6.4200/16.405 SP23 - Lab #6 Report: Executing Optimal Race Car Trajectories with Search-based Motion Planning and Pure Pursuit

Team #20

Aaron Zhu
Jessica Rutledge
Chuyue Tang
Nihal Simha
Shara Bhuiyan

6.4200/16.405 Robotics: Science and Systems (RSS)

April 27, 2023

1 Introduction (Aaron)

Motion planning is an essential aspect of robotics. With applications ranging from navigation to joint movement, motion planning allows for systems to autonomously achieve complex and abstract tasks by discretizing a desired trajectory into a sequence of actions.

However, motion planning presents various challenges. Not only must the calculated trajectory be feasible (i.e. avoids obstacles), but it should be optimal. The goal should be reached in an efficient manner as to reduce time and power resources expended. Ensuring optimality is already a challenge in itself, but its existing trade-off with computational time makes it even more challenging. Algorithms that guarantee more optimal paths come at the cost of needing longer to be computed. Although differences could be on the order of fractions of a second, such seemingly negligible differences could be significant in systems that need to react spontaneously to the dynamic real world. Last but not least, challenges also extend beyond the planning stages to the execution stages. Even with an optimal path constructed, enabling the system to precisely follow the given path is difficult. Deviations from the optimal trajectory can often result in serious safety implications in the real world.

In this lab, we build upon our previous work with localization to create two additional modules (A* motion planner and pure pursuit controller) which are integrated together to construct our path planner. Given a start and end pose on a map, the racecar is able to plan and execute the optimal trajectory to safely arrive at the predetermined destination.

This paper demonstrates the following contributions:

- Through a detailed discussion comparing sampling-based algorithms and search-based algorithms, we justify our decision to implement an A* algorithm over a Rapidly-exploring Random Trees (RRT) algorithm. We demonstrate how our A* algorithm calculates an optimal trajectory between any two points, appropriate for the purposes of this lab.
- By taking advantage of our trajectory representation, we show how our pure pursuit controller allows our race car to execute the optimal path with relatively high precision.
- After integrating our motion planning and pure pursuit modules, we demonstrate through experimental evaluations in simulation and on the physical race car that our resulting path planner can successfully plan for and safely execute optimal trajectories.

We detail our technical implementations of the lab in Section 2 and evaluate our experimental results in Section 3. We briefly discuss potential areas for future work in Section 4 before concluding in Section 5 and sharing our lessons learned in Section 6. Videos and images associated with this lab are available at: here.

2 Technical Approach (edited by Aaron)

In this section, we detail the various modules that comprise our integrated path planner. We begin with motion planning, where its objective is to define a sequence of actions that constructs a continuous, obstacle-free path between two points. We discuss the general principles of search-based and sample-based planning algorithms and describe how A* and Rapidly-exploring Random Trees (an example of each, respectively) are implemented. A more in-depth comparison of the two types of algorithms is elaborated on in the Experimental Evaluation section (see Section 3.1). Next, we describe the design of our pure pursuit module. At a high level, pure pursuit takes advantage of the path representation to determine a target point to follow on the path, which is then used to calculate the driving angle needed to reach it head-on. Finally, a general illustration of how the modules are integrated together to form the path planner, as well as the dilation techniques used to ensure safer paths, is included.

2.1 Search-based Motion Planning (Chuyue)

Search-based algorithms essentially take the search problem as an input and return either a solution or an indication of failure. They expand a search tree rooted in the initial state. Each node can be expanded by considering all applicable actions in that state and generating child nodes for the resulting states. If the child states are newly reached, or the states are reached by a lower cost, these nodes will be pushed to a frontier. Nodes in the frontier will be ranked by an evaluation function f, which is the essence of search algorithms, and the node with the highest priority will be first popped out from the frontier to be expanded in the next round. Figure 1 defines our path planning problem as a general search problem.

General Search Problem: $< \mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I}, \mathcal{G} >$

state space: $\mathcal{S} = \{(x, y) | 0 \leq x < 1300, 0 \leq y < 1730\}$

obstacle mapping: $\mathcal{O} : \mathcal{S} \to \mathcal{R}$

$$\mathcal{O}(s) = \begin{cases} 100, & \text{if this is the obstacle border} \\ 0, & \text{if this is free road} \\ -1, & \text{if this is inside the obstacle} \end{cases} , \forall s \in \mathcal{S}$$

action space: possible neighbors to for path construction

$$\mathcal{A} = \{(0,1), (1,0), (0,-1), (-1,0), (1,1), (1,-1), (-1,1), (-1,-1)\}$$

action cost: $\mathcal{C} : \mathcal{A} \to \mathcal{R}$, $\mathcal{C}(a) = ||a||_2, \forall a \in \mathcal{A}$

deterministic transition function $\mathcal{T} : \mathcal{A} \times \mathcal{S} \to \mathcal{S}$, $\forall a \in \mathcal{A}, \forall s \in \mathcal{S}, \mathcal{O}(s) = 0$

$$\mathcal{T}(a, s) = \begin{cases} s' = (s.x + a.x, s.y + a.y), & \text{if } s' \in \mathcal{S} \text{ and } \mathcal{O}(s') = 0 \\ s, & \text{otherwise, the move is illegal} \end{cases}$$

Figure 1: Formulation of the path planning algorithm via the general search problem variables. Search problems in general have the following parameters: state space, obstacle mapping, action space, action cost, and deterministic transition function. (Figure Credit: Chuyue)

2.1.1 A*

With the map provided, informative search can be conducted which is a form of search that is guided by some heuristic function $h(n)$ that evaluates the current states with respect to the goal states. An example of such an algorithm is A*.
Different informative search algorithms differ mainly in their evaluation function $f(n)$ which provides the algorithm an order to expand search nodes. Uniform Cost Search (UCS), for example, only uses the actual cost of the path $g(n)$ from the root to the node n as its evaluation function. Greedy-Best First Search (GBFS) only uses the heuristic function $h(n)$ which measures the distance to goal as its evaluation function. A* uses a combination of both the actual path cost and the heuristical distance as its evaluation function as $f(n) = g(n) + h(n)$.

For the path planning problem, the Euclidean distance from the current state to the goal is used as the heuristic. Notably, this heuristic is admissible, meaning that it never overestimates the actual best cost path to the goal. Furthermore, the total cost at each node also accounts for the accumulated action cost leading up to that node. The complete algorithm is shown in Algorithm 1.

As a search-based algorithm, A* has many strengths. A* is complete, meaning that it successfully returns a solution when one exists and can also correctly report if no solution exists. With a finite search space, A* is guaranteed to avoid searching in repetitive cycles. A* is also optimal, meaning it can reliably return the path with the lowest cost out of all possible paths.

Algorithm 1 A* Search Algorithm

```
1: function ASTAR-SEARCH(problem, F) returnsa solution node or failure
2:     node ← NODE(State = problem.initial, parent = None, costs = 0)
3:     frontier ← a priority queue ordered by F
4:     reached ← a lookup table with key of state and value of costs reaching that state
5:     add node to frontier
6:     reached[node.state] = 0
7:     while frontier is not empty do
8:         node ← POP(frontier)
9:         if problem.is_goal(node) then return node
10:        for action in problem.actions(node) do
11:            costs ← node.costs + problem.action_cost(action)
12:            state ← (node.state.x + action.x, node.state.y + action.y)
13:            if state not in reached or costs < reached[state] then
14:                child_node ← NODE(State = state, parent = node, costs = costs)
15:                add child_node to frontier
16:                reached[state] = costs
17:    return failure
18: function F(node) returns minimal cost that follows node until goal
19:    return node.costs + problem.goal_distance(node)
```

2.2 Sampling-based Motion Planning Algorithms (Shara)

Alternatively, motion planning can be solved with a sampling-based approach. In general, sampling-based algorithms randomly sample feasible values in the search space to create paths between reachable nodes. Alike search-based algorithms, sampling-based algorithms also use trees to create paths; however, nodes are extended via random values (that are checked for collision-free feasibility) rather than a finite set of actions.

2.2.1 RRT*

Rapidly-exploring Random Trees (RRT) is an example of a sampling-based algorithm. In general, RRT works by first initializing the root of the tree at the start point. Then, points are randomly sampled in a given domain (e.g. x and y) via a uniform distribution. This process continues until a point is sampled in the goal region, where the points are finally connected to obtain a path from the initial position to the goal region.

RRT* is another sampling-based algorithm that is very similar to RRT, but includes an additional step as it aims to find a more optimal path than RRT. After a random sampled is added to the path, the algorithm considers the neighboring nodes and their costs to see if a more optimal path can be found, rewiring the tree. This process can make RRT* computationally slower than RRT, but it can be alleviated through using techniques such as parallelization and avoiding repetitive computations.

In our implemented RRT* algorithm, goal biasing is also used during sampling. By sampling points near the goal node, the algorithm can ensure the tree expands in the

correct direction. Furthermore, our implementation evaluates the cost of a subpath by considering randomly sampled nodes that are closer to neighboring nodes. In all, this allows for the algorithm to rewire the tree continuously so existing nodes can be connected to other nodes in paths that are more minimal (optimal) in cost. Algorithm 2 describes our implementation of RRT*.

Algorithm 2 RRT* Algorithm

```
map ← grid containing values of -1, 0, 100 to indicate obstacles (0 indicates no obstacle)
function RRT_STAR() returns a solution path
    start_node ← NODE(State = start_point, parent = None, costs = 0)
    nodes ← [start_node]
    for i = 0, ..., max_iter do
        random_state ← get_random_state(map, i)
        nearest_node, nearest_distance ← get_nearest_node(random_state)
        new_node = steer_towards_random(nearest_node, nearest_distance, random_state)
        if map[new_node.y, new_node.x] != 0 then
            continue
        if new_node.x == nearest_node.x and new_node.y == nearest_node.y then
            continue
        if not check_collision(new_node, nearest_node, map) then
            near_indicies = get_near_nodes(new_node)
            new_node = choose_parent(new_node, near_indices, map)
            nodes.append(new_node)
            rewire(new_node, near_indicies, map)
        if within_goal_region(new_node) then
            path ← backtrack(new_node)
            return path
```

2.3 Pure Pursuit Controller (Jessica)

Once a target trajectory, represented as a connected sequence of points, is obtained from the motion planning module, a control algorithm must be implemented for the car to drive accurately along this path. To accomplish this, a pure pursuit algorithm is used, due to its simplicity and effectiveness. The pure pursuit algorithm follows three critical steps:

1. First, the line segment closest to the car is determined, which corresponds to the line segment the car is currently traversing.

2. Next, a target point is determine using the closest line segment and a lookahead distance.

3. Last, the pure pursuit algorithm is run on the target point to calculate the driving angle.

In this section, the design of the pure pursuit controller is detailed through each of the three steps.

2.3.1 Determining the Closest Line Segment

As the path is represented as a collection of points that are connected as line segments, the first step is to process the path data. By representing the path with only the closest segment (to the car) and its ensuing segments to the goal, segments behind the car can be eliminated as possibilities for pure pursuit target points. Thus, this step is necessary in greatly reducing the computational time needed to run the pure pursuit module.

In determining the minimum distance to a line segment, there are three possible cases to consider. Vectors are used to first model the distances between points A, B, and C (where A is the starting point of a line segment, B is the ending point, and C is the location of the car). Equations 3, 4, and 5 detail how the distances are mathematically calculated by subtracting the corresponding positional components.

$$\begin{bmatrix} AB_x \\ AB_y \end{bmatrix} = \begin{bmatrix} B_x \\ B_y \end{bmatrix} - \begin{bmatrix} A_x \\ A_y \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} BC_x \\ BC_y \end{bmatrix} = \begin{bmatrix} C_x \\ C_y \end{bmatrix} - \begin{bmatrix} B_x \\ B_y \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} AC_x \\ AC_y \end{bmatrix} = \begin{bmatrix} C_x \\ C_y \end{bmatrix} - \begin{bmatrix} A_x \\ A_y \end{bmatrix} \tag{5}$$

The obtained distances are then used to determine which case should be followed:

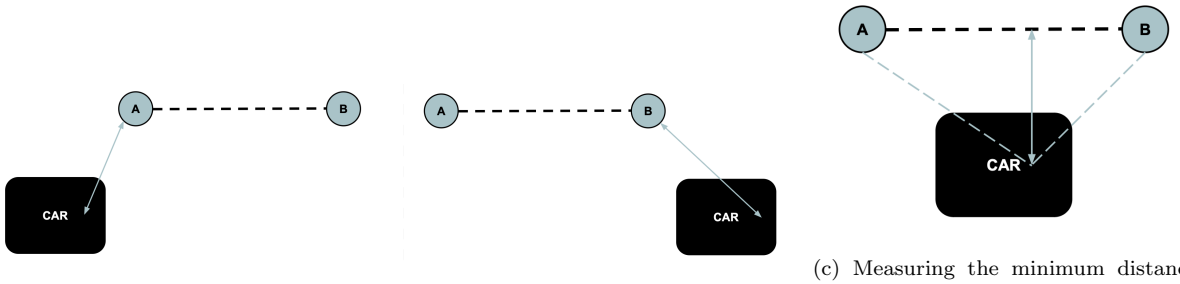
- **Case 1: Closest Point is Point A** If vector AB points in the opposite direction of vector BC, the dot product of the vectors will be negative (Equation 6 is satisfied). This means that the car will still be approaching the line, as demonstrated in Figure 2a. In this case, the point on the line segment closest to the car is Point A. As a result, the minimum distance is the distance from the car to Point A.

$$\begin{bmatrix} ABx \\ ABy \end{bmatrix} \cdot \begin{bmatrix} ACx \\ ACy \end{bmatrix} < 0 \tag{6}$$

- **Case 2: Closest Point is Point B** If vector AB points in the same direction as vector BC, the dot product of the vectors will be positive (Equation 7 is true). The car will have passed the line segment, as shown in Figure 2b. The minimum distance from the car to the line segment is now the distance between the car and Point B.

$$\begin{bmatrix} ABx \\ ABy \end{bmatrix} \cdot \begin{bmatrix} BCx \\ BCy \end{bmatrix} > 0 \tag{7}$$

- **Case 3: Closest Point is Between Points A and B** If neither Cases 1 or 2 are satisfied, then the car is between Point A and B, as shown in Figure 2c. To determine the minimum distance to the segment, a triangle is created with points A, B, and C (the car). The minimum distance is then the height of this triangle.



(a) Measuring the minimum distance from the car to the line segment when the car is behind the line. In this case, the minimum distance from the car to the line segment is the distance from the car to Point A.

(b) Measuring the minimum distance from the car to the line segment when the car in front of the line. In this case, the minimum distance from the car to the line segment is the distance from the car to Point B.

(c) Measuring the minimum distance from the car to the line segment when the car is between the start and end point. In this case, we use a triangle to calculate the minimum distance from the car to the line segment, as this is the height of the triangle.

Figure 2: Different cases to find the shortest distance to the trajectory. (Figure Credit: Jess)

Equations 8 and 9 summarize how the minimum distances are calculated for each of the three cases.

$$\text{Minimum Distance} = \begin{cases} \sqrt{\begin{bmatrix} AC_x \\ AC_y \end{bmatrix} \cdot \begin{bmatrix} AC_x \\ AC_y \end{bmatrix}}, \text{Case 1} \\ \sqrt{\begin{bmatrix} BC_x \\ BC_y \end{bmatrix} \cdot \begin{bmatrix} BC_x \\ BC_y \end{bmatrix}}, \text{Case 2} \\ \frac{1}{b} \cdot \left| det \left(\begin{bmatrix} C_x & C_y & 1 \\ A_x & A_y & 1 \\ B_x & B_y & 1 \end{bmatrix} \right) \right|, \text{Case 3} \end{cases} \tag{8}$$

where

$$b = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2} \tag{9}$$

2.3.2 Determining the Target Point for Pure Pursuit

Using the segment on the path closest to the car, the target point that will be used for pure pursuit is obtained via the intersection points with the car's lookahead distance. A circle, with a radius equivalent to the look ahead distance, is centered around the car's base link as shown in Figure 3. To determine the intersection point(s) between a car and a line segment, the following variables are defined:

- Q = point denoting the car's base link
- r = value of the lookahead distance
- A = point denoting start of the line segment
- $\mathbf{V}$ = the line segment in vector form

Any point M on the vector $\mathbf{V}$ can be represented as the following, where $t \in [0, 1]$. represents the fraction of the line segment:

$$M = A + t\mathbf{V} \tag{10}$$

If the point M on the vector is a distance r away from the car's base link Q, then M intersects the circle spanned by the lookahead distance:

$$|M - Q| = r \tag{11}$$

Combining Equations 10 and 12, the following is obtained to express the lookahead distance:

$$|A + t\mathbf{V} - Q| = r \tag{12}$$
$$(A + t\mathbf{V} - Q) \cdot (A + t\mathbf{V} - Q) = r^2$$

By expanding the dot product and rearranging the equation, the equation can be formatted as a quadratic equation in t. Solving for t using the quadratic formula:

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

where the coefficients are:

$$a = \mathbf{V} \cdot \mathbf{V}$$
$$b = 2(\mathbf{V} \cdot (A - Q))$$
$$c = A \cdot A + Q \cdot Q - 2A \cdot Q - r^2$$

Once t is obtained, it is substituted back into Equation 10 to determine M, the intersection point of the segment on the lookahead circle.

Since t is solved for using a quadratic equation, there are possibly 0, 1, or 2 solutions. If no solutions exist, the line segment is not considered. If there is 1 solution, then the corresponding point is kept as a possibility to serve as the target point for the pure pursuit algorithm. If there are 2 possible solutions (as shown in Figure 3), the point closest to the endpoint of the segment is selected to ensure the car is always driving towards the end of the path. This is done by comparing the calculated distances of the intersections points to the end of the line segment.

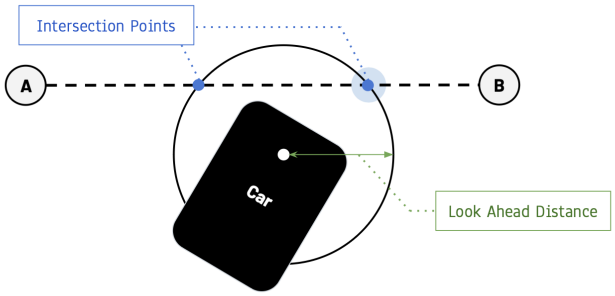


Figure 3: Illustration of the case where there are two intersection points between the segment and the lookahead circle. The car uses a circle with a radius equivalent to the look ahead distances to find intersections with a line segment. If there are two intersections, as shown in the image, the point closer to the end point of the line (Point B), as shown by the blue highlighted circle. (Figure Credit: Jessica)

The process of finding the intersection points is done on the line segment closest to the car and on all of the ensuing line segments. The point that is finally considered as the target point for the pure pursuit is the intersection point found furthest along the path, as demonstrated in Figure 4.

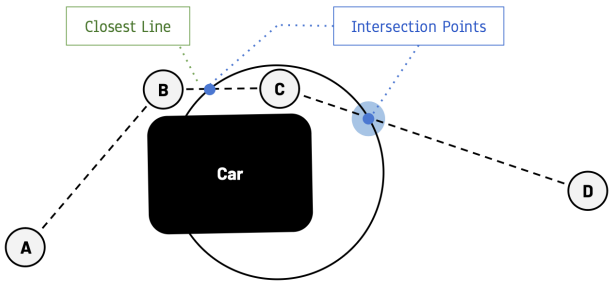


Figure 4: Illustration of how the target point is selected for pure pursuit. After determining the closest line, the line segments composing the remainder of the path are iterated through. Instead of simply using the intersection point on the closest segment, the furthest intersection point along the path is selected. Despite BC being the closest line segment to the car, the intersection point on CD, highlighted blue, is chosen as it is closer to the end of the path and ensures the car is driving in the correct direction. (Figure Credit: Jessica)

2.3.3 Pure Pursuit Algorithm

With a target point obtained, the pure pursuit algorithm calculates δ, the angle at which the car would need to drive to reach the target point head-on. The pure pursuit equation is shown in 13 and its variables are represented visually in Figure 5.

$$\delta = \arctan \frac{2Lsin(\theta)}{Distance} \tag{13}$$

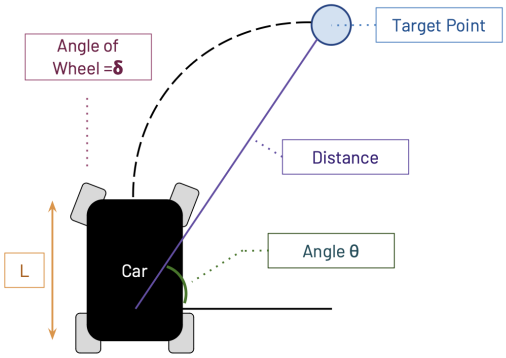


Figure 5: Illustration of the parameters for the pure pursuit equation. The pure pursuit algorithm determines δ, the angle at which the wheels need to turn in order to reach the target point. The equation utilizes the values shown: L, the length of the car; θ, the angle of the car to the target point; *Distance*, the distance from the car to the target point. (Figure Credit: Jessica)

2.4 Integration (Nihal & Shara)

At a high level, the localization, motion planning, and pure pursuit modules are integrated together to construct our path planner. The `/initialpose` topic communicates an initial pose to the particle filter, which is used for the car to localize itself on the map. In addition, this initial pose also provides the starting point for the trajectory used in the motion planning module. Next, `/move-base-simple/goal` publishes a goal point that the A* motion planner uses to find an optimal trajectory. With both the optimal trajectory and the car's localized pose estimate published, the pure pursuit controller runs and publishes the relevant driving commands (e.g. velocity and steering angle) to the race car. Figure 6 illustrates how the aforementioned modules communicate with each other.

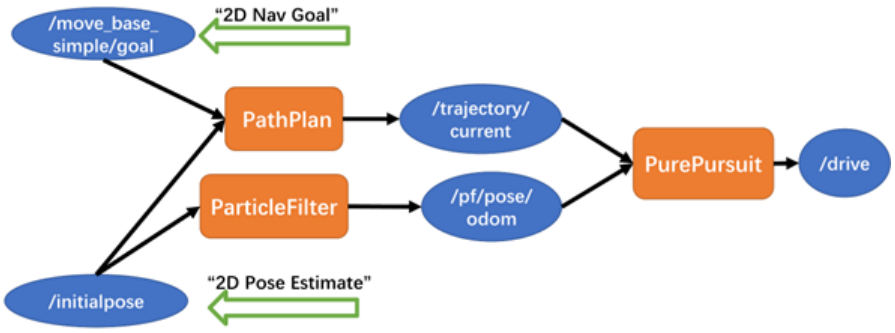


Figure 6: This figure illustrates how different topics on the car link to each other. `/initialpose` communicates to both PathPlan and ParticleFilter as it provides the start pose and the initial pose for each of those modules respectively. `/move-base-simple/goal` provides the end pose to PathPlan which lets it find a feasible and optimal trajectory. `/trajectory/current` publishes the trajectory and `/pf/pose/odom` publishes the pose of the car which lets us use the PurePursuit module to send a steering angle command to the `/drive` topic. (Figure Credit: Chuyue)

2.4.1 Dilation (Chuyue)

To further prioritize the car's safety in the path planner, map dilation is introduced to avoid produced trajectories from being too close to obstacles or walls. In an image F, dilation sets a pixel at (i, j) to the maximum value of all pixels in the surrounding neighborhood S centered at (i, j).

$$(F \otimes S)[i, j] = \max_{(x,y)\in S} F[i + x, j + y] \tag{14}$$

In the supplied occupancy grid map used for this lab, pixels denoting obstacles or walls were given a value of 100 and pixels denoting free, unoccupied spaces were given a value of 0. As a result of obstacle pixels being represented with a higher value, image dilation will increase the size of the areas pertaining to obstacles (borders) and thus reduce the number of pixels denoting free, unoccupied spaces. For example, a simple border line will be expanded after the following dilation:

$$\begin{bmatrix} 0 & 100 & 0 \\ 0 & 100 & 0 \\ 0 & 100 & 0 \end{bmatrix} \otimes S_{2\times 2} \rightarrow \begin{bmatrix} 100 & 100 & 100 \\ 100 & 100 & 100 \\ 100 & 100 & 100 \end{bmatrix} \tag{15}$$

Image dilation forces our path planning algorithms to consider a tighter search space when returning an optimal path (as shown in Figure 7). Thus, the optimal paths returned

are safer, as they are further from actual obstacles and walls. This greatly reduces the chances of the car coming too close to obstacles and potentially hitting them, allowing more freedom for our integrated path planner to slightly deviate in executing such paths while still prioritizing the car's safety.

(a) Original map without dilation. Feasible, unoccupied spaces are abundant in area.

(b) Map borders adjusted by a dilation factor of 5. As the dilation increases, the region of feasible spaces becomes more restricted.

(c) Map borders by dilating the map with a factor of 10. If the dilation factor is too large, it may cause paths to break in the dilated map, causing paths that originally traverse those areas to no longer be feasible.

Figure 7: Illustrating the effects of map dilation. Black areas represent unoccupied spaces that the car can travel to and thus fesible regions that a path can be planned through. Map dilation is used to ensure paths are safer and further away from walls and other obstacles. (Figure Credit: Chuyue)

3 Experimental Evaluation (edited by Aaron)

To evaluate the success of our entire path planning system, we first examine each individual module (A* and pure pursuit controller) individually in simulation. Specifically, we ensure that our A* algorithm can indeed return the optimal path (especially in comparison to RRT) and that our pure pursuit controller can follow such a path with minimal error. Once the success of each individual module is verified, we then test the integrate in simulation. Demonstrated success in all simulated tests finally justified implementing the integration onto the physical race car. Qualitative and quantitative measures were used to investigate and tune the parameters to achieve the desired actions of the system. In this section, we cover the various analyses conducted at each stage that enabled the resulting success of our physical system.

3.1 Motion Planning (Chuyue & Shara)

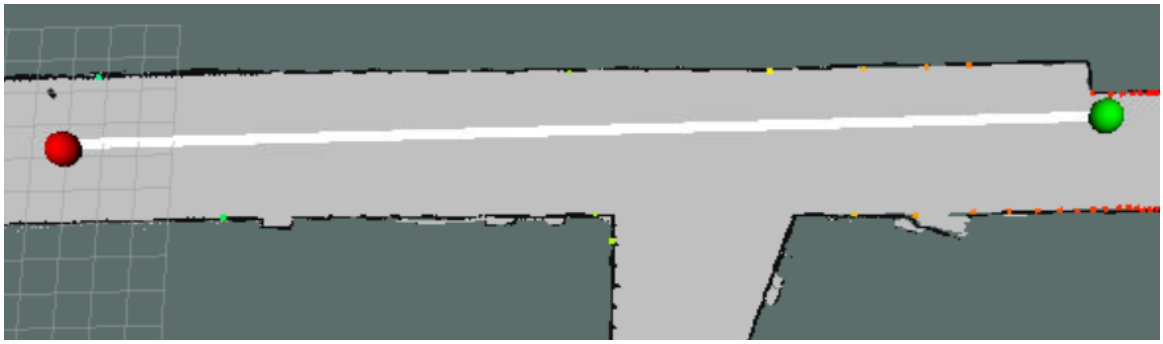
We begin by describing the experimental measures used to evaluate both our implemented search-based algorithm (A*) and our sample-based algorithm (RRT). The empirical results are interpreted in an ensuing discussion that furthers the comparison between search-based and sample-based algorithms.

3.1.1 Search-based Algorithms

To evaluate the optimality of our A* algorithm, metrics such as the number of generated nodes, the length of the calculated path, and the computational time were obtained [1]. Such measures were also collected for other search-based algorithms including Uniform Cost Search (UCS) and Greedy-Best First Search (GBFS) to demonstrate the optimality of A* among search-based algorithms. Metrics were collected by testing in three different scenarios:

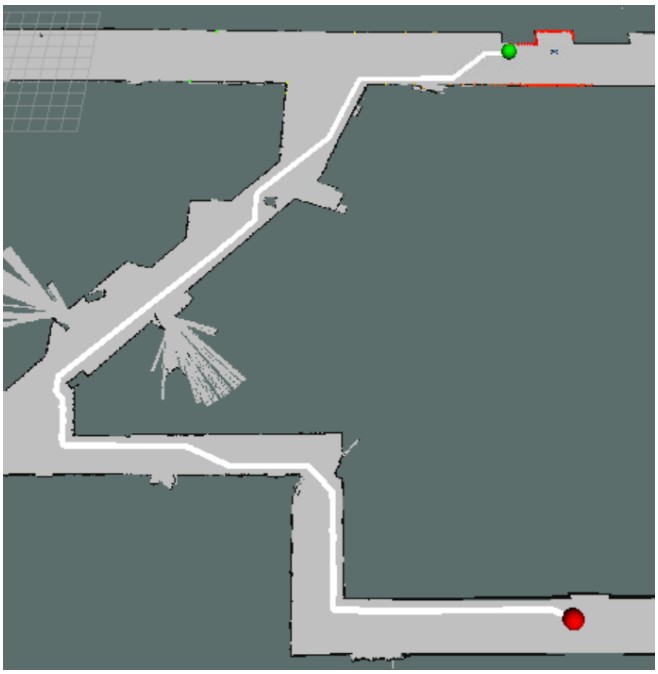
1. A straight hallway (Figure 8a)
2. A short path around various corners (Figure 8b)
3. A long path throughout the Stata basement (Figure 8c)

Figure 8 illustrates the paths returned by the A* algorithm.



(a) Test 1: Straight hallway. A* is able to find a direct straight-line route.

(b) Test 2: Short path around corners. A* is able to find an optimal path as it cuts through corners optimally.



(c) Test 3: Long path throughout the Stata basement. A* is able to proceed to the destination point in the correct, optimal direction.

Figure 8: Illustration of the paths returned by A* for each of the three testing scenarios. Testing with various straights and turns allowed for evaluation in a variety of real-world driving scenarios. (Figure Credit: Chuyue)

In the first test, all algorithms are able to obtain the optimal, straight-line path; however, the algorithms differ largely in the number of nodes generated. GBFS greedily goes toward the final goal, generating the least number of nodes. UCS on the other hand explores every direction, causing a lot of nodes to be generated. As A* is guided by both the path cost and the heuristical distance to the goal, it only generates a moderate number of nodes.

In the second test, the path includes several turns, dead ends, and narrow pathways. Although GBFS still generates the least number of nodes, it greedily goes to a dead end along the road which appears to lead it closer to the goal, resulting in a non-optimal path. A* and UCS both can find the optimal path, but A* uses less nodes to do so.

In the last test, GBFS searches through nearly 5 times the number of nodes of that of A* and UCS, yet still can't find an optimal path; A* and UCS however are both able to do so. Moreover, UCS does so by generating fewer nodes than A*.

Type of search		**Test 1: Straight hallway**	**Test 2: Short obstacles**	**Test 3: Across basement**
	Start Point (in px)	(1140, 991)	(785,710)	(1140,991)
	Goal Point (in px)	(550, 988)	(923,321)	(1150,294)
A*	Nodes Generated	4470	**42366**	270632
	Path Length (in m)	29.799	**34.982**	73.018
UCS	Nodes Generated	111145	98777	**228930**
	Path Length (in m)	29.799	34.982	**73.018**
GBFS	Nodes Generated	**1781**	18810	1154218
	Path Length (in m)	**29.799**	37.182	80.528

Table 1: Data comparing the number of generated nodes and returned path length for three search-based algorithms (A*, uniform cost search (UCS), and greedy best first search (GBFS)) collected in three different testing scenarios as illustrated in Figure 8. The data suggests GBFS might be more computationally efficient (smaller number of nodes generated) but in turn, optimality is sacrificed (higher path length). On the other hand, UCS is more optimal but may be computationally inefficient as has no understanding of the goal location and thus conducts inefficient search. A* is generally better in balancing the trade-off between optimality and computational efficiency (as demonstrated in various scenarios), and thus A* is used as the motion planning algorithm. (Table Credit: Chuyue)

From these analyses, the resulting three conclusions are made:

1. GBFS may be more computationally efficient, but does so at the cost of optimality.
2. UCS finds the optimal path, but is computationally inefficient as it often conducts search in unnecessary directions.
3. A* not only finds the optimal path but it is also generally efficient across different driving scenarios. As a result, A* was the search-based algorithm chosen to be implemented.

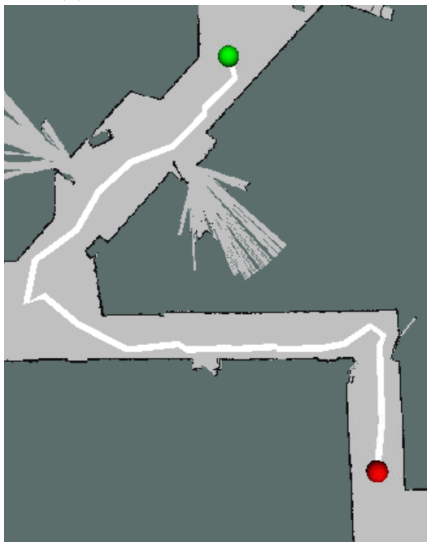
3.1.2 Sampling-based Algorithms

To evaluate the optimality of our RRT* algorithm, the same metrics and testing scenarios were used as to evaluate our A* algorithm in Section 3.1.1.

Due to the random nature of RRT*, the algorithm does not provide the same path with each execution. As a result, each testing scenario was run four times, and the measures obtained were averaged to get a more general representation of RRT*'s ability. Averaged empirical data collected from these tests are shown in Table 2. Tables 3, 4, and 5 list

more specifically the details of each trial for each testing scenario. Figure 9 shows the paths produced by RRT* for all three testing scenarios.

(a) Test 1: Straight hallway. RRT*'s random sampling causes the path to be curved, and thus unoptimal.



(b) Test 2: Short path around corners. RRT* eventually gets to the goal, but does so inefficiently with large corner turns.

(c) Test 3: Long path throughout the Stata basement. RRT* chooses to proceed in the opposite direction (compared to the path found by the A* algorithm).

Figure 9: Illustration of the three path scenarios used to test RRT* algorithm. Testing with various straights and turns allowed for evaluation in a variety of real-world driving scenarios. (Figure Credit: Chuyue)

From the data, it can be observed that for tests with higher numbers of nodes sampled, larger run times were needed. It can also be seen that the average number of nodes sampled and the average path length is higher for the second testing scenario (short path around various corners) compared to the third testing scenario (long path throughout Stata basement). This perhaps can be attributed to the fact that in Testing Scenario 2, the area needed to be explored is small and there are many corners to account for. Since many turns could possible be taken in a smaller area, more nodes need to be sampled and explored for the path in case of unexpected obstacles. In Test Scenario 3, a larger area is needed to be explored (which encapsulates the area covered by Testing Scenario 2) – but since it is a larger area it is easier to sample in many places and find a path to the goal faster.

Notably, the computational times for RRT* are high. We attribute to this to focusing more on our A* algorithm implementation for this lab. We detail later in Section 4 our future approaches to further optimize our RRT* implementation.

RRT* Tests	Start Point	Goal Point	Avg Time (s)	Avg Nodes Sampled	Avg No. Nodes in Path	Avg Path Length
Test 1: Straight Hallway	(1140, 991)	(550, 988)	32.26	182	82	133.79
Test 2: Short Obstacles	(785,710)	(923,321)	228.63	1128	611	1125.882
Test 3: Across Basement	(1140, 991)	(1150,294)	141.197	960	455	187.033

Table 2: Averaged measurements over four iterations for each of the three testing scenarios using RRT*. Due to RRT*'s reliance on random sampling, multiple trials needed to be used and averaged. (Table Credit: Shara)

RRT* Test 1 Iterations	Nodes Sampled	Nodes in Path	Path Length	Time
1	187	82	132.2402	33.4244
2	107	65	135.82201	18.6825
3	112	52	132.8694	18.7853
4	322	128	134.2306	58.1877

Table 3: Data obtained for the four trials conducted in the straight hallway test using RRT*. Data values vary with each trial due to the randomized nature of RRT*. The higher the number of nodes sampled, the longer it was needed to complete the algorithm execution. Compared to other testing scenarios, the straight hallway test took the least time. This is expected because this testing scenario looks at a shorter path with less obstacles than the other two scenarios.(Table Credits: Shara)

RRT* Test 2 Iterations	Nodes Sampled	Nodes in Path	Path Length	Time
1	1237	665	1125.0166	253.59
2	1067	592	1129.927	221.889
3	997	528	1123.9747	229.6796
4	1212	657	1124.611	209.35

Table 4: Data obtained for the four trials conducted in the shorter path with turns testing scenario using RRT*. The lengths of the paths in this scenario are larger than that of other scenarios, such as Testing Scenario 3. Since many turns are taken in a shorter area region, more nodes are sampled and explored for the path. (Table Credits: Shara)

RRT* Test 3 Iterations	Nodes Sampled	Nodes in Path	Path Length	Time
1	600	283	191.1969	77.779
2	557	311	183.434	80.112
3	852	396	187.934	123.77
4	1832	830	185.568	283.127

Table 5: Data obtained for the four trials conducted in the long path across Stata basement using RRT*. As this scenario covers a larger area with less turns and obstacles, more points can be sampled that more easily cover the area, allowing for less nodes to be sampled and shorter paths to be returned. (Table Credits: Shara)

3.1.3 Empirical Comparison of A* and RRT* (Nihal & Chuyue)

An empirical discussion comparing our implementations of A* (search-based algorithm) and RRT* (sampling-based algorithm) inform us of our justified decision to use A* as the motion planning algorithm for our path planner. Table 6 displays the run times for A* and RRT* as an average of four runs on the same three testing scenarios as previously mentioned.

	Test 1	Test 2	Test 3
A*	0.174 s	1.195 s	17.833 s
RRT*	18.6825 s	209.35 s	77.779 s

Table 6: Comparing the run-times for each of the three testing scenarios using A* and RRT*. The times for RRT* tests are the fastest times from the set of runs of each test. (Table Credits: Shara)

A general comparison between the two methods are listed in Table 7. In the end, we choose A* as our motion planning algorithm mainly because of its guaranteed optimality.

RRT*	A*
1) Probabilistically complete.	1) Deterministically complete.
2) Faster in finding a solution.	2) Slower in finding a solution.
3) Performs worse in narrow hallways.	3) Affected less by obstacles.
4) Will not terminate if a solution doesn't exist.	4) Will terminate if a solution exits.
5) Not always the optimal path.	5) It is optimal.
6) Can take dynamics into account and plan for a dynamic environment.	6) Cannot take dyanmics into account and plan for a dynamic environment.

Table 7: Comparing RRT* and A*. Although RRT* has certain benefits, we justify our decision to choose A* for the purposes of this lab because A* reliably guarantees an optimal path in terms of distance. (Figure Credit: Nihal and Chuyue)

3.2 Pure Pursuit Controller (Nihal)

In order for the pure pursuit controller to work well, the lookahead distance needs to be tuned. Using a constructed step trajectory, the step response of the car is tested in simulation at different lookahead distances for a particular velocity. Figure 10 illustrates this trajectory.

Figure 10: Constructed step trajectory used to test the pure pursuit controller. The white path shows the step trajectory which is the test path. The green point represents the start position and the red point is the goal position. (Figure Credit: Nihal).

The testing procedure involves setting the lookahead distance to 0.4m, 0.6m, and 0.8m, while holding velocity constant at 1.5 m/s. The error, defined as the shortest distance to the line, is recorded during the path execution. Figures 11, 12, and 13 graph the recorded error for the tests over the different lookahead distances. In general, when the lookahead distance is too low, then the car has oscillatory behaviour. As the lookahead distance increases, the car starts to turn earlier and is able to handle the trajectory better. When the lookahead distance is increased further, however, it results in sluggish behaviour and the car does not follow the trajectory well.

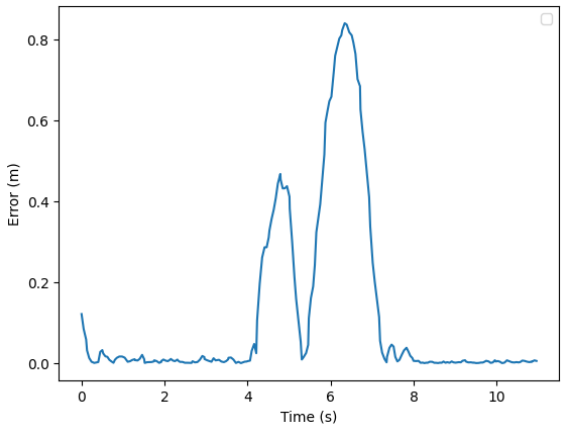


Figure 11: Error in following the step trajectory for a lookahead distance of 40 cm. The large peaks correspond to the car overshooting when turning along the step. The maximum error is close to 80 cm. (Figure Credit: Shara)

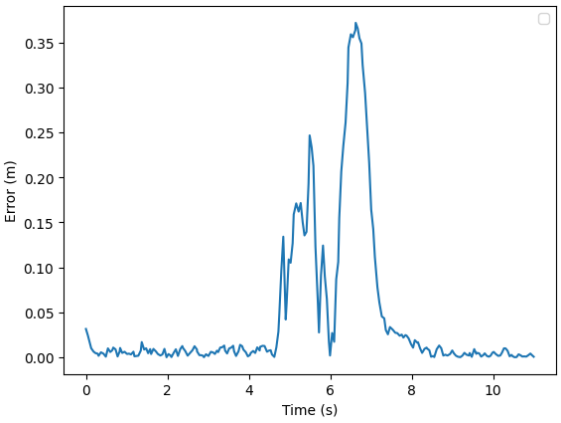


Figure 12: Error in following the step trajectory for a lookahead distance of 60 cm. The large peaks correspond to the car overshooting when turning along the step. The maximum error is close to 35 cm. (Figure Credit: Shara)

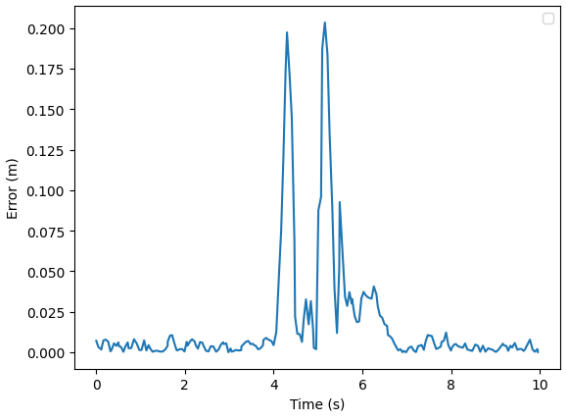


Figure 13: Error in following the step trajectory for a lookahead distance of 80 cm. The large peaks correspond to the car overshooting when turning along the step. The maximum error is close to 20 cm. (Figure Credit: Shara).

3.3 Integration

In testing the integration of our path planner, we first used simulation to assess the logic of our A* and pure pursuit algorithms and obtain a primitive understanding of the parameters. Once we verified our integration was working successfully in simulation, the path planner was then implemented on the physical race car for more intensive evaluation and tuning.

3.3.1 Simulation (Nihal)

Simulation was first used to test whether all the different components would work as expected when integrated together. To do this, the start and end points in Figure 14 were used to find and follow an optimal path. Figure 15 illustrates the error in following this trajectory with the same lookahead distance and velocity values used in Section 3.2. The error stays close to zero for most of the trajectory except during some turns where the maximum error does not deviate more than 20 cm from the setpoint.

Figure 14: Planned path using A* that the car will follow to complete the racetrack course setup. The green point represents the start position and the red point represents the finish position. (Figure Credit: Nihal).

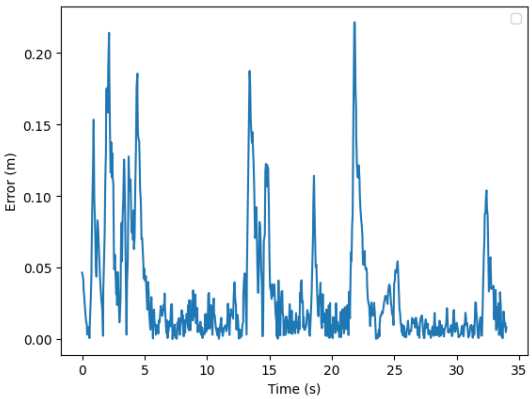


Figure 15: Error in executing the planned trajectory shown in Figure 14. The peaks correspond to pure pursuit cutting corners due to the trajectory being made with piecewise linear functions. The maximum error remains below 20 cm. (Figure Credit: Shara).

The path planner's demonstrated success in simulation suggested it was ready to be implemented onto the physical race car.

3.3.2 Physical Implementation (Aaron)

Two main objectives were considered while testing the implementation of the path planner on the physical race car: 1. to allow the car to execute the optimal path as fast as possible, and most importantly 2. to ensure the safety of the race car during the path execution. With these objectives in mind, we investigated how the car's performance changes as a result of various tunable parameters, including the dilation factor, velocity, and look-ahead distance. Table 8 summarizes the general testing procedure carried out for the physical race car.

First, the dilation factor was examined. As the dilation factor increased, the optimal trajectory returned forom the motion planner was farther from walls and other obstacles. When testing on the actual race track, it was discovered that increasing the dilation factor beyond 8 (corresponding to 1.33x the actual size of the race car) resulted in the motion planning algorithm being unable to return the direct-most route to the finish line (as shown in Figure 16). Recognizing that the safety of the car should be the utmost priority, we settled on a dilation factor of 8. This allows for the optimal trajectory returned to be as far away from obstacles as possible while ensuring the direct route from the start to finish line in the race track remains feasible.

Priority	**Test Type**	**Attribute to be Evaluated**	**Test Description**	**Measurement of Success**
2	Executional Accuracy	Dilation	Attempt to plan for the optimal path at varying dilation factors, and qualitatively assess if the resulting path directly connects the start line to the finish line.	The dilation constant should be as large as possible (to prioritize the car's safety by returning paths farther away from obstacles) but still allow the direct race track path to be feasible.)
2		Velocity	Holding the dilation factor constant (i.e. 8) and the lookahead distance constant (i.e. 1.0m), examine the resulting error when running the car at velocities of 1.0 m/s, 1.5 m/s, and 2.0 m/s on a test path involving a straightaway and a turn.	The velocity should be maximized to allow the race car to execute the path as quickly as possible while still minimizing the error from the optimal path.
2		Lookahead Distance	Holding the dilation factor constant (i.e. 8) and the velocity constant (i.e. 1.5 m/s), examine the resulting error when changing the lookahead distance to 0.8m, 1.0m, and 1.2m on a test path involving a straightaway and a turn.	Tune the lookahead distance such that the error between the car's localized position and the optimal path is minimized.
3	Executional Time	Time	The time is measured for the car to complete the race track from the start to finish line.	The time should be as fast/low as possible.
1	Safety	Safety	Qualitatively examine if the car comes into danger of hitting any walls or obstacles during the executed path.	The car never comes in close danger of hitting any wall or obstacle during the executed path.

Table 8: Testing plan for the integration of the path planner on the physical race car. In general, testing was primarily conducted to investigate the general effects of each parameter (i.e. dilation, velocity, lookahead distance) before parameters were more officially tuned when testing for time on the actual race track. Safety was the upmost priority throughout all of our testing cases. (Figure Credit: Aaron)

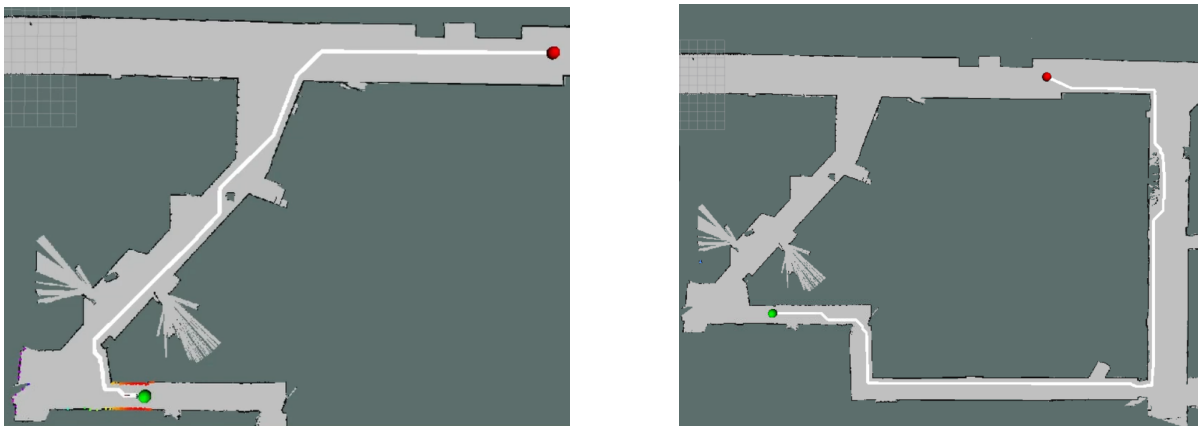


Figure 16: Returned optimal path by A* with a dilation factor of 8 (left) and a dilation factor of 9 (right). Increasing the dilation factor allows for safer paths further away from wall and obstacles to be returned; however, increasing the dilation factor by too much may cause for optimal paths to become unfeasible, as seen in the figure on the right where the path travels all away around the map. We settled on a dilation factor of 8 to prioritize the car' safety while also ensuring the optimal race track path could be found. (Figure Credit: Aaron)

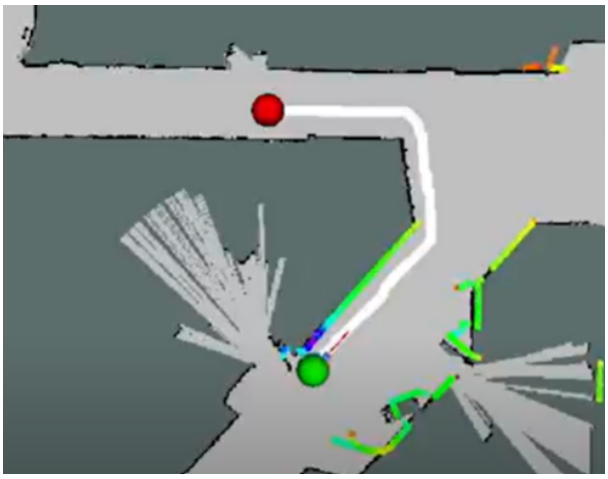
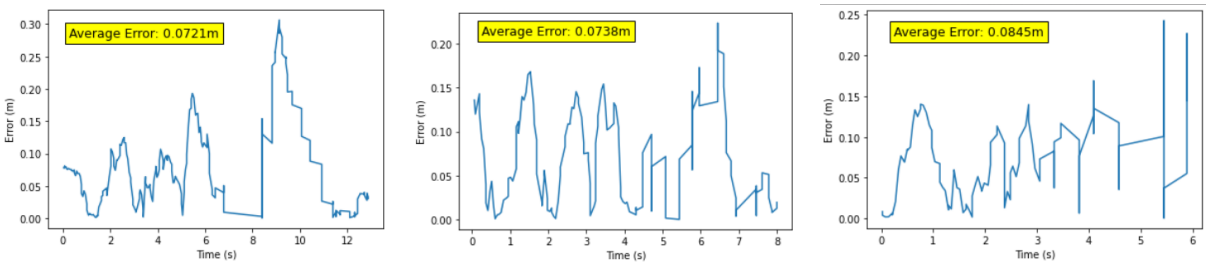


Figure 17: Illustration of the path used to test the physical implementation of the path planner, starting from the green point to the red destination point. The path contains a simple straightaway and a turn, two scenarios most common in the real-world. Careful consideration was taken to also choose an environment where there were no random obstacles that were unaccounted for in the given map (e.g. boxes and clutter) so it wouldn't confused the car's localization module and thus provide more reliable measures of error. (Figure Credit: Aaron).

To further investigate the velocity and the lookahead distance, a simple test path was constructed involving a straightaway and a turn. Figure 17 illustrates the test path used.

While holding constant the dilation factor of 8 and a lookahead distance of 1.0m, the velocity was altered to investigate how it would affect the performance of the race car – which is essential as the objective is to run the path planner at higher velocities. The velocity was tested at three values including 1.0 m/s, 1.5 m/s, and 2.0 m/s. As the velocity increased, the average error increased and oscillations in the race car's movements became more prominent, as illustrated in the graphs in Figure 18. This suggested the need to tune the parameters of the path follower as a function of velocity, especially as the path planner is prepared for final testing on the race track.



(a) Error for velocity of 1.0 m/s. The average error during the path execution is approximately 0.0721m. Oscillations in the error are relatively small in magnitude as seen in the graph.

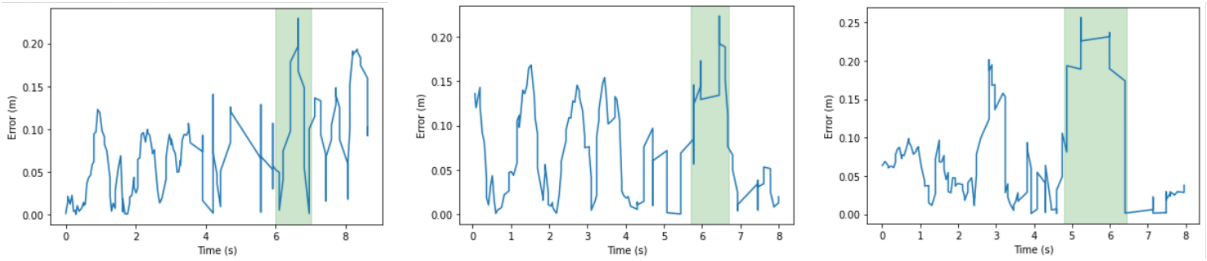
(b) Error for velocity of 1.0 m/s. The average error during the path execution is approximately 0.0738m. Oscillations in error become more frequent as can be seen in the graph.

(c) Error for velocity of 1.0 m/s. The average error during the path execution is approximately 0.0845m. Oscillations in error are larger in magnitude as seen in the graph.

Figure 18: Error plots holding constant a dilation factor of 8 and a lookahead distance of 1.0m, while varying the velocity to (from left to right) 1.0 m/s, 1.5 m/s, and 2.0 m/s. The average error over the duration of the path execution is also shown in yellow. As the velocity increases, the average error increases. This motivates for parameters to be tuned potentially as a function of velocity as the objective is to run the car at increasing speeds. (Figure Credit: Aaron)

The lookahead distance was then examined on the same test path, now holding constant the dilation factor at 8 and the velocity at 1.5 m/s. The error between the car's executed path and the optimal planned path was obtained at lookahead distances of 0.8m, 1.0m, and 1.2m. When the lookahead distance was smaller, the race car visibly became more oscillatory during straightaways. At a lookahead distance of 0.8m, the average error during straightaway was approximately 0.0721m; notably, however, the average error during the turn was 0.112m. When the lookahead distance was larger at 1.2m, the oscillatory behavior during straightaways became minimized. The average error during straightaways reduced to 0.0529m, but the average error during the turn increased to 0.177m. Seemingly, a larger lookahead distance allowed the car to perform better in terms of error to the optimal path during straightaways, but a smaller lookahead distance allowed the car to perform better during turns. This suggested the need to consider the tradeoff inherent with the lookahead distance when assessing the intended performance of the race car. Figure 19 depicts the associated error plots while varying the lookahead distance.

Ultimately, in deploying the system on the race track, a lookahead distance of 1.5m was decided upon. This lookahead distance was chosen to be as large as possible to minimize any oscillatory behavior on straight sections, yet was still small enough to enable the race car to safely execute any turns. Tuned with a dilation factor of 8 and at a velocity of 1.75 m/s, the race car is able to complete the race track in 30.52 seconds. A timed video of our system executing the race track can be found here.



(a) Error for lookahead distance of 0.8m. During straights, the race car averages an error of 0.0721m. During turns, the race car has an average error of 0.112m. Visually, the error on the graph is more oscillatory, and the error peak during the turn is somewhat narrow.

(b) Error for lookahead distance of 1.0m. During straights, the race car averages an error of 0.0575m. During turns, the race car has an average error of 0.148m. Visually, the error on the graph is still somewhat oscillatory, but notably the error peark during the turn is wider.

(c) Error for lookahead distance of 1.2m. During straights, the race car averages an error of 0.0529m. During turns, the race car has an average error of 0.177m. Visually, oscillations in the error graph are minimized, but during the turn the error peak is larger and wider.

Figure 19: Error plots holding constant a velocity of 1.5 m/s and a dilation factor of 8, while varying the lookahead distance by (from left to right) 0.8m, 1.0m, and 1.2m. The green shaded regions on the plots indicate when the race car was performing a turn on the test path (as shown in Figure 17. These tests indicated an inherent trade-off in tuning the lookahead distance. Higher lookahead distances allow for the race car to perform better on straight paths. but lower lookahead distances allow for the race to perform better on turns. (Figure Credit: Aaron)

4 Future Work (edited by Aaron)

Although our path planner works well in a variety of tested scenarios, our experimental results suggest possible areas of improvement that can be considered for future work. In this section, we briefly discuss two possible areas where future work can be conducted to improve our path planner if time constraints were disregarded.

4.1 Motion Planning (Nihal)

In terms of motion planning, we discuss two possible areas for improvement related to the efficiency of our RRT* algorithm and our path representation.

- Currently, our RRT* algorithm requires large computational times. To optimize RRT* to run faster (which will be important especially for the final challenge where systems must plan dynamically), strategies such as vectorizing, parallelizing code, and using better data structures can be used.
- We can also consider implementing the usage of Debian curves instead of piecewise linear functions to allow pure pursuit to follow paths better.

4.2 Pure Pursuit (Jessica)

The resulting conclusions from our experimental evaluations suggested valuable ideas in tuning the parameters of our pure pursuit controller that we were unable to implement due to time constraints.

- One of the experimental conclusions from testing the physical integration of the path planner was that the parameters should perhaps be tuned as a function of velocity. When the velocity of the car is too high, it is unable to make sharp turns.

The lookahead distance should, therefore, be decreased around sharp turns and increased for straight paths.

- Experimental results also revealed that when the look-ahead distance is adjusted to effectively make a sharp turn, it oscillates along straight lines. However, when it is tuned for straight sections, it is unable to make turns. Therefore, the look-ahead distance should be a function of the velocity and/or radius of a turn (i.e. having two separate lookahead distances for straightaway and turns).

5 Conclusion (Jessica / edited by Aaron))

In this lab, we successfully designed, implemented, and tested a module that allows our robot to plan an optimal trajectory and follow it. Search and sample-based planning methods were first implemented through A* and RRT* which were then compared experimentally to determine which fits the purposes of the lab best – a motion planning algorithm that is efficient, feasible, and optimal. RRT* proved to be slower due to its randomness and inefficiency in navigating cluttered and narrow environments; on the other hand, A*, despite occasionally being slightly slower, guarantees an optimal path. Next, a pure pursuit algorithm was implemented to effectively follow a target point mathematically calculated on a given path. By integrating our planning methods and pure pursuit algorithm, our path planner is able to safely and efficiently traverse a path from a start to end point.

Quantitative and qualitative analyses were used to debug our code, tune our values, and ultimately determine the success of our modules. From our testing procedures, key trade-offs in the different planning methods such as efficiency and path optimization were recognized. Additionally, testing allowed for the lookahead distance to be tuned so that the pure pursuit algorithm could allow the car to travel effectively around tight corners and along straight paths. Ultimately, we were able to justify our choices by demonstrating our car quickly planning for and executing a path in the race track challenge.

Overall, the fundamentals of this lab will be vital in our implementation of the final challenge. An effective pure pursuit controller will be necessary for quickly traversing the racetrack. Having efficient and optimal path planning module will be vital when finding a path trough the city. By continuing to develop and integrate our algorithms, we will be prepared to succeed in completing the final challenge.

6 Lessons Learned

Aaron: On the technical side, I learned different algorithms' strengths and weaknesses can inform when they are best used – that one algorithm isn't necessarily better than the other. More specifically, when comparing the RRT algorithm with the A* algorithm, we experimentally proved how RRT is computationally faster than A* but isn't able to guarantee an optimal trajectory. For the purposes of this lab, we determined A* was more appropriate since we cared more about quickly executing an optimal trajectory rather than the time needed to pre-compute the path. However, for the final challenge when our system will need to react quickly to the dynamic world, perhaps RRT's faster

computational time will be more appropriate. On the communication side, I learned how synchronous, in-person time has become extremely underappreciated. Especially after COVID, the world has transitioned to rely more on asynchronous, remote work. While this system allows more flexibility (which is especially important for us as busy college students), it can be quite difficult to ensure everyone in the group is on the same page. In-person time is extremely valuable as it provides a platform to hold everyone accountable and allow for high-productivity – for instance, clarifying questions can be instantaneously answer in-person instead of needing to expend additional effort to clearly type and wait for a response asynchronously.

Jess: First of all, I learned the importance of creating a priority hierarchy when limited in time. This lab was a bit accelerated due to other deadlines being pushed back, so determining what was most important to use and completely those tasks first was vital. At the end, we were able to reflect on which objectives weren't completed and the different ways we can improve our code moving forward. On a more technical side, I learned the importance of creating modular code for testing. We had a small error where we had accidentally replaced addition with multiplication in the code. Being able to test east section one at a time allowed us to isolate where the error was located and debug efficiently.

Chuyue: Technically, I learned how to evaluate different algorithms. For example, we can use the number of generated nodes and path length to evaluate different motion planning algorithms. Also, the time test might be dependent on computers. At first, I tested my A* algorithm on my computer and it took a long time, but it turned out to be x10 faster on my teammate's computer. Moreover, I expected that sampling-based methods like RRT* would be faster than search-based methods, but it turned out my naive implementation of RRT* is less efficient. And I am still trying to figure out better data structures for the explored tree, which can support finding near nodes, updating costs, and other operations more quickly. Furthermore, I also appreciated the importance of teamwork, and how my teammates devoted their time to designing good test cases for real cars.

Nihal: From a technical standpoint, I learnt the importance of having to produce metrics to decide which algorithms to use when there are many options. This taught me about decision-making and about comparing and contrasting different approaches quickly due to the time crunch that this lab was. I also learned about the importance of integrating and testing as early as possible as this allows for time to find issues and fix them quicker. From the communications standpoint, I learnt about how important it is to realise the strengths of each teammate and communicate it to them in order to produce the best possible solution. Furthermore, I also learnt about the importance of having documented what was being done in order to have important ideas and issues passed on to teammates when we were working asynchronously.

Shara: In this lab, I learned more about how to evaluate the tradeoffs and benefits of different algorithms. We had to determine which motion planning algorithm would work best for this lab. Although we decided to use A* for this lab because it always produces an optimal path, we may opt to use RRT* for the final challenge because it would work better in a dynmaic environment. I also learned more about optimizing algorithms and

hope to continue to learn more. Right now our RRT* algorithm is not very fast but we will need to change this for the final challenge, we can try many approaches to optimize by starting off with data structure representations and reducing the need for repetitive computation. In this lab, I also learned more about organizing tasks and budgeting time for them and as a result figuring out the importance and priority of different tasks. I also learned more about team communication and being able to work on things simultaneously as well as delegating. The week that this lab occurred was busy for many team members so we had to learn how to communicate details and hand off code better.

References

[1] Stuart J. (Stuart Jonathan) Russell. *Artificial intelligence : a modern approach*. Pearson series in artificial intelligence. Pearson, Hoboken, NJ, fourth edition. edition, 2021. ISBN 9780134610993.