Lab 6 - Path Planning and Pure Pursuit

6.141 Spring 2020

Team 3

Aaron Garza, Aaron Makikalli, Edward Rivera, Kevin Sun, & Kuan Wei Huang

1. Introduction (Aaron Makikalli)

Any truly autonomous vehicle must be able to plan its trajectory based on its surroundings. Autonomous vehicles operate in dynamic environments characterized by randomness and uncertainty, so their path-planning algorithms must operate in real time based on current sensor readings rather than simply follow a precalculated trajectory. The motion planning process can be broken down into several steps: Environment recognition, localization, path planning, and path execution. In this lab, we focused on path planning and execution.

Our team's primary goal in this lab was to determine a path from the robot to the desired destination and follow that path as accurately as possible. As a general motivation for this objective, all autonomous vehicles must be capable of recomputing their optimal trajectory in order to account for changes to the environment and imperfections in the robot's sensor readings and motor outputs. The RSS final challenge is a more specific motivation for this lab, as our car will need to plan valid paths quickly in order to safely navigate an obstacle course.

Our solution to this lab is organized into two major sections: path planning and pure pursuit. In the path planning section, our robot computes a valid and efficient path from the robot's current position to a specified goal pose using a probabilistic roadmap and an A* search algorithm. Then, pure pursuit uses a path-tracking algorithm to constantly update the car's steering angle and velocity as it navigates from point to point along the trajectory. These two sections work in concert with each other to safely bring the robot to its destination.

2. Motion Planning (Edward Rivera) - Probabilistic Road Map (PRM)

Determining a valid, physically feasible path for a robot to reach a desired location is known as motion planning. Generally, there are a few classes of practical motion planners: state space search-based, potential field-based, and sample-based. They are often characterized by computational complexity/runtime and their completeness. An algorithm is said to be complete if it returns an optimal path or that there is no path if there isn't one. The following is the rationale for choosing the sample-based algorithm known as the Probabilistic Road Map (PRM).

A search-based method uses a discretized grid, which discretizes a map into a grid, but here lies the first issue. What if a cell in your grid is partially filled by an obstacle? If you label the cell as occupied, then your space of available points is dependent on the resolution of your grid. After searching this graph (each cell in the grid is a node and neighbors are immediate unoccupied cells) with a single source shortest path algorithm (SSP), the path returned is optimal according to the resolution. This is known as resolution complete. For the sake of argument, let's say the resolution is really good (meaning the grid has very little cells partially occupied), then the resulting path will be a series of small edges between cells. In a physical context, it is very hard to tell a car to travel short distances at a time (Figure 1). Therefore, the path must be sparsified in some way. The most common method is known as waypoint extraction, which goes through the

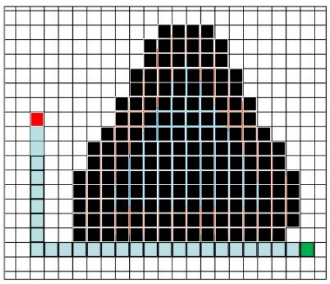


Figure 1: Waypoint Extraction

path and converts points along the same straight line into a single point called a waypoint. Although this creates a more physically feasible path, the runtime for the entire algorithm becomes exponential in the length of the path, which is not computationally feasible for this lab.

A potential field-based method suffers from optimality. The idea would be for obstacles to repel the car and for the goal to attract the car according to some mathematical relation (often based on distance). However, one could imagine a situation where the car was pushed into a U shaped obstacle (Figure 2). Each wall would apply pressure yielding a small trough otherwise known as a local minima, thus the car may get stuck and the algorithm fails to return an optimal path to the goal. The issue of local extrema vs global extrema is well characterized in numerous fields (e.g. mathematics, machine learning). As a result, the difficulty of constructing a mathematical relation that designs a field that is infinitely close to obstacles and smooth everywhere with a single global minimum at the goal made this option less viable for this lab.

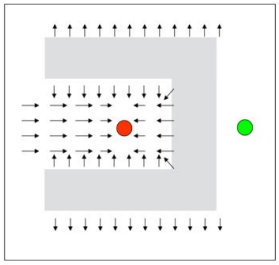


Figure 2: Potential Field

Although we are only considering practical motion planners, it is important to discuss configuration space methods to highlight an important feature of sample-based methods. Generally, a configuration space method relies on configuration space obstacles known as c-obstacles, which are inefficient to compute given they rely on a series of convex hull algorithms and Minkowski additions for every degree of freedom (DOF) (e.g. rotation, reflection, translation, etc.). Exact cell decomposition, an example configuration method, takes c-obstacle vertices and projects them to an axis and generates convex regions that span the space between the axis and obstacle (Figure 3). However, based on the inefficiently generated c-obstacles, the algorithm may also create an exponential number of regions resulting in a very impractical algorithm. However, this sheds light on an important feature of sample-based methods. Sample-based methods do not need to construct c-obstacles but rather than ignore the physical relationship between the car and obstacles, a sample-based method changes the perspective from obstacles to the car.

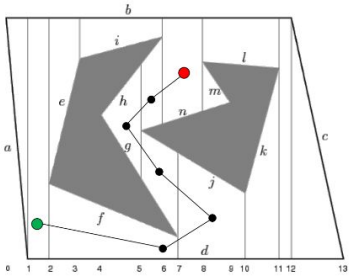


Figure 3: Exact Cell Decomposition

The PRM, a sample-based algorithm, can be divided into 2 main parts: **graph generation** and **path finding**. However, the PRM derives its power from the graph generation, which can be further subdivided into the **sampler** and **edge validator**. The sampler generates points/nodes or possible robotic poses along the map according to some scheme and the scheme takes into account the physical relationship of the car to the obstacle. For example, you can uniformly sample nodes on the spatial dimensions of the map and validate nodes if they are not occupied by an obstacle **and** if some set region around the sampled node is also obstacle-free. This is known

as circle-space or representing the car as a circle, which estimates the physical relationship of the car and obstacle in a variety of possible orientations thus abstracting away that DOF. Then, once the points are generated, the edge validator can draw edges between validated nodes and validate edges according to another scheme (e.g. edges are valid if they do not cross an obstacle). After this graph is constructed, path finding is conducted which can be separated into two more subparts: searching the graph and converting edges into motion plans. The graph can be traversed/searched with any SSP algorithm and a motion plan can be created for each edge on the optimal path (a straight edge is the easiest).

However, there are a few issues with the PRM. Firstly, it relies on the "visibility" of the map, or a map that does not have extremely confined spots thus putting pressure on the sampler to sample those exact spots so that straight-line edges may be drawn without hitting any obstacles. Secondly, the PRM assumes the edges are "straight line". In other words, PRM does not take into account the drivability of the course or the kinematic/differential constraints of the car. The last step of the PRM - generating a motion plan for each edge along the optimal path - may take kinematic constraints into account but the differential constraints that go beyond a single edge traversal are not considered. The RRT algorithm takes into account differential constraints as well as kinematic constraints by physically driving to points, but we did not choose RRT for two reasons. The environment we are given is assumed to be static and the car does not have many differential constraints. The RRT algorithm's flaw and power is that it explores to create the path, which may not always yield an optimal path. A PRM is closer to optimality with a good sampler, the map given is fairly "visible", and we ignored kinematic constraints (more on this later in the path finding section).

The specifics of our schemes in the graph generation and the details of our path finding are outlined in the following sections.

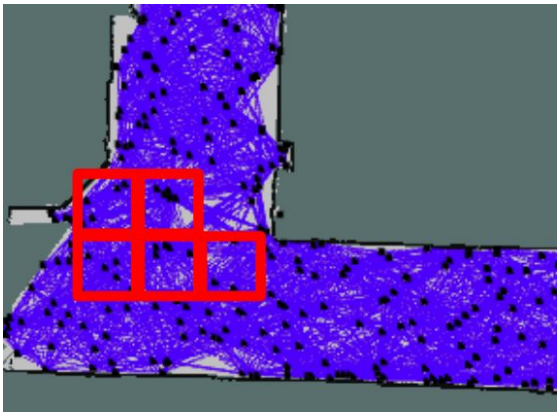
2.1 Graph Generation (Kevin Sun)

To plan a path between two points on a known map, we need to model the space our path searches over. However, any actual path taken by a robot follows a curve, and the validity of a path also depends on the controls allowed to the robot. As specifying a full path implies specifying a continuous object, we must settle for an approximation and somehow describe a path discretely. To simplify our problem, we chose to restrict our search space to paths in a graph, where vertices represent possible locations/intermediate points and edges between them represent pairs of intermediate points that the robot can choose to reach consecutively. As our robot can not follow a poly-line path, but rather a smooth path, our computed path will only be an approximation that our robot attempts to follow.

We now go into more detail on our exact choice for the graph we used as well as the method used to generate this graph efficiently. The map was given to us as a occupancy bitmap, marking locations that are free from those that contain an obstacle. All vertices were chosen from locations that are free in this occupancy map. Since our robot does not follow our exact

trajectory, we restrict our choice of vertices to locations which are not within a certain distance of any obstacle.

Our method of picking vertices is based on random sampling. We use an algorithm slightly more sophisticated than randomly sampling points in the map. We first divide up the map into 2500 blocks. In blocks that contain some unoccupied locations, we choose a few points. This allows us to get a more uniform spread of points across the map and make sure we don't miss any corners. We end up choosing about 5000 vertices total on the provided testing map.



The second part of our graph generation is to determine the edges in our graph. For simplicity, we will say that two vertices are reachable from each other if the straight line between them does not pass through any occupied points. Since we sampled many vertices, it is not computationally feasible to check all pairs of vertices and see if they cross an unoccupied area, as well as storing. This is also better for creating a path. Finally, it's left to assign a cost function to the edges for our pathfinding algorithm; we simply use the straight-line Euclidean distance.

2.2 Path Finding (Edward Rivera)- A* and other

The second step of the PRM is to conduct path finding, which is divided into 2 steps: searching the graph and converting edges into motion plans.

To search the graph for an optimal path we used an SSP algorithm called A*. The recipe for A* is as follows:

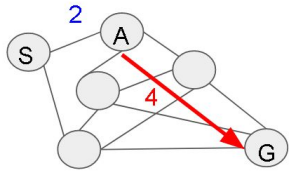
- Given a current node **c,** look at the neighbors of c
- For every neighbor **n**, if the distance to the current node + the distance along the edge(**c**, **n**) < the current known distance to neighbor n, **update** the queue
- Pop off the node with the **lowest** value on the queue

There are a few important points in the steps outlined above. Firstly, the queue. The queue is ordered according to a function f (n) = d(n) + h(n), where d(n) is the distance to the node and h(n) is the heuristic value of n. The update only occurs for some node n if the **distance along the edges** to that node n at that moment in the algorithm is less than the **current known**

distance along the edges to that node n. For example, in Figure 1, we look at node A for the first time from the perspective of S. Since we did not look at node A before, it has a theoretical distance of infinity and the distance along edge(S, A) = 2 and is less than infinity. Then once an update is initiated, the value for node n in the queue is updated to the new distance + its **heuristic** value. A heuristic is any consistent metric for nodes in a graph. The update ends with updating the parent of the node n (the node that precedes the updated node aka the current node at that time step) and reordering the queue. In Figure 1, the heuristic we use and in our implementation is the straight line distance from a node n to the goal point. When we update at A after looking at A from S, we update A's queue value to be the new edge distance 2 + 4 = 6. Subsequently, we update A's parent to be S. Using the parent data, we construct the optimal path from a starting point S to the goal.

*After looking at A from S

Node	Value	Parent
S	0	S
A	6	S



This is the heuristic

Figure 1

After constructing a path, the final step of the PRM is to convert edges into motion plans. However, this is not present in our implementation for a few reasons. Firstly, given the time constraints we found it challenging to implement (we were considering Dubin Curves). Secondly, we put a lot of faith in our sampler by sampling numerous points and using circle space techniques to hopefully give the car enough room along an edge to correct itself. This correction and other control techniques are outlined in the next section.

3. Pure Pursuit (Kuan Wei and Garza)

Pure pursuit is a path tracking algorithm that involves finding a point on the trajectory that is a fixed distance away from the robot and moving towards that point. Essentially, the robot computes a speed and angle using its current position to reach some lookahead point. After moving towards the lookahead point, the robot then recomputes the speed and angle to reach another lookahead point. This process repeats until the final destination goal is reached. In this lab, we broke the pure pursuit process down into two main components:

1. Finding a point on the trajectory closest to the car
2. Calculating the lookahead point

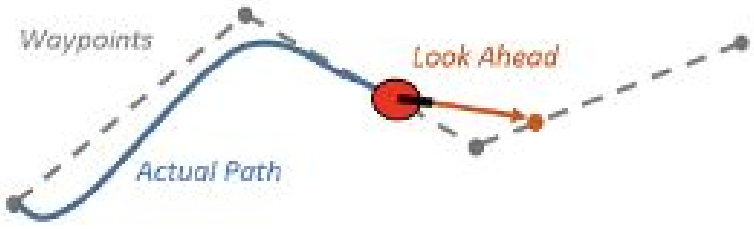


Figure 1 Visualization of pure pursuit

3.1 Finding Closest Point on Trajectory

We did this by iterating through all the points on the trajectory and using the distance formula to calculate the point closest to the car and returning its index. The code in Figure 2 illustrates our approach mentioned above in finding the index of closest point on trajectory.

```
points = self.trajectory.np_points
x_pos, y_pos = self.position
x = points[:,0] - x_pos
y = points[:,1] - y_pos
min_index = np.array((x**2 + y**2)**(0.5)).argmin()
return min_index
```

Figure 2 Finding closest point

3.2 Calculating Lookahead Point

3.2.1 Finding Intersection

Once we have the closest trajectory segment, we have to find the lookahead point. We did this by finding the intersection between the circle around the car, where the radius is the lookahead distance (which we set as 1 meter) and the circle's center is the car's position. We also have to be careful of edge cases, as there could be two intersections with the circle and we would need to take the bigger solution to move the robot forward.

3.2.2 Finding the Lookahead Point (Aaron Garza)

The next part of the pure pursuit algorithm involves searching for the lookahead point. In order for the car to follow the desired trajectory we need to search for the point that the car is essentially going to pursue. The search for the lookahead point begins at the line segment that contains the nearest point on the trajectory to the car. In Figure 3, this is the point corresponding to i = 1. Therefore, the closest line segment on the trajectory would be the line connecting points i = 1 and i = 2. The actual location of the lookahead point is then found by finding the intersection between the circle centered at the car's location defined by a radius of the lookahead distance and the line segment on the path as described in section 2.2.1. If a valid lookahead point is not found on the first line segment then our code continues along the path segments until one is found. With reference to the figure, this means that the search would continue to the path segment connecting points i = 2 and i = 3 and then continue down the trajectory

if a point is not found. Starting the search for the lookahead point at the closest path segment removes unnecessary searching through the rest of the path that the car has already traversed. This helps ensure that we are looking for a lookahead point in front of the car rather than behind it, and that a lookahead point is found quickly after only checking a few segments.

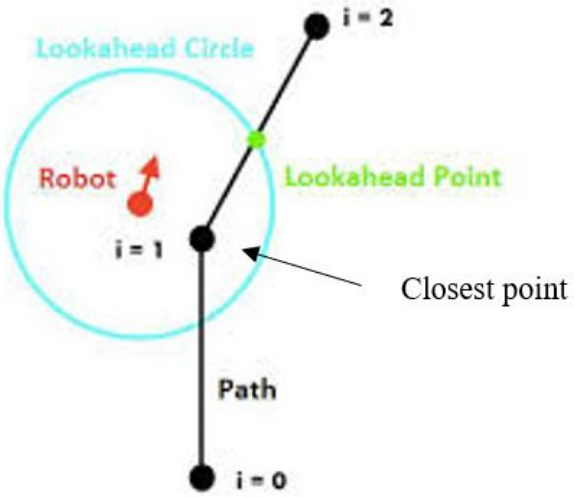


Figure 3: Visualization of the lookahead point search

3.3 Calculating the Steering Angle (Aaron Garza)

Using the lookahead point, we can then calculate the desired steering angle to drive the car along the trajectory. First we have to find the radius R in Figure 4 below that defines the arc between the lookahead point and the current vehicle position. The radius is defined by the following equation where R is the radius, D is the lookahead distance and x_L represents the horizontal offset distance of the lookahead point from the car.

$$R = \frac{D^2}{2x_L}$$

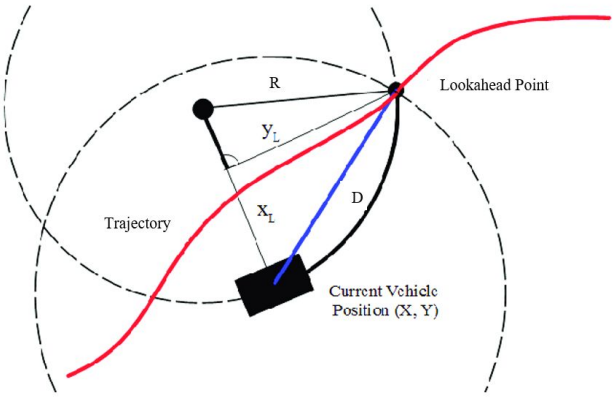


Figure 4: Illustration showing the relationship between the current location of the car and the lookahead point

Then, using the bicycle model shown in Figure 5 below to represent the car, the steering angle, δ, can be found using the following equation where L is the wheelbase length of the car and R is the radius calculated from the previous equation.

$$\delta = arctan(\frac{L}{R})$$

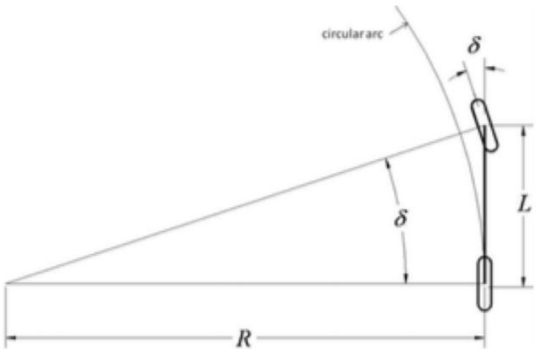


Figure 5: Illustration showing the geometry of the steering angle, δ

This steering angle is the angle required for the car to follow the arc joining the current location of the car and the lookahead point, and therefore follow the desired trajectory. Once the desired steering angle has been found this is published to the drive topic to drive the car toward the path.

4. Conclusion (Aaron Makikalli)

Our code for this lab consists of two main parts: path planning and pure pursuit. Path planning plans trajectories from the car to the goal pose, avoiding obstacles along the way. Pure pursuit takes the trajectory produced by the path planning section and follows it as closely as possible towards the goal pose. By implementing these two parts, we bring our car to the goal pose safely and efficiently.

Our code works in nearly all cases at this point, but sometimes the code experiences difficulties when following edges very close to the wall. Since the path planning algorithm is designed to find the most efficient path, it sometimes attempts to cut corners too closely, leading to a collision. Looking ahead, we can improve our path planning algorithm by applying a differential around the edges so that the car has some additional room to maneuver without colliding with the wall.

This lab is relevant to our final project because our car will require the ability to plan safe and efficient trajectories in real time in order to navigate the obstacle course. Once we are confident that our path planning algorithm does not cut corners too closely, we hope that the code we developed in this lab will be directly applicable to the final project. The implementation of efficient path planning and execution algorithms will be instrumental to completing the final challenge and bringing home the first place prize.

Lessons Learned

Technical

- **Aaron Makikalli -** In 16.410 last semester, I learned about some basic search algorithms like A* and path planning algorithms like RRT. In this lab, I enjoyed applying some of the knowledge I learned last semester while implementing a PRM, a new path planning algorithm for me.
- **Kuan Wei Huang** - Start the lab earlier and set higher expected work time.
- **Aaron Garza** - When an a desired behavior isn't observed sometimes it is often caused by a simple error rather than a large one in the general framework
- **Edward Rivera -** Especially for ROS programs, it is important to conduct integration tests rather than focus solely on unit testing specific modules
- **Kevin Sun** - Better time management.

Communication

- **Aaron Makikalli -** Now that we are working virtually, I've learned that it's important that we all check in with each other more frequently, even if we're not immediately working on something. Maintaining open lines of communication helps us take care of each other while making sure that all tasks are completed without anyone's tasks accidentally overlapping excessively.
- **Kuan Wei Huang** - Make sure everyone's on the same page to avoid confusion and increase productivity.
- **Aaron Garza** - Keep in contact with one another to make sure that we are each doing a part that we're supposed to do. This was lacking somewhat during the process and is something that set us behind
- **Edward Rivera -** As the team leader, it is essential that I connect teammates who may be struggling to the team and to connect the team to that member. It is always about the team.
- **Kevin Sun** - Increase communication to be more effective