Lab 6 Report: Path Planning

Team 13

Ra Mour (Ronald Alvarez)
Srinath Mahankali
Nandini Thakur
Jorge Tomaylla
Sarah Zhang

6.4200: RSS

April 26, 2023

1 Introduction

Author: Nandini Thakur

We've started seeing many autonomous vehicles in the real world complete navigation from one point to another, full autonomously. Companies such as Tesla see this as the next important step in autonomy - to get a human to their destination while maintaining the right speed and dodging cars, people, and other obstacles.

This lab had many similar requirements. An optimal path needed to be generated from start to end without getting too close to any obstacles, and the racecar had to traverse this path accurately and ideally at a high speed.

Both pure pursuit and localization are algorithms implemented in previous labs, which were important for the success of this lab. The localization algorithm from the previous lab provides the current estimated location of the car so that the car can know where along the path it is, and thus where it should go next. The pure pursuit algorithm used for line following was slightly modified to fit the requirements of this lab.

We split the work into two modules - the path planning module and the path following module. The path planning module receives the location of the robot as the start point and a destination point, and it runs the RRT* algorithm to find an optimal path between the points. The generated path is guaranteed to not intersect with any walls and to always be at least 0.75 meters away from the wall. The path following module subscribes to the topic sending the path, as well as to the topic where localization publishes an estimated pose. The algorithm finds a lookahead point on the path, which is a point a fixed distance away from the car, and directs the car to drive to that point. This is a very similar implementation to the pure pursuit technique used previously.

Both of these modules will be discussed in further detail, as well as the evaluation criteria and results we achieved after testing our implementation.

2 Technical Approach

2.1 Path Planning

Author: Sarah Zhang

We chose to implement the rapidly-exploring random tree (RRT) algorithm and its improved version, RRT*, to execute path planning. RRT is a sampling-based method that randomly samples points in the feasible space to add to a tree, and the tree expands away from the start pose until an obstacle-free path to the desired goal pose is attainable. RRT* improves upon RRT by using additional logic to ensure new nodes are connected to a point in the tree with the shortest path to the node, which produces smoother, more optimal final trajectories. RRT is probabilistically complete and is one of the more efficient planning algorithms due to its random sampling nature.

We use ROS with Python 2.7 to implement RRT and RRT*. To represent the tree, we use a dictionary with the index tuple of child nodes as keys and its parent node index as values. The map was represented as an OccupancyGrid message. The pseudocode for our RRT* implementation is as follows. (The RRT pseudocode is the same except without lines 17-27 and lines 33-39.)

```
    procedure PLANPATH-RRT(startpose, endpose, step_size, neighbor_radius)
2:      // Initialize tree structure
        goal_reached_flag ← False
4:      tree[startpose] ← None
        costs[startpose] ← 0
6:      if path-collision-check(startpose, endpose) then
            tree[endpose] ← startpose
8:          goal_reached_flag ← True
            return
10:     else
            while not goal_reached_flag do
12:             node_sampled ← sample-map()
                node_nearest ← find-nearest-vertex(node_sampled)
```

```
14:             node_new ← calculate-new-node(node_sampled, node_nearest, step_size)

16:             if not path-collision(node_new, node_nearest) then
                    // Find most optimal, e.g. minimum cost, neighbor
18:                 neighbors ← find-neighbors-within-radius(node_new, neighbor_radius)
                    node_min ← node_nearest
20:                 cost_min ← costs[node_nearest] + neighbors[node_nearest]
                    for neighbor, neighbor_dist in neighbors do
22:                     neighbor_cost ← costs[neighbor] + neighbor_dist
                        if neighbor_cost < cost_min then
24:                         node_min ← neighbor
                            code_min ← neighbor_cost
26:                     end if
                    end for
28:
                    // Connect new node with most optimal neighbor
30:                 tree[node_new] ← node_min
                    costs[node_new] ← cost_min
32:
                    // Rewire the tree based on the newest edge
34:                 for neighbor, neighbor_dist in neighbors do
                        if costs[node_new]+neighbor_dist < costs[neighbor] then
36:                         // Replace neighbor's parent with new node
                            tree[neighbor] ← node_new
38:                     end if
                    end for
40:
                    // Check end condition
42:                 if dist(node_new, endpose) ≤ step_size and not path-collision(node_new, endpose) then
                        goal_reached_flag ← True
44:                     tree[endpose] ← node_new // update tree
                        return
46:                 end if
                end if
48:         end while
        end if
50: end procedure
```

The first thing we do is check if a collision-free path exists between the *startpose* and *endpose*, since if this exists then there is no need to run the algorithm. The way we check for collisions is by interpolating points frequently along the line between endpoints, and checking each of these points against the Occupancy-Grid map to see if any overlap with a known obstacle. If there is any collision, we will then proceed to build the RRT (lines 11-49).

In the RRT and RRT* algorithm, the first step is to create a new node by randomly sample a point from anywhere in the feasible region of the map (line 10, *sample-map()*), find the nearest vertex by measure of Euclidean distance to the newly sampled point that exists in the tree (line 11, *find-nearest-vertex()*), and set the new node to be the point that is at most a given stepsize away from the nearest vertex in the direction of the sampled point (line 12, *calculate-new-node*). We sampled points via sampling pixels from the map, which have a very fine resolution of .05 m x .05 m. If the line between this new node and the nearest node is collision free, we proceed to add it to the tree.

In RRT, we simply connect the new node to the previously identified nearest vertex (line 29). In RRT* however, we do a bit more work to determine the most optimal vertex to connect to, where optimality is defined as having the shortest path from *startpose* to the new node. This is done by identifying a set of vertices that are within a given radius of the new node and finding the one that would give the shortest path to the new node (lines 16-26).

RRT* also includes a post-processing step which consists of rewiring the tree to improve paths based on the newest node (lines 32-38). Specifically, this means iterating through the neighbors again and checking if the path that can be formed by concatenating the path from the *startpose*, through the new node, and then to the neighbor node is shorter than the existing path to the neighbor node - if so, the neighbor node's parent node is reassigned to be the new node.

Lastly, the end condition is defined by being within one step size of the goal pose and the line between the new node and goal pose being collision-free (lines 40-45). This function builds the tree, and once finished, we simply recurse up through the tree starting from the *endpose* to construct the desired trajectory.

We also apply dilation on the given map with OpenCV2 to address the fact that our planner treats the car as a single point while in real life, the robot's size limits where it can go. Dilation makes all the walls "thicker" and effectively reduces the feasible space our planner works with - after dilation, anywhere within 0.75 meters from a wall is considered an obstacle.

2.2 Pure Pursuit

Author: Srinath Mahankali

After a trajectory for the car to follow is determined using path planning, the low-level control of the car is performed using our pure pursuit approach to have the car actually follow this trajectory. At a high level, our pure pursuit approach first determines a point in front of the car which lies on the trajectory which the car should follow. After this, the car drives towards this point using PD control. This process repeats, with the point that the car must follow continuously moving towards the end of the trajectory. Now, we describe each of these steps in detail.

2.2.1 Finding the Lookahead Point

The first step in our approach to having the car follow the trajectory is to determine the "lookahead point," which is the point the car must drive towards.

Finding the Closest Trajectory Segment To find this lookahead point, we assume that the trajectory is piecewise linear, i.e. composed of several connected line segments in sequence and find the segment closest to the car's current position. To do this, we first compute the point on the trajectory closest to the car's position.

We do this by computing the closest point on each trajectory segment to the car. We rely on the formula for projecting a point onto a line. For a point p_{robot} representing the robot's position and points p_{start}, p_{end} representing the start and end points of the trajectory segment, we define

$$t = \text{clip}\left(\frac{(p_{robot} - p_{start}) \cdot (p_{end} - p_{start})}{\|p_{end} - p_{start}\|^2}, 0, 1\right).$$

The variable t represents how close the projection p_{proj} of p_{robot} onto the line segment is to p_{start} compared to p_{end}. For example, if $t = 0$, then $p_{proj} = p_{start}$ and if $t = 1$, then $p_{proj} = p_{end}$. Thus, the projection onto the line segment is

$$p_{proj} = p_{start} + t \cdot (p_{end} - p_{start}).$$

We calculate p_{proj} for each trajectory segment and we set the closest trajectory segment to be the segment containing the closest projection point, where ties are broken arbitrarily.

Determining the Lookahead Point Now that the closest trajectory segment has been calculated, we can determine the point that the car must drive towards, or the lookahead point. To do this, we set a parameter ℓ to be the desired distance, in meters, that the lookahead point should be away from the car, and we aim to find a point $p_{lookahead}$ such that $p_{lookahead}$ is on the trajectory and is a distance ℓ away from the car. We rely on the formula for the intersection between a circle and a line to compute the points on the trajectory that are a distance ℓ from the car in the following code snippet:

```
V = end_point - start_point  # Vector along line segment

# Quadratic equation coefficients
a = V.dot(V)
b = 2 * V.dot(start_point - robot_position)
c = (start_point.dot(start_point)
    + robot_position.dot(robot_position)
    - 2 * start_point.dot(robot_position)
    - self.lookahead**2)

disc = b**2 - 4 * a * c
if disc < 0: # Quadratic has no real solutions
    return False, None

sqrt_disc = np.sqrt(disc)
t = (-b + sqrt_disc) / (2 * a)

p1 = start_point + t * V

# Prioritize point closer to endpoint than startpoint
if 0 <= t <= 1:
    return True, p1
else:
    return False, None
```

This code snippet returns two values: a boolean representing whether a given current trajectory segment contains a point that satisfies the condition of being a distance ℓ from the car, along with the point itself (or None if it does not exist). We also only return the point that is closer to the endpoint than the start point of the trajectory because the car should be moving towards the endpoint of the trajectory. Finally, we require that $0 \leq t \leq 1$ in the code above to return a point because otherwise, it means the intersection of the circle and the line lies outside of the segment. We run this code snippet to find the intersection of the circle of radius ℓ centered at the robot's position starting at the trajectory segment closest to the robot, which we previously found. Finally, we iterate through the trajectory segments starting from there until a lookahead point is found.

2.2.2 Controlling the Car with PD Control

Now that we have determined the lookahead point $p_{lookahead}$, the car can now drive towards $p_{lookahead}$ from its original position p_{robot} in a similar fashion as the line-following lab. Specifically, we determine the relative coordinates of the lookahead point in the world frame rather than the robot frame, and calculate the angle between the car and the lookahead point using the formula `angle= np.arctan2(self.relative_y,self.relative_x)`. We treat this angle as the error signal for our PD control, and since we use PD control, we also determine the derivative of the angle as follows:

```
current_time = rospy.get_time()
angle_derivative = (angle - self.prev_angle)/(current_time - self.prev_time)
```

Finally, we use the formula for PD control to determine the steering angle:

```
steering_angle = (self.steering_kp * angle
    + self.steering_kd*angle_derivative)
```

We set the velocity of the car to a constant multiplied by the distance between the car and the lookahead point. Since the lookahead point is designed to be a distance ℓ away from the car, this distance is always approximately ℓ. Finally, the lookahead point is constantly recalculated as the car follows the trajectory, updating to be a distance ℓ away in front of the car, allowing the car to fully drive through the trajectory.

2.3 Integration

Author: Nandini Thakur

Integrating the two modules was very simple. We ran the path planning algorithm to publish a trajectory instead of the script which loaded a hand drawn trajectory. The pure pursuit algorithm was already subscribed to the correct topic to receive a path. From there, the path following code allowed the simulated car to follow the path perfectly. We were unable to test the integration on the real racecar due to issues with */map* not existing, but we plan to revise and test on the car as we move into the final challenge.

3 Experimental Evaluation

3.1 Path Planning

Author: Ra Mour

We evaluated RRT and RRT* by measuring the average path length and runtime at several locations in the Stata basement map.

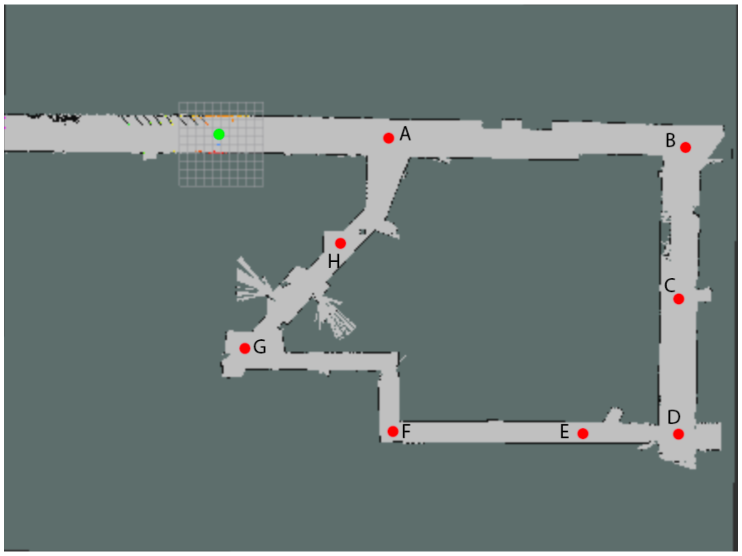


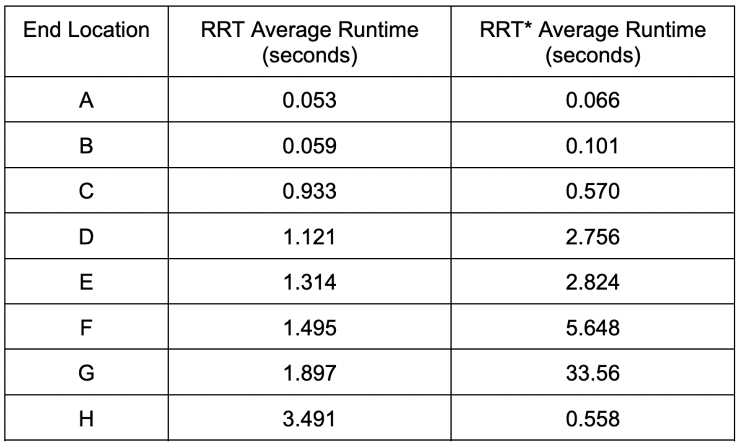
Figure 1: Green: Start location. Red: End locations for average runtime and path length evaluation.

The path lengths for RRT* were consistently and significantly shorter, except for endpoints A and B. For these locations, a straight line segment is selected as the path in both RRT and RRT*.

End Location	RRT Average Path Length (meters)	RRT* Average Path Length (meters)
A	20.97	20.63
B	55.70	55.70
C	74.70	74.23
D	90.46	87.69
E	108.57	101.97
F	99.93	69.363
G	81.26	55.51
H	75.99	34.11

Figure 2: Average path length for RRT and RRT*.

The runtimes are comparable for endpoints that are closer to the start, but RRT* takes significantly longer as the endpoints are further away or less directly reachable from the start.



End Location	RRT Average Runtime (seconds)	RRT* Average Runtime (seconds)
A	0.053	0.066
B	0.059	0.101
C	0.933	0.570
D	1.121	2.756
E	1.314	2.824
F	1.495	5.648
G	1.897	33.56
H	3.491	0.558

Figure 3: Average runtime for RRT and RRT*.

3.2 Pure Pursuit

Author: Nandini Thakur

The pure pursuit algorithm was evaluated by finding the minimum distance from the racecar to the trajectory, which was already being calculated as part of the algorithm itself. Zero error equated to the racecar being perfectly on the trajectory, and therefore the minimum distance from the car to the trajectory was 0.

3.2.1 Simulation

We first evaluated in simulation, running on the loop2 pre-built trajectory. The following tests were run at 1m/s.

Velocity = 1m/s, Lookahead Distance = 1m, $K_P = 1$, $K_D = 0.13$:

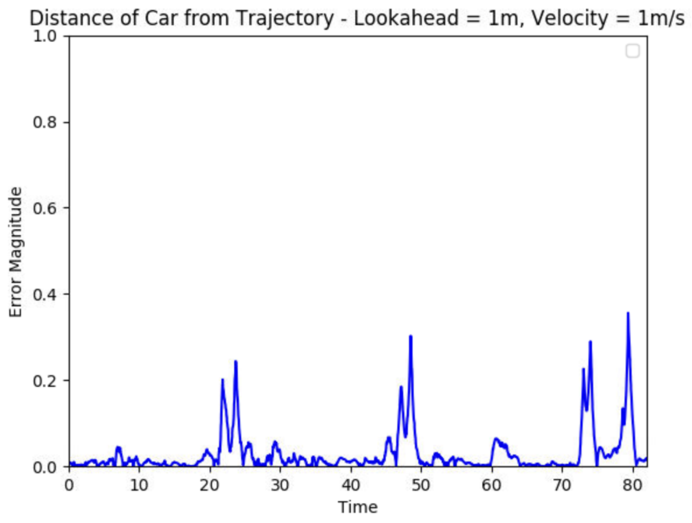


Figure 4: Error spikes considerably at turns with a high lookahead distance. Average error = 0.0289m

Velocity = 1m/s, Lookahead Distance = 0.7m, $K_P = 1$, $K_D = 0.13$:

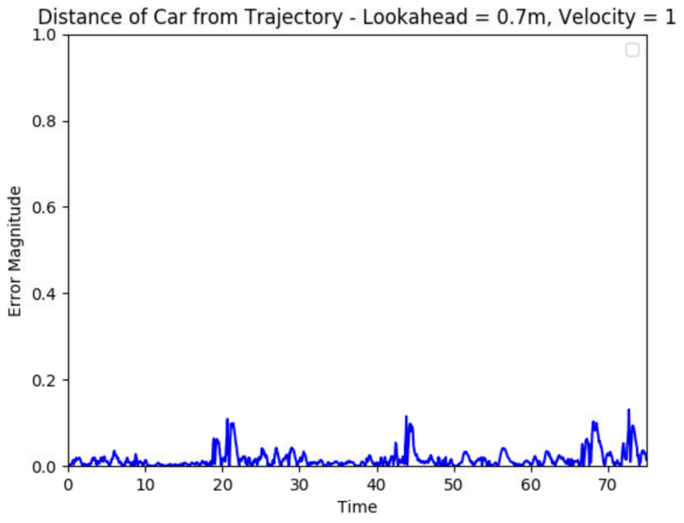


Figure 5: Very minute spikes at turns when lookahead distance is decreased. Average error = 0.0159m

In this test we observed the effects of changing the lookahead distance. We observed that lookahead distance is integral to having small error at turns. If the lookahead distance is too high then the racecar will effectively be following a point that is many steps ahead of it, and will therefore do all actions, such as turns, earlier than they should be. The lookahead distance was tuned for the velocity of 1m/s to be close enough such that the car didn't turn early, and we settled on a value of 0.7m at this velocity with $K_P = 1$ and $K_D = 0.13$.

We wanted to increase the speed of driving to finish the loop in a small amount of time while still being pretty accurate. The following tests were run at 2m/s with a lookahead distance of 1.2m. The lookahead distance was increased for these tests because the car is faster, so there is a possibility that noise in the localization could cause the lookahead point to end up behind the car, which would mess up the whole test.

Velocity = 2m/s, Lookahead Distance = 1.2m, $K_P = 1$, $K_D = 0.13$:

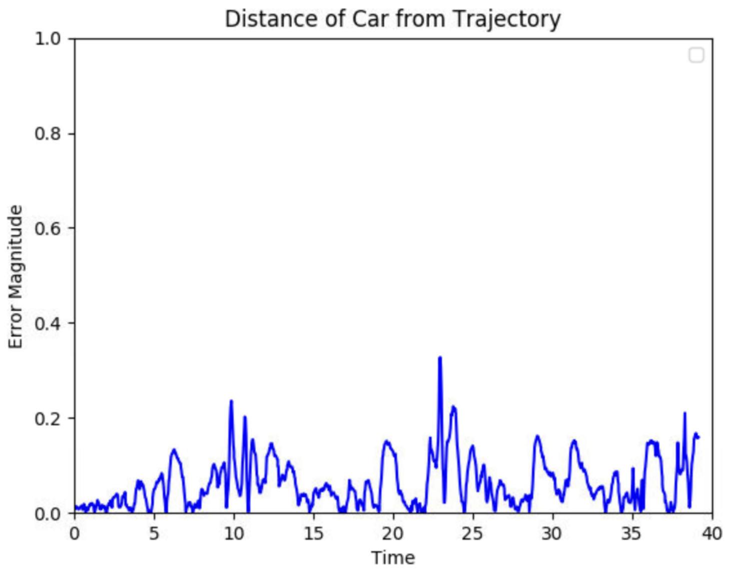


Figure 6: Large spikes at turns and generally higher error throughout path. Average error = 0.0626m

Velocity = 2m/s, Lookahead Distance = 1.2m, $K_P = 1.4$, $K_D = 0.135$:

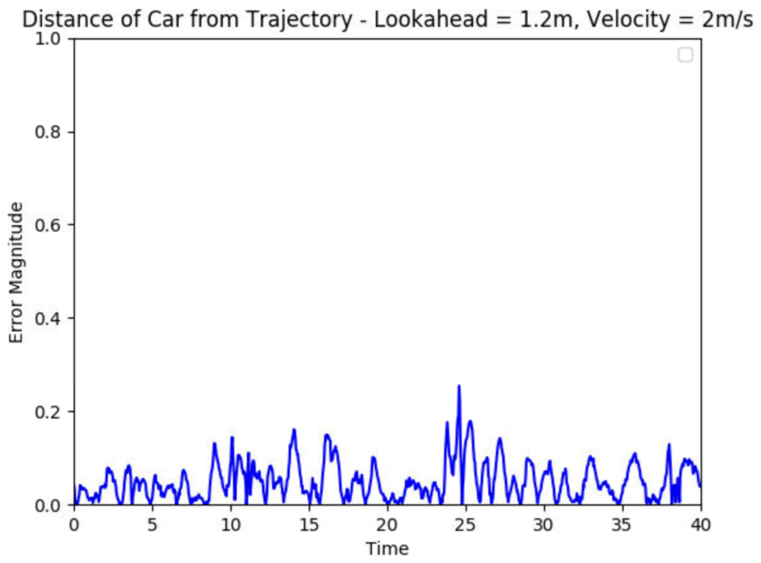


Figure 7: Smaller spikes at turns and more uniform and smaller error throughout path. Average error = 0.0500m

In this test we observed the effects of tuning the K_P and K_D constants. We found that increasing the K_P value for the pure pursuit controller helped the car follow the trajectory better, as expected. At higher speeds the error was generally higher than when the car traveled at lower speeds. This was expected due to the fact that this entire algorithm depended on accurate localization. The error value being calculated used the pose found from localization as part of the calculations, so an inaccurate localization would be a big reason for seeing high error. Here, we see that the average error is pretty low - 5cm - but overall the error chart looks very jittery. We determined that this was because of the noise added to our localization algorithm, which caused the car's predicted position to change in minute ways on every time step. This resulted in the error chart also having very rapid changes, even though visually the movement of the car looks very smooth.

We decided that this amount of error was acceptable since it was largely attributed to localization noise, and qualitatively the car looked like it was following the path well. We concluded with $K_P = 1.4$ and $K_D = 0.135$ at a velocity of 2m/s.

3.2.2 Racecar

Once the pure pursuit algorithm was working with very low error in simulation, we started testing and evaluating on the racecar.

Velocity = 2m/s, Lookahead Distance = 1m:

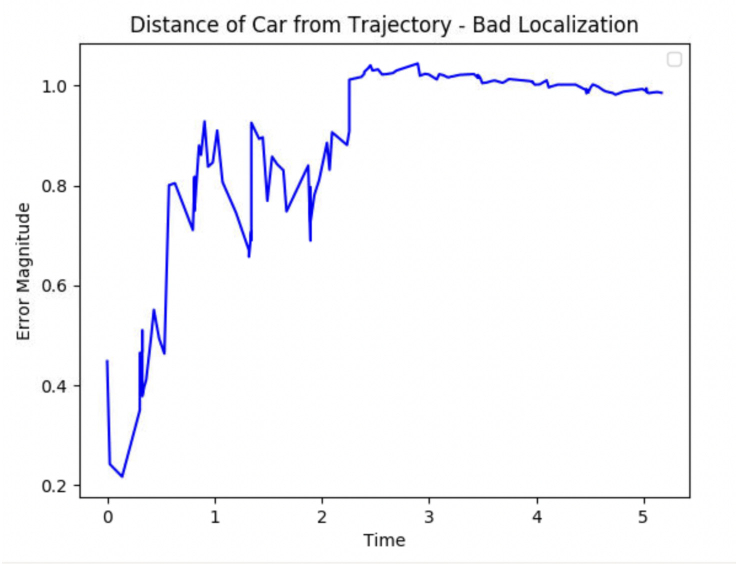


Figure 8: Really high error for a small straight section of driving. Average error = 0.882m

In this test we noticed how high of an error we were getting quantitatively, and even qualitatively we could see a lot of jitteriness in the robot's motion. We once again determined that localization was the main issue that needed to be fixed here because the change in estimated robot position between timesteps was too big. This was fixed by resampling particles such that the particles remained much closer to the robot rather than being spread apart.

Velocity = 2m/s, Lookahead Distance = 1m:

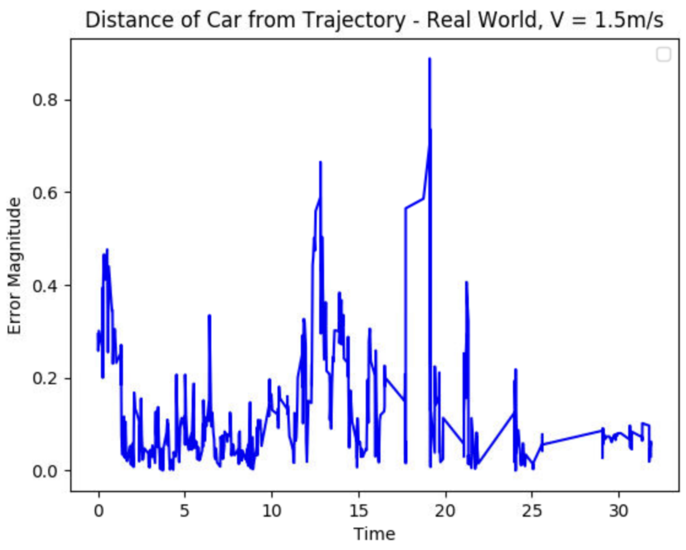


Figure 9: Smaller spikes at turns and more uniform and smaller error throughout path. Average error = 0.1367m

After fixing our localization and tuning K_P and K_D values to work better on the racecar, the above plot was error while the car completed the course from start to finish. While the error still seems high in some places, such as the spike going to around 0.9m, these only were seen at the turns, where it's likely that localization is still slightly inaccurate. Visually, the car drove very smoothly along the path, and only a little bit of side-to-side oscillation was observed. We ended testing at an average error of 0.1367m with $K_P = 0.2$ and $K_D = 0.075$. We expect that these values can be improved with further time for testing during the final portion of the course.

4 Conclusion

Author: Jorge Tomaylla
In this lab we have implemented the final module in RSS that makes up autonomous driving: navigation and path planning. Path planning is essential in successfully moving from a starting location to a goal pose while avoiding obstacles, a feature that is needed for user safety and time effectiveness. We accomplished this by dividing the tasks into a path planning team in charge implementing the RRT algorithm, and a pure pursuit team to trigger the car to follow the generated trajectory. We later upgraded the algorithm to RRT*, combining some elements of RRT and A* and significantly improving the quality

of optimal generated paths as well as enlarging the scope of possible trajectories generated anywhere in the map. With this robust path planning feature, our race car now has the major characteristics of an autonomous vehicle that will succeed in the complex tasks of the final challenge.

5 Lessons Learned

5.1 Ra Mour

Lab 6 gave us the opportunity to try an unfamiliar, sampling-based path-planning algorithm. After writing the skeleton code and implementing several helper functions we were able to visualize paths, but we ran into a collision-checking glitch. We struggled to identify the source of the issue until another team member looked at the code, then several components of the planner followed smoothly. This pointed out how useful it is to have someone look at code with fresh eyes.

5.2 Srinath Mahankali

This lab reinforced the importance of collaboration and working in parallel to accomplish different tasks. Through this lab, I learned that it is important to test whatever modules are completed, even before the full lab is implemented. Along with Nandini, I implemented the pure pursuit controller. We tested the performance of our controller as soon as possible on the real racecar rather than in simulation, before path planning was implemented, which allowed us to have pure pursuit results ready for the briefing.

5.3 Nandini Thakur

In this lab I learned how helpful it is to start evaluating early. From the first couple rounds of testing, I recorded error data, rosbags, and videos of the test, and this really helped put our briefing and report together with data showing where we started and where we concluded. I also learned how to be more efficient in collaboration by splitting up work and giving bigger tasks to people with more time available to commit to the class. This ensured that we didn't fall behind with implementation and integration.

5.4 Jorge Tomaylla

On the technical aspect, this lab taught me the benefits and disadvantages of incorporating different path planning algorithms. On one hand, probabilistic-based algorithms like RRT do not guarantee making the most optimal trajectory, but it does promise computing a good trajectory in a rapid way. Other algorithms like A* guarantee returning the most optimal path at the price of great computational time. I think our final design to mix the best of both worlds was very efficient and truly heightens the need to thoroughly weight options and take the best out of them in final implementations.

5.5 Sarah Zhang

The biggest lesson I learned in this lab was the importance of starting work early and doing continuous testing and evaluation (as opposed to waiting until the entire system is fully integrated to begin testing). I also learned that sometimes splitting up work too much is disadvantageous, as one of our first goals was to implement A* and RRT* side by side, but after running into roadblocks in RRT that prevented us from progressing fast enough to meet our timeline, we decided to abandon A* and refocus the entire effort on RRT. Overall, I really enjoyed the technical challenge of implementing RRT and RRT* in this lab.