as path length increases. This means that while A* may be fine for short planning applications, longer ones may result in excessively long planning times.

Overall, we can see that A* provides good time performance for maps where the distances are short and the spaces are wide enough for large discretization sizes. However, in applications where the path is narrow or extremely large, the cost to run A* may make it unsuitable for time-sensitive planning tasks.

Path Length (Top) and Discretization Size (Left)	Short ($\tilde{1}$0m)	Medium ($\tilde{3}$0m)	Long ($\tilde{5}$0m)
1	0.444s	6.797s	56.912s
3	0.016s	0.360s	2.018s
5	0.008s	0.112s	0.544s

Figure 8: Average planning time (seconds) for A* on paths of various length with various discretization sizes

3.2 RRT Performance

Author: Zhenyang
We repeat the experiment process for RRT algorithm and recorded the average planning time for RRT. For each discretization size, we can see the planning time of RRT increases, but not too much. This makes sense because RRT randomly explores the environment which makes the difference of planning time for long and short distance not distinguishable, but longer path still requires more exploring efforts.

As discussed before, RRT will perform better when it comes to large state space. Here we can provide more detailed comparison with data support. When discretization size=1 (which means using the original map with size 1730*1400), we can see a huge difference between RRT and A*, where RRT 50 times faster

than A* when planning a long path. But after discretizing with box size=3, RRT has no significant time advantage compared to A*. So we can say, based on the computational platform we have on the racecar, the two algorithms we implemented show a difference when the scale of state space reach 10^6. If the scale of the planning problem is smaller than this value(short path in large space or in small space), we prefer using A* which has less time cost and always find optimal solution. But when the problem scale is larger than this, RRT is definitely a more reasonable choice for planning and will reduce the time cost significantly.

Path Length (Top) and Discretization Size (Left)	Short ($\tilde{1}$0m)	Medium ($\tilde{3}$0m)	Long ($\tilde{5}$0m)
1	0.289s	0.722s	0.922s
3	0.560s	1.241s	1.969s
5	0.665s	1.076s	2.306s

Figure 9: Average planning time (seconds) for RRT on paths of various length with various discretization sizes

3.3 Path Following Error

Author: Meenakshi

Another metric we used to evaluate the performance of the paths generated by A* and RRT was the path following error. We calculated the minimum distance between the goal path trajectory to follow and the ground truth position of the racecar at various speeds. This error also indicates the "smoothness" of the generated paths.

Speed	1 m/s	2 m/s	3 m/s	4 m/s
A* Average Error	0.081m	0.079m	0.073m	0.086m
RRT Average Error	0.127m	0.118m	0.136m	0.122m

Figure 10: Average minimum distance between the path to follow and the ground truth position of the racecar on paths generated by A* and RRT at various speeds

Speed	1 m/s	2 m/s	3 m/s	4 m/s
A* Maximum Error	0.560m	0.561m	0.566m	0.727m
RRT Maximum Error	0.611m	0.711m	0.722m	0.669m

Figure 11: Maximum distance between the path to follow and the ground truth position of the racecar on paths generated by A* and RRT at various speeds

We conducted trials at different speeds and calculated both the average minimum distance error (Figure 10) as well as the maximum distance error for

each algorithm (Figure 11). Notably, the largest distance errors occur at sharp turns, where the angle of the turn was less than 90 degrees. Overall, we see that for both average error and maximum error, A* had consistently lower distance error, indicating that the paths generated by A* were smoother and that the racecar was able to follow the path more closely.

4 Conclusion

Author: Meenakshi
Overall, we found that when comparing our current implementations of A* and RRT, A* generated optimal paths for the map under 1 second for all three path lengths that we tested when using a discretization size of 5. Meanwhile RRT paths were generated much faster for discretization sizes that were smaller, due to the larger state space that is explored in A* to guarantee an optimal path versus the randomly sampled points used by RRT.

The racecar also had a higher distance error following the paths generated by RRT as opposed to A*, due to the shape of the paths generated by RRT being less smooth. As a result, there was more of an oscillatory path where the racecar tended to overshoot the turns, resulting in a higher path following error for RRT.

In the future, we want to optimize both of these algorithms further. For the pure pursuit controller, we plan to adjust the lookahead distance to account for curvatures in the path. Additionally, we want to try testing different heuristics for A*, such as Dubins curves, which provide a better representation of paths for non-holonomic motion. We also want to test using different data structures for A* to improve runtime of the algorithm. For RRT, we need to tune parameters to improve performance going forward. We also plan to implement RRT*, which is an extension of RRT that returns the optimal path.

While there is room for improvement in optimizing our planning algorithm and path following, we were able to successfully generate paths of varying distances using two path planning algorithms. We then integrated these generated paths with our localization and pure-pursuit controller to navigate the known map at different speeds and evaluated their performance according to time taken to plan the path and the smoothness of the path.

5 Lessons Learned

5.1 Meenakshi

This lab helped me learn about the usefulness of path planning algorithms and how they are applicable to different situations. I was able to understand the trade off between computational expense and correctness through our tests of A* and RRT. I also learned more about how our work from previous labs integrates and fits into the flow of controls.

From a communication standpoint, this lab helped me ease back into the flow of things after I had been sick in the last lab. Because we were making up part of Lab 5, it was a little tricky to balance both at the same time, but I am grateful for how my team was able to communicate and handle everything. Even though I still feel like I wasn't able to contribute as much technically, I learned that through being present and available for the team, I could still learn a lot about the lab.

5.2 Zhenyang

As described on the GitHub page, this lab is probably the most rewarding part of 6.141 so far. We implemented and compared the pros and cons of two useful planning algorithms. A* demo a good performance in finding optimal result in small state space, while RRT shows more efficient behavior in larger space. But besides that, what intrigues me is knowing how we can build this complicated and powerful racecar control system step by step and integrate everything together. Having a clear thoughts on the process and understand the key details in every algorithms, in the whole system is essential. This is the challenging part in engineering as well as the charming part. I hope I can keep learning to be a better system engineer in the final challenge.

For the communication part, I am very happy that we have a smart and reliable new teammate Nisarg joining us and Meenu gets well and can work with us again. After having a tough week for lab5, we now have a more powerful team in which we can work together to tackle challenges. It is good to see we build the team dynamic again, and hope we can keep this momentum and move on.

5.3 Nisarg

On the technical side, this lab was very helpful in learning about the pros and cons of various path planning algorithms. In particular, I was able to understand how sample based algorithms can exhibit better performance than search based one, but at the cost of correctness and reliability. I also learned how choices of discretization lead to resolution complete algorithms, while properties of randomness leads to probabilistically complete algorithms.

From a communication perspective, this lab was quite unique in that I had to join a new team. Through this, I gained a new perspective on how to approach a lab in general, and also learned how to work with the different skill sets of my teammates. Overall, this lab was great for integrating the best habits of my old team with my new one to get the best of both worlds.

5.4 Kwadwo

This lab sort off settles some things I suspected with the last lab, that a full team makes work much easier. As I had a significantly easier time with this lab. I also improved my ability to explain code and math (In my opinion), I think I did a pretty good job of that.

On the technical side of things I learnt a lot about vectorizing math with numpy to perform actions on many points at once. I gained a better understand of moving between the world frame and the robot frame. It might be useful in future to have some function for that since we seem to do that much more than I expected. Finally, I realized a simpler solution to problems works much better especially for scale. My initial solution to target placing involved fitting a line to the path and solving the intersection, which was iffy and had no benefits over the current implementation.