Lab 6 Report: Path Planning

Team 16: Sameen Ahmad, Phillip Johnson, Trevor Johst, Maxwell Zetina-Jimenez, Ellen Zhang

1 Introduction

Author: Sameen Ahmad

PATH planning is a core aspect of autonomous operations that allows vehicles to generate and follow a trajectory from a start pose, or position and orientation, to a goal pose. Path planning is an active area of research that involves exploring search-based and sampling-based motion planning algorithms to determine the shortest path, while avoiding collisions in a given environment. Integrating controls onto our race car is equally important to the path planning aspect of this lab, enabling our car to accurately follow a predefined trajectory.

The goal of this lab was to implement an algorithm to plan a trajectory from the car's current pose to a chosen goal pose in a known occupancy grip map. Additionally, we were instructed to integrate pure pursuit control onto our race car, allowing the car to follow the trajectory while avoiding collisions.

We developed and tested breadth-first search (BFS), randomly-exploring random trees (RRT*) and A* algorithms in simulation. After significant testing, we decided to integrate A* on our race car.

Despite using A* for this lab, we realized its limited by its inability to account for the car's dynamics. Therefore, we plan to use RRT* with Dubins curve dynamics for the final challenge. After generating a trajectory, we employ pure pursuit to computes the angular velocity that the race car should drive at so it aligns with a look ahead point along the predefined trajectory.

This lab relies heavily on the Monte Carlo localization algorithm that we implemented previously. We utilize on our particle filter to determine the pose of the car in a known environment, allowing our generated trajectory to begin at the correct position. The next steps involve directly applying our path planning and controls algorithms towards the final challenge, which involves racing against other cars and safely navigating dynamic environments.

Editor: Ellen Zhang

2 Technical Approach

Author: Sameen Ahmad

In this lab, we were tasked with implementing an algorithm that plans a path and follows it with pure pursuit controls while avoiding collisions. Our team explored search-based path planning by developing BFS, and A*, before deciding to integrate A* onto our race car.

We began by discretizing our occupancy map of the State basement to create our search domain for A* to parse through. We also eroded the map's edges to add clearance around obstacles so the robot would be able to avoid collisions in reality. Beginning at the start node, A* computes the total cost of each of the neighboring nodes by summing the cost of a node to the start point and the estimated cost of a node to the goal. We utilize Euclidean distance, which is the shortest distance between two points, in our heuristic function. Then the algorithm examines the neighboring node with the lowest cost and subsequently examines its neighbors, eventually constructing a path that has the lowest cost from the start to the goal pose. We recognized that A* possesses shortcomings, so we also explored sample-based planning and plan to implement RRT* with Dubin curves so our algorithm is able to better respond to the dynamics of the racecar. After we constructed our trajectory, we implemented pure pursuit to enable our car to drive along the predefined path. Pure pursuit finds the intersection between a circle defined by our look ahead distance and the trajectory to find the intersection point. Then, it computes angular velocity commands so the race car is able to reach the look ahead point. The look ahead point continuously moves along the path, allowing the car to follow along.

Editor: Maxwell Zetina-Jimenez

2.1 Search-Based Path Planning

Author: Maxwell Zetina-Jimenez

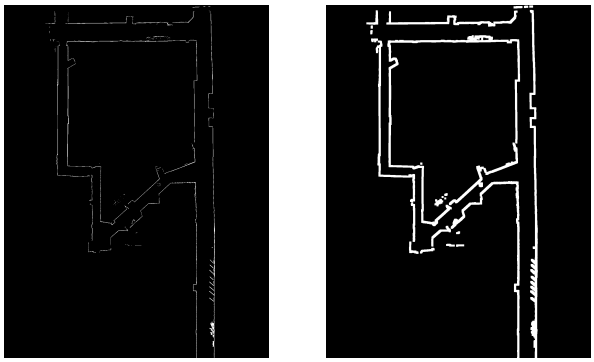
One strategy to path planning is using graph-search algorithms. An advantage of these algorithms is that they are complete, meaning that they terminate, find a path when one exists, and indicate failure when a path does not exist. Furthermore, these algorithms find the shortest path from the start to the goal. However, graph search requires a discrete search space. Therefore, search-based path planning can be tackled by first discretizing the continuous space into a graph and then by searching the graph to find a path. In this lab, we explored a dynamic discretization method and two graph-search algorithms: BFS and A*.

Editor: Trevor Johst

2.1.1 Map Discretization and Erosion

Author: Maxwell Zetina-Jimenez

The map of Stata basement was represented as an Occupancy Grid where each pixel in the map's image had a probability of being occupied. (Probabilities ranged between 0 and 100 where 0 indicated a free space and 100 indicated an obstacle.) The first step in path planning in this map was performing a precomputed erosion so that pixels close to an occupied pixel would also be labeled with a high probability of being occupied. We chose to do this image processing to enlarge the obstacles in the map. As seen in Fig.1, the erosion's goal was to thicken the walls of the original hallways so that any path planner could instead search the eroded map and not find paths that would be against the wall or too close to corners, which would make it difficult for the robot to follow and execute.



(a) Original Map (b) Eroded Map

Fig. 1: **Original Map vs. Eroded Map** Eroding the map with a Disk element (8 pixel radius) allowed for finding feasible trajectories that were not too close to walls and could be better executed by the robot in real space.

To discretize the space, rather than each pixel be a node, nodes were the (x, y) real coordinate of the center of each 1 meter by 1 meter unit square in the map. A certain node u at point (x, y) has eight edges, which connect to the nodes at the centers of the eight surrounding cells at points $(x-\Delta, y)$, $(x, y+\Delta)$, $(x+\Delta, y)$, $(x, y-\Delta)$, $(x+\Delta, y+\Delta)$, $(x+\Delta, y-\Delta)$, $(x-\Delta, y+\Delta)$, $(x-\Delta, y-\Delta)$ where $\Delta = 1$.

However, the uniform spacing of the nodes in the graph changed if near an obstacle — when the corresponding (u, v) pixel of the node's (x, y) coordinate has a probability of being occupied. If at least one of u's neighboring nodes is determined to be an obstacle, the new eight neighbors would be the (x, y) points at a distance of $\Delta/2$ away. For this dynamic graph, the initial choice of $\Delta = 1$ step size was made as a greedy choice: if there is no obstacle anywhere nearby, the search could traverse more physical space using less nodes. The logic behind the compression to a step size of $\Delta/2$ was that if one node is at an obstacle location, then we thought taking smaller steps (while preserving being a factor of the original step size) would help find a path through tighter spaces between obstacles. This discretization behavior can be seen in Fig.2.

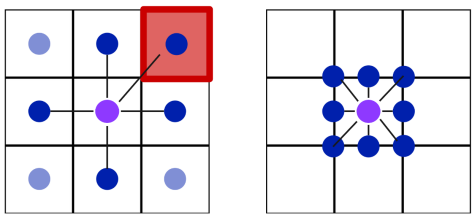


Fig. 2: **Dicretization of Search Space** For a given node (in purple), if at least one neighbor node is an obstacle (in red), then the remaining nodes (light blue) will no longer be checked and instead the new eight neighbors are closer to the node. This allows the robot to take advantage of free space yet find paths in tight spaces.

Editor: Trevor Johst

2.1.2 *Breadth-First Search*

Author: Maxwell Zetina-Jimenez

BFS is a linear time algorithm that finds the shortest path from a start node to a goal node in an unweighted graph. In the context of the lab, we implemented BFS to path plan from the starting pose to the goal pose in our discretized search space. As described in Algorithm 1, BFS keeps track of the nodes (the (x, y) coordinates in the map), it visits and each node's parent. The algorithm thus starts at the source node and adds each neighbor to a queue of nodes to visit next. Once visited, a node is added to the set of visited nodes so that a node is only visited if it has not been visited before. BFS continues to grab nodes from the queue and expand outwards like this until either the goal node is reached or the queue becomes empty (the whole graph is searched). If the goal node is reached, then the path is traced back through the parents of each node back to the start node. Otherwise, an empty path is returned, indicating the lack of a path from start to goal.

Algorithm 1 Breadth-First Search

$visited \leftarrow \{x_{start}\}$
$queue \leftarrow [x_{start}]$
while queue $\neq \emptyset$ **do**
  $x_i =$ QUEUEPOP( )
  if ISGOAL(x_i) **then return** PATHFROM(x_i)
  end if
  for x_j **in** NEIGHBORS(x_i) **do**
    if $x_j \notin visited$ **then**
      $visited \leftarrow \{x_j\}$
      $queue \leftarrow \{x_j\}$
      $parent[x_j] = x_i$
    end if
  end for
end while

In practice, this algorithm would take the starting point (x_s, y_s) and goal point (x_g, y_g) and associate them to their closest node in the discretized space $(x_{s,closest}, y_{s,closest})$ and $(x_{g,closest}, y_{g,closest})$. BFS would then search the space until the goal was reached. As shown in Fig. 3, this resulted in paths that were not optimal in terms of dynamics for the robot's driving; due to the discretization of the search space, there were many sharp changes in direction and corner-cutting in the paths.

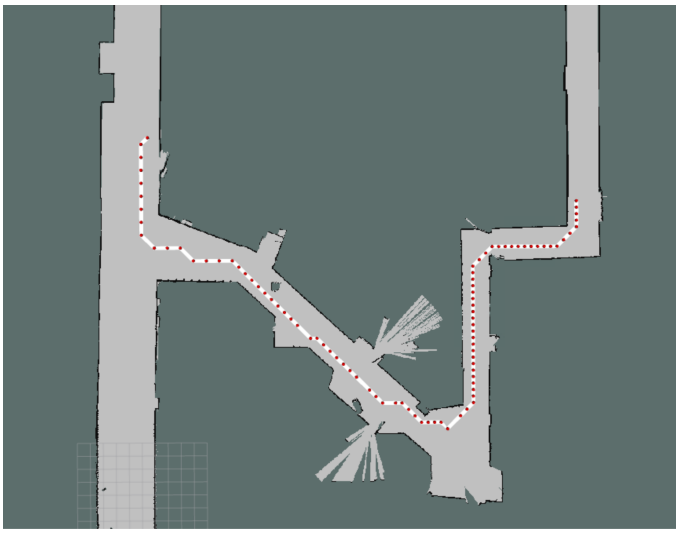


Fig. 3: **A path generated by BFS** Because of the discretization necessary to be able to search the space with BFS, paths are not as smooth for driving feasibility. There are staircase-like segments are sharp-angled turns.

Editor: Trevor Johst

*2.1.3 A**

Author: Sameen Ahmad

The A* algorithm is a prominent path finding technique that employs a heuristic approach to effectively navigate graphs and generate a path from a starting point to a goal.

A* combines aspects of Djikstra's Algorithm and Greedy BFS to generate a path that consists of nodes with the lowest cost. Dijkstra's Algorithm examines nodes that are closest to the start point and expands its search outwards until it comes across the goal. Then, it generates the shortest path from the start point to the goal. However, the algorithm is often computationally slow. Greedy BFS generates a path by examining nodes that are closest to the goal and working backwards to construct a path to the start point. It relies on a heuristic (an estimation method) of the distance of a vertex from the goal when considering which nodes to explore first.

A* aims to generate a trajectory with the lowest cost from the start and end states. Cost is a measure of the resources that the algorithm requires to reach a given node, which is a possible position that the car can occupy as defined by the search domain. The total cost of a node also depends on a heuristic function, which estimates the cost of a node to the goal. Beginning at the start node, the algorithm calculates the costs of neighboring nodes and constructs a path based on nodes with the lowest cost.

A* computes the total cost of the node with the following equation

$$f(n) = g(n) + h(n) \tag{1}$$

Here, $f(n)$ is the total cost of any given node n. $f(n)$ is the sum of the exact cost from the start to n, represented by $g(n)$ and the heuristic estimated cost from n to the goal, denoted as $h(n)$. We define $h(n)$ as the distance between the start point and the end point.

There were several possibilities to compute distance including Manhattan, Diagonal, and Euclidean.

Manhattan distance is the standard heuristic for square grid that allows four directions of movement. It computes the distance, d, between adjacent grid spaces by

$$d = \Delta x + \Delta y \tag{2}$$

Diagonal distance is applicable on square grids that allow for eight directions of movement, including diagonal motion. This can be computed by

$$d = (\Delta x + \Delta y) * \texttt{min}((\Delta x, \Delta y)) \tag{3}$$

Euclidean distance computes the shortest distance between two points given that movement at any angle is allowed. It is calculated as

$$d = \sqrt{(\Delta x^2 + \Delta y^2)} \tag{4}$$

We chose the Euclidean distance as our heuristic metric as it avoids overestimation, unlike Manhattan and Diagonal distances.

Given a search domain, A* aims to minimize the cost of the path by balancing $g(n)$ and $h(n)$ for each node. Our implementation relies on a `PriorityQueue()`, which is a data structure where each element is associated with a priority. We build a queue to store the nodes that are candidates for examining. We also initialize a dictionary to keep track of each node that has been visited and their neighbors, linking a parent node with its children. The algorithm first creates a start node and adds it to the priority queue. Then, it iterates until the queue is empty, extracting nodes and exploring their neighbors. If the goal node is found, it returns the path. Otherwise, for each neighbor, it checks if it has been visited before. If not, it creates a new node and adds it to the queue. If the neighbor has been visited, but a shorter path has been found, it updates the node's scores and reinserts it into the queue. This process is described in Algorithm 2.

After comparing the performance of BFS and A*, we decided to integrate A* onto the race car since it is computationally faster and more efficient at generating optimal trajectories, which are the shortest path for our level of discretization.

Editor: Ellen Zhang

2.2 Sample-Based Path Planning

Author: Trevor Johst

The primary downside of many search-based algorithms is their inability to account for the dynamics of the vehicle. When a planner such as A* finds a path from the start to the goal, it treats the world as a discrete grid with no regard for the orientation of the robot or how it travels. It is possible to extend these algorithms into higher dimensions and account for dynamics, but this can cause a serious increase in computation time.

Sample-based path planning differs from search-based approaches in that it operates on random samples of the configuration space (c-space). This can offer a number of advantages such as faster computation, the ability to incorporate dynamics, and online planning. Treating our robot as a 2D car with Ackermann steering we will have either an $\mathbb{R}^2$ c-space without dynamics, or an $SE(2)$ c-space with dynamics.

Algorithm 2 A*

```
nodelookup ← {x_start}
queue ← [x_start]
while queue ≠ ∅ do
    node ← QUEUEGET( )
    if ISGOAL(node.pose) then
        return PATHFROM(node)
    end if
    for n in NEIGHBORS(x_i) do
        if n ∉ nodelookup then
            n_obj ← new Node
            nodelookup[n] ← n_obj
        else
            n_obj ← nodelookup[n]
        end if
        tentative_gscore = gscore + hscore
        if tentative_gscore < n_obj.score then
            n_obj.gscore ← tentative_gscore
            n_obj.fscore ← tentative_gscore + hscore
            if n_obj ∉ queue then
                QUEUEPUT(n_obj)
            end if
        end if
    end for
end while
```

Two popular sample-based path planning algorithms are Rapidly-exploring Random Trees (RRT) and Probabilistic Road Maps (PRM). RRT is a single-query planner that builds a tree from the start to goal and returns a single path. PRM builds a map of nodes on the occupancy map and can find multiple paths across it using standard search-based methods.

Editor: Sameen Ahmad

2.2.1 Dynamics

Author: Trevor Johst

To leverage the advantage of traversable paths, we chose to implement RRT with Dubins curve dynamics [1]. Similar to the search-based approach, we use the same occupancy map to determine obstacles in the environment. If we erode our map by a sufficient amount such that the robot could feasibly occupy any position, our resultant trajectories will be both dynamically possible and collision free. For our implementation we used the `pydubins` library, which constructs these paths between any two states when given a turn radius.

A Dubins curve is the shortest possible path between two points in a two-dimensional plane when a constraint is put on the maximum curvature of the path, as shown in Fig. 4. By specifying both the initial and final state, $\mathbf{x} = [x, y, \theta]$, we are able to generate a dynamically plausible path between them. The lower bound for the turn radius could be determined from the maximum steering angle and some geometry. In reality these small radii turns can still prove difficult to follow, so a much larger value of 0.4 m was used for the lower bound as determined through experimentation.

Editor: Sameen Ahmad

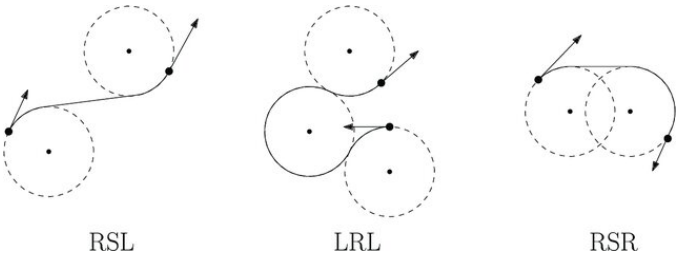


Fig. 4: **Examples of Dubins curves between various points.** Each path is generated following the curvature of a circle with a set radius, and connecting the circles via their tangents.
Source: [2]

2.2.2 RRT*

Author: Trevor Johst

The concept behind RRT* is to sample the configuration space randomly, connecting states as you do so to generate a tree. Starting from the initial state, the tree will radiate outwards and explore new areas. When the tree is finally connected to the goal, you know that a path exists from the start to the goal and no search is necessary. An example path is pictured in Fig. 5.

Fig. 5: **A path generated by RRT* with dynamics.** The initial pose of the robot (the green node) was facing towards the top of the image. The robot is able to preemptively turn to make the tight corner.

The * at the end of RRT* indicates some form of optimality. Since RRT probabilistically constructs its tree, this is only guaranteed if also rewire the tree to remove longer paths and then run it for an infinite amount of time. RRT* is only asymptotically optimal, meaning it approaches the optimal solution but never reaches it. This addition of the rewiring step will produce smoother paths, but will never give a truly optimal result in finite runtime.

Sampling a point simply means generating a random position. To speed up the collision detection phase, we

do all of our sampling and planning from the perspective of the occupancy map, and transform the result to the world frame. This means we are able to just generate an integer value using `random.randint` that represents the coordinates of some pixel on the occupancy map. The angle of that position can then be randomly generated as a float using `random.uniform` between $-\pi$ and π.

It is important that we also sample the goal position with some fixed probability. This effectively biases the path towards the goal during the steering phase, and also makes it easier to reach the exact end goal position. A sample rate of 10% was chosen through experimentation.

Once the point is sampled, it must be connected to the tree. A search is conducted through every node comparing distances, and the nearest node is found. For the sake of performance, the Squared Euclidean Distance (SED) is used. Once this nearest node is found, the aforementioned Dubins curve is generated between our two states for the steering phase. Finally, the curve is interpolated to a set maximum distance. This interpolation forces the tree to radiate outwards from already explored portions of the map and also results in smoother final paths.

To determine if a path is collision free, we want to iterate over every point on the path and individually check if they are obstacles on the occupancy map. Because our points are often interpolated to the curved portion of the Dubins path, we can do some broad phase collision detection by checking the middle-most point on the path first. We then iterate over the path backwards, since it is known that the start of the path is already in our tree. These two actions let us conduct collision detection rapidly by ruling out paths if their most-likely-to-collide points are occupied.

The final step in the RRT* loop is to rewire the tree, as described in Algorithm 3. The initial step in this phase is to find every other node that is within some distance of the robot. To determine this radius, we use the formula described in [3], which is outlined as

$$R = \min \left[\left(\gamma \frac{\log N}{N} \right)^{1/d}, \eta \right] \tag{5}$$

where N is the number of nodes, d is the dimension, and η is the set maximum rewiring distance. The value γ is a constant determined by the free space of the occupancy map and is calculated as

$$\gamma = 2^d * \left(1 + \frac{1}{d}\right) * \frac{\pi n a^{-2}}{v_{ball}} \tag{6}$$

where n is the number of free tiles in the occupancy map, a is the resolution of the occupancy map, and v_{ball} is the volume of the unit ball in your dimension d.

Algorithm 3 Rewire

$R = \text{CALCRADIUS}(len(N))$
$N_{near} = \text{FINDNEAR}(R)$
for x **in** N_{near} **do**
  $path = \text{STEER}(x, x_{new})$
  if $\text{COLLISIONFREE}(path)$ **then**
    $newCost = x.cost + \text{COST}(path)$
    if $x_{new}.cost < newCost$ **then**
      $x_{new}.parent = x$
      $x_{new}.cost = newCost$
      $x_{new}.path = path$
    end if
  end if
end for

Once this distance is determined, we find every node with an SED of less than R^2. For every node within our search radius, we then construct a new Dubins path from the node we are rewiring. If a new Dubins path offers a shorter distance from the start, it will become the new parent node of the node we are rewiring, changing the tree. This distance is tracked by having every node track the path length from the start to that node, reducing computation time for rewiring significantly.

This loop is repeated until a full path to the goal is found. Once it is is found, we are able to simply reconstruct the path backwards from the goal to the start and return it. The full algorithm is outlined in Algorithm 4.

Algorithm 4 RRT*

$tree \leftarrow x_{start}$
for $i = 1$ **to** n **do**
  $x_{rand} = \text{SAMPLE}(\ )$
  $x_{near} = \text{NEAR}(x_{rand}, tree)$
  $x_{new},\ path = \text{STEER}(x_{rand}, x_{near})$
  if $\text{COLLISIONFREE}(path)$ **then**
    $tree \leftarrow x_{new}$
    $tree = \text{REWIRE}(tree, x_{new})$
    if x_{new} **is** x_{goal} **then**
      return $\text{PATHTO}(x_{new})$
    end if
  end if
end for

Editor: Phillip Johnson

2.3 Trajectory Following

Author: Phillip Johnson

Once a path is found, the new task becomes creating a controller that can quickly follow the created path. To do this, different types of controllers were considered including PID and various pure pursuit algorithms. We eventually settled on a pure pursuit algorithm which works by following the point at which the path intersects with the look-ahead circle. This section will walk through the decision process and theory behind the different controllers and why the eventual decision was made about the current algorithm.

2.3.1 Follower Considerations

Author: Phillip Johnson

Three algorithms were considered when deciding how to follow the provided trajectory:

1) PID
2) Pure pursuit based on intersection points
3) Pure pursuit based on path intersection

This section will discuss the theory behind each algorithm and why it was either chosen or not chosen.

2.3.2 PID

Author: Phillip Johnson

PID (which stands for proportional-integral-derivative) control is a very well-studied controller for actively adapting to the control subject's environment. Proportional control works to minimize the control error with the following equation

$$u = -k * x \tag{7}$$

In this case, x would be the difference between the desired angle toward the next point and the actual angle. Thus, the proportional control for the robot is better represented by the following equation, where the $\tan^{-1}$ is a four quadrant arctan

$$u_p = -k_p * \tan^{-1} \frac{y_{actual} - y_{desired}}{x_{actual} - x_{desired}} \tag{8}$$

The derivative control in this case is shown as

$$u_d = -k_d * \dot{u_p} \tag{9}$$

Finally, the integral control can be shown as

$$u_i = -k_i * \int u_p \tag{10}$$

The PID algorithm worked well in simulation. However, the multiple environments that the robot needs to travel at high speed meant that tuning the gains would be incredibly difficult to do. Thus, the decision was made to switch to pure pursuit control.

Editor: Maxwell Zetina-Jimenez

2.3.3 Basic Pure Pursuit

Author: Phillip Johnson

The pure pursuit controller works by projecting a point onto a circle at a set distance and finding the angle between the car's position and the projected point. This process is represented in Fig. 6.

As seen in Fig. 6, the pure pursuit algorithm minimizes η which is the angle between the robot and the intersection of the path on the look-ahead circle.

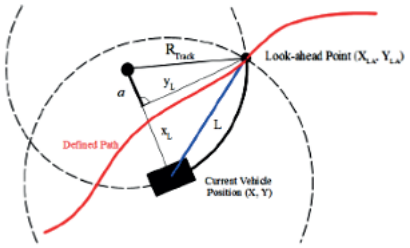


Fig. 6: **Basic Pure Pursuit Controller [4]** The lookahead point is found by projecting a point onto a circle.

The pure pursuit algorithm is shown in Algorithm 5.

As shown in Algorithm 5, the basic pure pursuit works by first finding the closest path point $x_{closest}$ relative to the car's pose x_k. Then, the slope to that point is found and projected onto the circle with the radius equal to the look-ahead distance. Finally, the angle between the car and the projection is found and passed in as the drive control.

Algorithm 5 Pure Pursuit

$x_{closest} = \text{FINDCLOSEST}(x_k)$
slope $= \frac{x_{closest}[1]}{x_{closest}[0]}$
intersection $= \text{PROJECTCIRCLE}(slope, lookahead)$
$\delta = arctan2(2 * wheelbase * intersect[1], lookahead^2)$
$\text{DRIVE}(speed, \delta)$

2.3.4 Pure Pursuit with Intersection Points

Author: Phillip Johnson

The initial approach to trajectory following was to prune the linear path output by the A* algorithm to only contain line segment intersection points. Then, the basic pure pursuit controller could be used with the remaining path points.

On the car, this can be represented in simulation as shown in Fig. 7. Here, the black point is the projected look-ahead based on the purple intersection point that is closest to the robot. Projecting a far look-ahead allows for minimal turning angle change, but controlling based on closely-spaced points leads to problems with overturning. This can be seen in Fig. 8.

As shown in Fig. 8, the circle projection creates an extreme turning angle with tightly-spaced points. If the path that was created was simple line segments, this algorithm would be sufficient. However, future path planning will use dynamics and non-linear paths, so a different pure pursuit algorithm must be used.

Editor: Maxwell Zetina-Jimenez

2.3.5 Pure Pursuit with Path Intersection

Author: Phillip Johnson

To handle tightly-spaced points, an update to the pure pursuit algorithm was implemented to instead focus on the intersection between the path and the look-ahead circle. This is shown in Algorithm 6 and works by searching line segments between the closest two points and finding the distance of the intersection point on the circle. Finally, the index of the point corresponding to the segment with the closest circle intersection is returned.

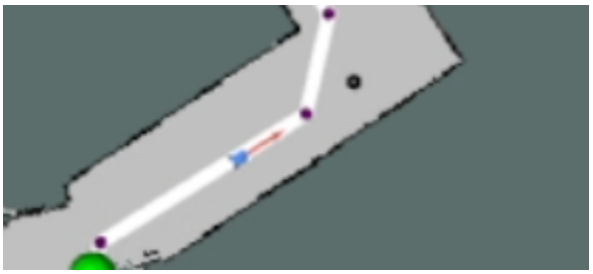


Fig. 7: **Pure pursuit based on intersection point.** The car follows the closest intersection point through calculations based off the slope.

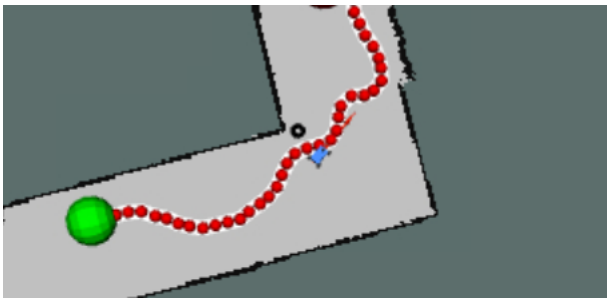


Fig. 8: **Intersection-based pure pursuit fails with non-linear path.** The car would turn towards the black intersection point here which would cause it to go off course.

This update to the algorithm ends up with a pure pursuit follower as demonstrated in Fig. 9. With this update, the pure pursuit algorithm can reliably follow non-linear paths. The next section will evaluate this controller and how reliably it is able to follow the path.

Algorithm 6 Line Intersection

```
min_distance = ∞
closest_point = none
for x_i, x_{i+1} in path do
    distance = CIRCLEINTERSECT(x_i, x_{i+1})

    if distance < min_distance then
        min_distance = distance
        closest_point = x_i

    end if
end for
return min_distance, closest_point
```

Editor: Maxwell Zetina-Jimenez

3 Experimental Evaluation

Author: Ellen Zhang

The path planning and trajectory following are integrated to create a race car program that effectively and efficiently computes and then follows trajectories from its initial pose to the goal pose. In this section, we quantitatively analyze various path planning algorithms (A*, BFS, RRT, RRT*) in simulation and pure pursuit with path intersection to evaluate the performance and discuss possible improvements of the model.

Editor: Sameen Ahmad

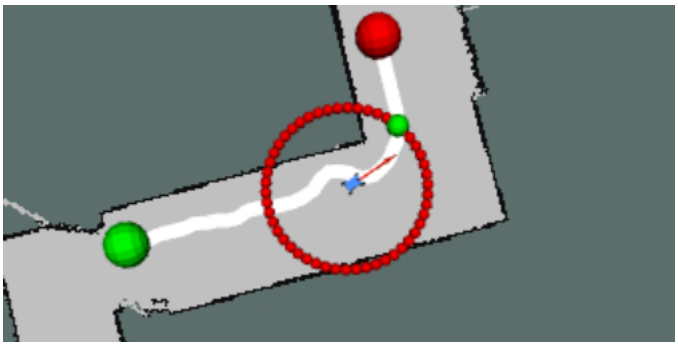


Fig. 9: **Pure pursuit based on path intersection.** The look-ahead point is shown in green, and is found by intersecting the trajectory with a dynamic look-ahead distance.

3.1 Simulation

Author: Ellen Zhang

To evaluate the performance of path planning (BFS, A*, RRT, RRT*) and trajectory following (pure pursuit with path intersection) in simulation, we implement a variety of metrics.

3.1.1 Path Planning

Author: Trevor Johst

When comparing path planning algorithms, the primary metrics of interest are computation time, trajectory traversability, and path length. Computation time and path length are rather straightforward to compare, but must be averaged across multiple runs for probabilistic planners such as RRT*.

Traversability is more difficult to quantify, but can be estimated based on the physical path that is produced. A planner that incorporates dynamics will inherently be traversable in ideal scenarios, and all planners can be visually inspected for sharp corners or close encounters with walls as is done in Fig. 10.

Computation time was recorded over four different set paths, and each path was averaged over 15 runs. Each test was chosen to compare how both path length and number of turns impacted runtime. The results of the tests can be seen in Table 1. A* consistently performed better than BFS, which makes sense considering the addition of the heuristic.

RRT* is harder to compare, as it did better in some situations and worse on others. It specifically struggled with the narrow corners in the bottom right of the map, which test 4 was entirely focused on. It is also worth noting that RRT* almost always performed worst when considering the maximum time and path length.

To determine how smooth a path is, generated trajectories were compared for the same initial conditions, as shown in Fig. 10. Across all of the tests it was generally noted that both A* and RRT* cut corners the closest.

It is especially worth noting test 3, which was purposefully chosen to demonstrate another advantage of dynamics. As shown in Fig. 11, A* finds a shorter path from the start to the goal, but it is not dynamically feasible. The robot was initialized pointing upwards in the hallway, which means the path generated by A* is actually impossible to traverse through pure pursuit alone.

Algorithm	Test	Time [s]		Length [m]	
		Avg	Max	Avg	Max
BFS	1	7	7.95	66.31	66.31
	2	8.62	9.96	58.36	58.36
	3	4.27	5.9	43.81	43.81
	4	2.33	2.74	25.04	24.04
A*	1	4.51	5.5	67.17	67.17
	2	4.59	5.26	59.73	59.73
	3	2.06	2.39	43.05	43.05
	4	0.6	0.93	24.24	24.24
RRT*	1	4.53	18.42	71.66	96.11
	2	1.14	2.62	62.68	66.73
	3	5.84	17.26	81.01	112.56
	4	9.48	74.61	40.39	137.61

TABLE 1: **Computation times for our three path planning algorithms averaged over 15 runs.** Test 1 had three turns, 2 had one, 3 had four turns if dynamically viable, 4 had two turns.

Fig. 10: **Our three algorithms all finding a path on the same initial conditions.** RRT* and A* seem to cut some of the corners too closely. Because RRT* incorporates the dynamics, these trajectories should be possible, but they are still worth noting.

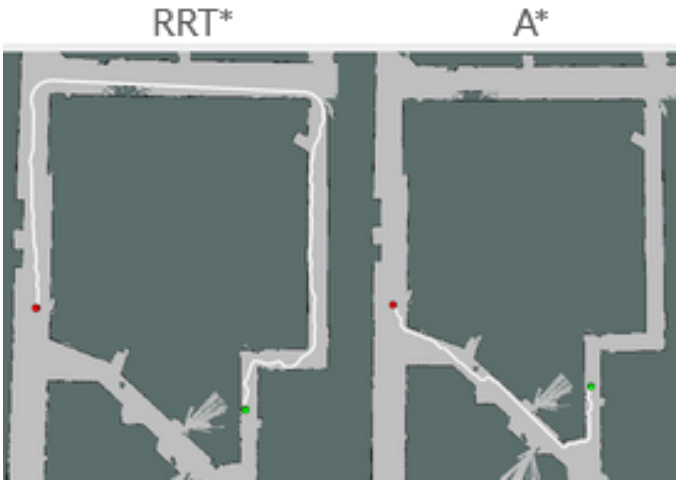


Fig. 11: **RRT* and A* both planning for test 3.** Although the path for A* is shorter, the robot was initialized as pointing upwards from the start (green node). Because of the required turn radius, the robot would not be able to follow A*'s path.

Editor: Sameen Ahmad

3.1.2 *Trajectory Following*

Author: Ellen Zhang

To evaluate our pure pursuit with path intersection performance, we followed a predetermined path in the Stata basement as shown in Fig 12. We collected various numerical metrics. For our purposes, the accuracy of the racecar in following the trajectory is defined as the difference between the y coordinate of the car y_{car} and the y coordinate of the closest trajectory point in front of the car $y_{closest}$:

$$error = y_{car} - y_{closest} \tag{11}$$

We use the difference in y coordinates instead of the x coordinates because in the car's coordinate system, the y axes are the left and right directions.

Furthermore, in order to analyze the trajectory of the car when making turns, we plot the slope of the closest point as well:

$$slope = \frac{y_{closest}}{x_{closest}} \tag{12}$$

A high absolute value in slope indicates that that the closest trajectory point in front of the car is to its left or right, and that the car should make a turn. We plot both the error from trajectory and slope in Fig 13 and Fig 14.

Upon observation of the racecar's trajectory in simulation, the racecar follows the path consistently, with no noticeable deviations. This is evidenced in the plot, where the absolute value of the error from trajectory never goes above 0.5 meters, and generally stays between -0.2 and 0.2 meters. There are two peaks, notably, at around 13 and 17 seconds, in both the error and slope graphs. These correspond to when the car is making a right-hand and left-hand turn, respectively, in which case, the error tends to be large because of the nature of our error formula, which takes the differences in y coordinates. In turns, even though the closest point is very close to the racecar, there will still be a noticeable difference in the y coordinates because the two positions lie on a curve instead of a straight line. This may be an indication that our error metric might be improved. Overall, the importance takeaway is that our error remains consistently low, especially on straight lines where the formula applies best, indicating that the pure pursuit algorithm works well in following a trajectory.

Editor: Sameen Ahmad

3.2 Physical Implementation

Author: Ellen Zhang

Because the path planning operates in code, we only need to evaluate the performance of the pure pursuit algorithm. We test the racecar on the same path in real life as in simulation.

Editor: Trevor Johst

3.2.1 *Trajectory Following*

Author: Ellen Zhang

We collect data for the same performance metrics, error from trajectory and slope, for the racecar on the same trajectory. The resulting plots are shown in Fig 15 and Fig 16.

Fig. 12: **The racecar following the trajectory in simulation.** The red dot indicates the closest point, from which we calculate the error of the pure pursuit algorithm in following the trajectory

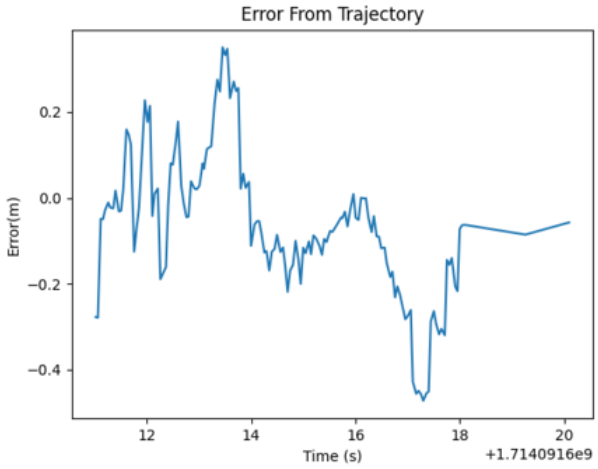


Fig. 13: **Plot of simulation error from trajectory vs time, with error on the y axis.** The error stays relatively low, peaking when the racecar makes turns due to the nature of our error formula.

The graphs have a noticeable difference when compared to the simulation graphs, particularly the error graph. Overall, the commands calculated such as the turning angle don't change, but the way the car behaves in real life is different than in simulation. For example, our racecar has a consistent left bias when driving, and thus we add an offset of 0.5 in our turning angle in real life to mitigate it. Because the car behaves differently in real life than in simulation, the program calculates different errors and slopes, which explains why the graphs look different. Based on our observations, the car still follows the path relatively well, albeit overturning on the right-hand turn in the beginning, which

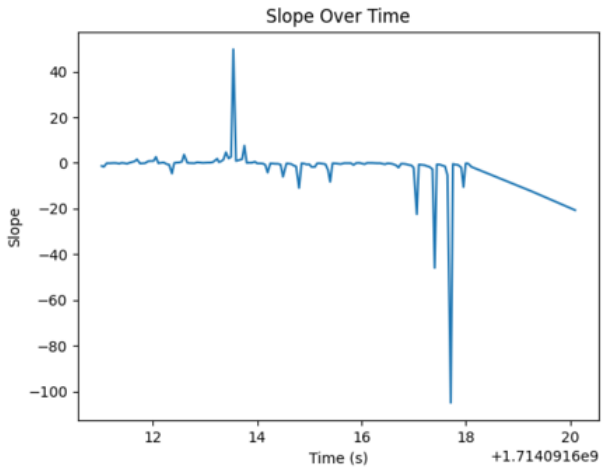


Fig. 14: **Plot of simulation slope of closest point vs time, with slope on the y axis.** Higher absolute value slope indicates the car should make a right or left turn, depending on the sign.

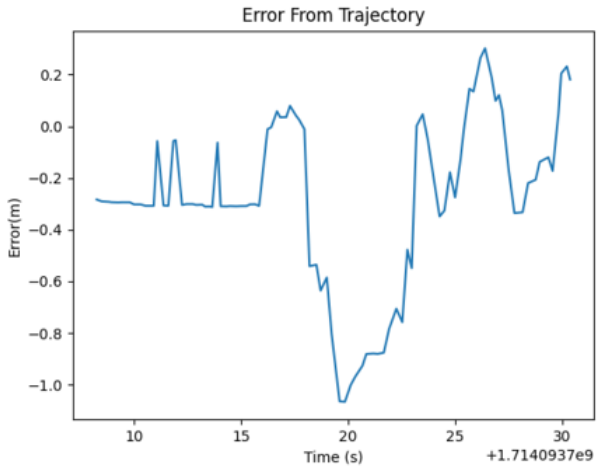


Fig. 15: **Plot of error from trajectory vs time, with error on the y axis.** The error has the same upper bound of 0.2 as simulation, but a lower bound of -1.0 at one point.

is likely the cause of the high error at around 20 seconds, when the car is off trajectory. Except for that area of the graph, the rest of the errors remain between -0.2 and 0.2 meters. Further testing is needed to help adapt the model to real-life factors and situations, particularly when turning corners.

Editor: Trevor Johst

4 Conclusion

Author: Sameen Ahmad

In this lab, we explored search-based and sample-based planning algorithms through BFS, A*, and RRT*. We implemented A* with a Euclidean distance heuristic onto our autonomous race car, allowing us to generate the shortest paths from the current pose of the robot to a goal pose. Additionally, we integrated pure pursuit controls to allow our car to precisely follow the predefined trajectory.

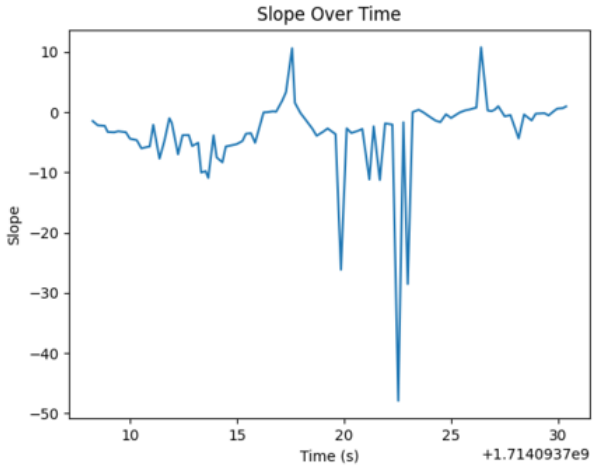


Fig. 16: **Plot of slope vs time, with slope on the y axis.** The overall trend of the slope is relatively similar to the one in simulation.

Our experimental evaluation in simulation and our physical implementation demonstrate the efficiency of our algorithm and controls methods. In simulation, our race car precisely and consistently follows the trajectory in increasingly complex layouts. Our physical implementation shows minimal deviation from the path, revealing areas of future improvement.

Looking forward, we plan to further implementing RRT* with Dubin curves so our system is able to better account for the kinematics of the car during travel. We hope this will make for a more accurate and robust path planning algorithm that advances our robot's capabilities so it is able to react appropriately to the complex environments of the final challenge.

Editor: Trevor Johst

5 Lessons Learned

5.1 Sameen Ahmad

Through this lab, I was able to explore the different types of algorithms and learn more about each of their strengths and shortcomings. I especially found the A* algorithm interesting and enjoyed the opportunity to practice its implementation. It was really rewarding to see the robot follow the predefined trajectories in real time. While testing, I learned that the simulation does not account for dynamics introduced by reality, so additional adjustments are necessary to properly tune our system. I also realized the importance of communication when integrating algorithms on the car and brainstorming ideas to resolve issues.

5.2 Phillip Johnson

One lesson I learned is that tuning and testing algorithms in simulation is far more efficient than in real life. Being able to test parameters quickly is very important, and the simulation allowed for that. However, I also learned that even perfect controllers in simulation may fail in real life and so perfecting quantities in simulation is not necessary for implementing the code onto the robot as those quantities will certainly need to change.

5.3 Trevor Johst

I enjoyed this lab because I was able to put a lot of time into developing and thinking about algorithms to help the robot do actual high level planning. The most valuable thing I learned was probably the methods I used for optimizing the RRT* algorithm. I have had experience with various forms of path planning before, but I have never had to design them for real time use on a robot. These actual implementations teach a lot about real world considerations.

5.4 Maxwell Zetina-Jimenez

This lab was great because I enjoyed putting theory into practice. After studying so many algorithms, it was nice to actually see them applied in a real setting. However, I learned that from theory to application, it is not a clear and direct path. There are modifications to be made to fit different problems, but nonetheless, it was great. I think I also learned that even though everything could be working in simulation, it does not mean it will immediately work perfectly in real life. But in those moments of debugging, it is best to have your teammates who can offer different solutions, and that is a great advantage to teamwork.

5.5 Ellen Zhang

In this lab, I was most interested in the pure pursuit as it was something I had never done before and I saw it as a new challenge. Taking inspiration from the work of my teammates, I got to incorporate new ideas and test them in real life, and it was very exciting to put together, especially when the racecar started working after much debugging. I realize the importance of collaboration with the team, as all of us have different ideas and it is important to bounce them off of each other.

References

[1] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of Mathematics*, vol. 79, no. 3, pp. 497–516, 1957. [Online]. Available: http://www.jstor.org/stable/2372560

[2] A. Nayak and S. Rathinam, "Heuristics and learning models for dubins minmax traveling salesman problem," *Sensors*, vol. 23, p. 6432, 07 2023.

[3] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011. [Online]. Available: https://doi.org/10.1177/0278364911406761