Lab 6: Motion Planning and Trajectory Following

Team 20

April 25, 2019

1 Overview and Motivations

Author: Christopher Hughes

Lab 6 was designed to build upon the work done in the previous localization lab. That is, once a location is obtained for the car in a known two dimensional environment, navigating around said environment becomes possible. In order to achieve robust navigation, a two phase process is required. Firstly, a path that can get the car from start to finish as efficiently as possible without any collisions must be generated. Once this path is planned, a tracking algorithm must issue the appropriate drive commands so that the car can properly drive along the planned path.

The aforementioned problem has been subdivided into three main components: the search-based path planning algorithm, the sample-based path planning algorithm, and the pure pursuit path following algorithm. Each component is capable of being run and tested individually. In order to plan and execute a path in a two phase process, one of the two path planning algorithms is chosen and feeds results directly into the pure pursuit path tracking algorithm, which executes drive commands.

In the first component - the search-based path planning algorithm (Section 2.1) - the first objective was to compose an accurate graph of our given map so that only the hallways correspond to valid spaces where our robot can be (Section 2.1.1). The second objective was to experiment with different graph search methods and implement the algorithm that most efficiently finds a path from start to finish (Section 2.1.2).

Similarly to the search-based algorithm component the objective of the sample-based algorithm component (Section 2.2) is to find a path that can efficiently navigate the car from start to finish. This time however, only the map and its characteristics are used (there is no pre-computed search graph).

The final component - the pure pursuit algorithm (Section 2.3)- takes the paths planned by either the search-based or sample-based path planning algorithm and uses localization to properly drive the robot along these paths accurately and efficiently.

2 Proposed Approach

2.1 Search-Based Algorithms

Author: Christopher Hughes

There were many different types of search-based algorithms we could've used. Some prominent examples were Breadth-First Search, Depth-First Search, Dijkstra's, and A*. For our application, we wanted to select the algorithm that would find the destination the fastest and return the shortest possible path from start to end.

2.1.1 Graph Model

Before attempting to choose what algorithm we wanted to implement, we first needed to determine how we were going to model the map of our environment (primarily the Stata basement). The map was given as an array of probabilities that represent the likelihood of there being an obstacle in a location for every pixel in the image representation of the map. For simplicity, we first clipped all the probabilities for each pixel to be either zero or one (with one designating the presence of an obstacle), in order to make them true probabilities if they were not already. We then dilated these probabilities so that if a non-obstructed point was next to an 'obstacle,' they would also become an obstacle. This dilation was done with a kernel applied across the array. The bigger the kernel we applied, the more obstacles there would be. This equated to padding the map edges so that our robot could safely navigate between obstacles without colliding into them.

After padding the map, we than began translating the map by adding every pixel to a graph if it was below a threshold probability. This threshold probability was loosely set so that we could fine-tune the graph if we ended up getting too little or too many 'valid' nodes in the graph. This threshold was the 'cut-off' on whether a pixel was an obstacle or not. After adding all the nodes to the graph, we than needed to add the edges connecting the nodes. For these edges, we used the same constraints as the nodes: if a pixel directly adjacent to it was not an obstacle, or below the threshold, we set that pixel as a valid neighbor.

After we did the process of filtering valid nodes and edges, we had a graph that we stored in an adjacency dictionary where every valid node (obstacle-free pixel coordinate (i, j)) is mapped to each adjacent obstacle-free pixel in the set $[(i-1, j), (i+1, j), (i, j-1), (i, j+1)]$. This provided us a tunable graph for us to apply a search algorithm to.

2.1.2 Search Algorithm

Our search algorithm needed to be able to find the shortest path between our start and end nodes in a quick and efficient manner. As discussed earlier, we had many different algorithms to consider. From personal experience, we knew that Depth-First-Search (DFS) was not going to be a good choice because

it greedily goes in a random direction and 'hopes' that it finds the solution in that direction. Thus, although the algorithm might sometimes get lucky and find the end quickly, it's more likely that it explores an area of the graph that is too far from the optimal path. The next algorithm to consider was a Breadth-First-Search (BFS) or Dijkstra's algorithm. Given that our graph had 'equally' weighted edges, so that the line segments connecting nodes along a path are equidistant, both of these algorithms would function exactly the same and spread out in a radial pattern from the start node until it finds the end node. These two algorithms would work well, but would take a long time because they'd also explore extraneous paths of the graph equally to relevant ones.

We therefore arrived at the A* algorithm, which reduces the exploration of irrelevant parts of the graph in favor of parts that are closer or more likely to lead to the goal. A* handles this by implementing a heuristic weighting scheme to the graph. This heuristic weighting scheme is simply a combination of the Euclidean distance from the end node to the current node and the Euclidean distance from the start node to the current node. This meant that every node on the optimal path would be weighed exactly the same and that most deviations from that path would increase the weight of the node. This allowed us to very quickly and accurately explore the graph while ignoring directions that increase the combined distances. With this small heuristic advantage, A* returned the optimal solution for that given graph. This lead to a relatively smooth and efficient path-finding algorithm that allowed us to explore the map relatively quickly and as accurately as we could.

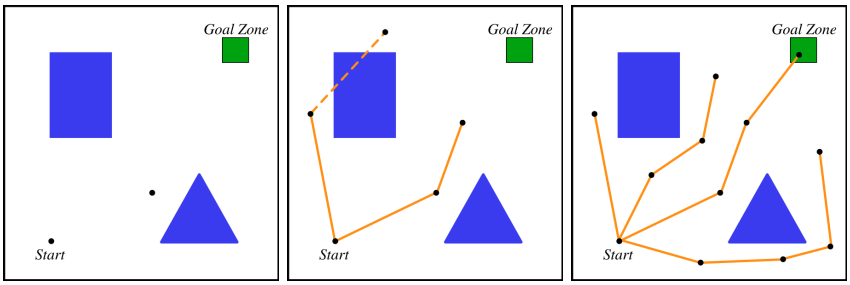
2.2 Sample-Based Algorithms

Author: Rowan Cheung

While search-based algorithms perform an exhaustive search for a guaranteed optimal path, sample-based algorithms randomly explore the search space. Although this may result in a slightly longer path, it can be computed much faster and does not require such extensive processing of the map. The sampling-based algorithm we chose to use is a rapidly-exploring random tree (RRT).

Before we proceeded with the algorithm, however, we performed a dilation on the map's boundaries by 10 pixels. This is to ensure that the found path does not cut any corners that are beyond the physical capabilities of our robot.

Our algorithm proceeded with a given start and end location and maintained a growing list of nodes. At every step of the algorithm, we randomly sampled a new point from the map and found the nearest point that already belonged in the map. We then steered this new point to the nearest node in the graph within an exploration distance of 50 centimeters. This was to allow us to perform tighter maneuvers if we needed to navigate obstacles. We then needed to check for collisions along the line from the nearest point to the new point - see Fig. 1b. If no collisions were found, we added this point to our list of nodes and saved the nearest node as its parent. This was repeated until we found the finish location. Finally, our path was created by repeatedly tracing from the



(a) Adding a possible next branching point.

(b) A situation in which the next branching point produces a collision.

(c) Full path to the goal reached within a margin of error.

Figure 1: Diagrams illustrating the key steps of the RRT sampling based planning algorithm.

end node to its parent node until we were back at the start.

In order to plan a full loop around the Stata basement, we first initialized a new boundary in between the start and end points, and then continued with our algorithm as usual. This forced the RRT to explore the entire loop.

2.3 Pure Pursuit Algorithm

Author: Eric Chen

After one of the path planners returned a sequence of points defining a collision-free path in a known two dimensional map, the robot's job was to drive along that path with as little deviation as possible. In order to accomplish this, a pure pursuit algorithm was implemented.

Pure pursuit is a popular path tracking algorithm that is derived from the natural way in which a human would maintain a trajectory while driving a car. The basic idea behind the algorithm is the look-ahead point. For a human driving a car around a bend, the look-ahead point could be any point along the road and in front of the car. In order to follow the curved path of the road, the human steers appropriately towards this chosen look-ahead point and then recomputes another look-ahead point a short time thereafter. Doing this recursively results in accurate path tracking that is able to robustly follow a large variety of paths.

The three main steps behind the pure pursuit algorithm are as follows. First, the point on the path closest to the robot's instantaneous location is determined. Second, the look-ahead point ahead of the closest point on the path is determined using a specified look-ahead distance. Third, the steering angle required to drive from the robot's current location to the look-ahead point on the path is calculated and the appropriate drive commands are published to the robot.

2.3.1 Determining the Nearest Point

In order to determine the point along the path nearest the location of the car, the shortest distance between the car and each line segment composing the full path was calculated. The distance between the robot's vectored location $v = (x, y)$ and any single line segment defined by the start vector $v_0 = (x_0, y_0)$ and the end vector $v_1 = (x_1, y_1)$ can be calculated by finding the projection of v onto an infinitely extended version of the line segment $s = v_0 + t(v_1 - v_0)$ as follows:

$$t = \frac{dx(x - x_0) + dy(y - y_0)}{(dx^2 + dy^2)} \tag{1}$$

This projection was used to calculate the point on the line segment closest to the robot's location v. If $t <= 0$, the closest point was the start point v_0. If $t >= 1$ the closest point was the end point v_1. If $0 < t < 1$, the closest point $v_c = (x_c, y_c)$ could be extracted as $v_c = (x_0 + tdx, y_0 + tdy)$. The distance between v and v_c was the distance between two points, $d = \sqrt{(x - x_c)^2 + (y - y_c)^2}$. Computing d for every line segment and taking the minimum distance yielded the line segment closest to the car, and the point on that particular line segment v_c closest to the car.

2.3.2 Determining the Look-ahead Point

Having obtained the line segment in the path closest to the robot, a look-ahead point was then calculated by using a specified look-ahead distance l. The concept of a variable look-ahead distance is discussed in Section 4.3, but for our current implementation the look-ahead distance was set to $l = 0.5$ meters. The look-ahead point was determined by finding the intersection of a circle with radius l, centered at the robot's location, and the nearest line segment that crosses this circle ahead of the closest point as shown in Fig 2.

In order to make calculations simpler, the global coordinate frame was translated so that the center of the circle (i.e. the robot's position v) was now the origin. The line segment defined by vectors $v_0 = (x_0, y_0)$ and $v_1 = (x_1, y_1)$ could now be extended to an infinite line with equation $y = mx + y_{int}$, with slope $m = \frac{y_1 - y_0}{x_1 - x_0}$ and y-intercept $y_{int} = y_0 - mx_0$. The x coordinate of the intersection between this infinitely extended line and a circle with radius l centered at the origin was found by solving the following equation for x:

$$x^2 + (mx + y_{int})^2 = l^2 \tag{2}$$

The above equation is simply that of a circle centered at the origin, $x^2 + y^2 = l^2$, with the linear equation $y = mx + y_{int}$ substituted for the y value. Conceptually, this equation described points along the line segment that solved both equations (the line and the circle). Hence, these points lied simultaneously along the circle centered at $v = (0, 0)$ with radius l and along the infinitely extended line

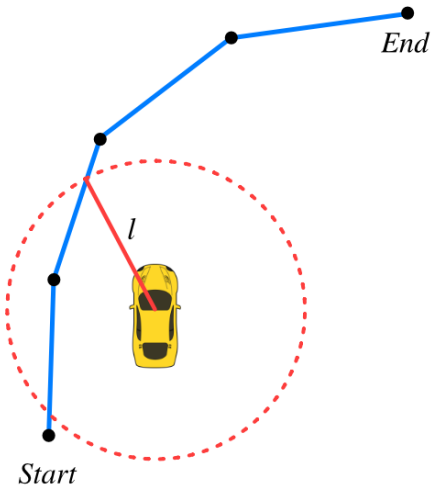


Figure 2: look-ahead point determined by intersecting a circle defined by look-ahead distance l with the nearest line segments in the path.

segment, and represented the intersection of the two. Consequently, this was a quadratic equation that would be solved for an x-coordinate as such:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3}$$

where $a = m^2 + 1$, $b = 2my_{int}$, and $c = y_{int}^2 - l^2$. There were three potential cases here. Solving for x would produce two imaginary roots, in which case there was no real intersection and no look-ahead point would be returned. If x was a single real root that lied between x_0 and x_1, then the circle intersected the line segment tangentially and this x and its corresponding $y = mx + y_{int}$ were treated as the look-ahead point. If x was outside the bounds of x_0 and x_1, then the circle intersected the infinitely extended line, but this intersection lied outside of the finite line segment of interest. Similarly, if solving for x produced two real roots, then the circle intersected the infinitely extended line twice and produced a chord with an associated arc. Then, both roots were tested to see if they lied within the bounds of the finite line segment (just as described in the tangential intersection case). If only one point lied within the segment, this point was returned as the look-ahead point. If both points lied within the finite line segment, then the point with x coordinate farthest away from x_0 (farthest ahead in the path) was chosen as the look-ahead point.

2.3.3 Determining the Steering Angle

Once a look-ahead point was determined, the appropriate steering angle must be calculated in order to drive the car towards said point. Obtaining the steering angle was a geometric calculation that involved extracting the curvature of the circular arc connecting the robot's location v and the look-ahead point p. This curvature was calculated using the look-ahead distance l and the difference between the robot's heading and the look-ahead heading (the heading of the vector that is drawn from the car v to the look-ahead point p). The curvature of this arc was then derived using the law of sines, given by the following equation:

$$k = \frac{2 \sin \theta}{l} \tag{4}$$

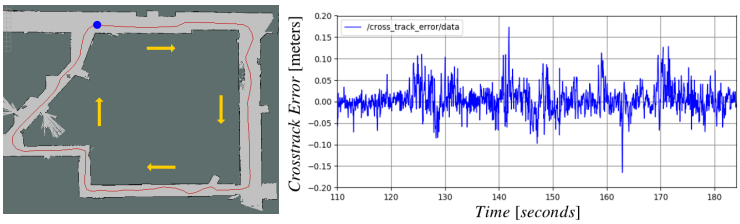
where θ is the difference between the robot's heading and the look-ahead heading. This curvature was then multiplied by the robot's wheelbase L (the distance from the front axle to the rear axle); then, the angle A (in radians) at which to turn the wheels was calculated via the inverse tangent: $A = \tan^{-1}(kL)$. This steering angle was published to the car as a drive command, and the car drove towards the look-ahead point.

2.4 Cross-track Error

Author: Carlos Cuevas

By repeating the three steps outlined above, the car effectively followed a variety of suitable trajectories and was tested successfully at a speed of $0.7 \frac{m}{sec}$ on a variety of reasonably curved paths (including various portions of the Stata Basement loop).

In order to understand how accurate the pure pursuit algorithm was in pursuing the projected path, we created a function that calculated the distance between the car and the closest point on the path, the cross-track error. This was a simple metric of how far the car was deviating from its intended location on the path. This was done by checking the line segment containing the look-ahead point and the two previous line segments against the robot's position. The cross product between the vector formed by the two points of each line segment and the vector formed by the rear point of the line segment and the position of the car was used to calculate the cross-track error. This value was then plotted over time (see Fig. 3), which gave a visual representation of the pure pursuit algorithm's accurate performance. The maximum absolute cross-track error on our graph was less than 0.2 meters, which demonstrated an acceptable performance for this particular implementation of pure pursuit.



(a) A loop around the Stata Basement map. The blue circle denotes both the start and end points, and the yellow arrows show the direction of the path.

(b) Cross-track error associated with the execution of the loop on the left at a constant speed of $0.7\frac{m}{sec}$. Notice the sharp peaks denoting a quick correction rate, and the decently small error overall.

Figure 3: The cross-track error for a specific path scenario.

3 Experimental Evaluation

3.1 Search-Based Algorithms

Author: Christopher Hughes

Our implementation of A* far exceeded our expectations for its effectiveness. The resulting graph of the Stata Basement loop that we did a majority of our tests on was generated in an average of 48 seconds on a laptop, and 143 seconds on the robot's computer. However, since we were able to recompute the graph and use it for future paths, this 'boot' cost associated with the algorithm was acceptable and easy to plan around.

When we ran A* repetitively on the sets of points about ten meters away from each other, using different points, such as straight-aways and curves, we would get an average time of 0.2 seconds for mostly straight paths, and 1.7 seconds for paths that went around significant bends. However, as the distance from one corner of the Stata basement loop to the other was increased, we found a path in 28-46 seconds, depending on the parameters we optimized for in the creation of the graph and heuristic algorithms. This turned out to be a relatively decent average of 35.4 seconds for a decent obstacle-clear path that did not intersect the walls or any object. These measurements were taken on a computer in simulation and when compared with the actual times on the car, in the worse case, the latter was three times slower.

There was a clear correlation between the number of way-points in our algorithm and the time it took to generate the whole path. The straighter and lower number of segments you divided the algorithm into, the faster it would be, thus allowing us to map the whole Stata basement after a few tries and closer tuning of the graph's parameters.

4 Lessons Learned

Author: Carlos Cuevas

In retrospect, our team did well overall with our path planning implementation. Both path planning algorithms were able to generate a plan between a variety of start and end points. Additionally, each algorithm was able to generate a path for the entire Stata Basement loop with some way-points and parameter tuning. The pure pursuit algorithm demonstrated excellent performance with a variety of hand drawn paths in simulation, including an entire loop of the Stata Basement map. In simulation, the pure pursuit algorithm was able to follow a variety of paths generated directly by both RRT and A*. On the real car, the RRT algorithm struggled to generate a path in the checkoff section of the Stata Basement where a pillar was located. Due to real world circumstances, the exploration distance between consecutive points was lengthened and some adjustments were made. However, our team was not able to plan around the pillar with RRT fast enough to complete the checkoff. Besides this, RRT planning worked successfully on the real car in the 2 other corners of the Stata Basement map where we were able to find adequate unoccupied space to test ahead of checkoff time. Considering this, we are thinking of using A* to plan our final path for the challenge. Although it is slower, it computes a smoother more optimal path that is easier to follow and is able to navigate successfully around the pillar that proved problematic for our current implementation of RRT. Aside from this, both algorithms demonstrated decent performance on the real car as described above. The pure pursuit algorithm also performed well on the real car, yielding an acceptably low cross-track error around both corners of the State Basement map where testing occurred. Video evidence of planning and execution on the real robot was recorded. In addition, a few improvements could be made with each component in order to achieve better performance in the final challenge. These are discussed for each individual component below.

4.1 Search-Based Algorithm

Author: Christopher Hughes

Our graph creation algorithm would sometimes add nodes that appear to be obstacles so it would sometimes plan an algorithm that goes through the "unexplored" or null space of the map and return to the obstacle free zone via a 'hole' or similar unexplored region thus generating a path through the wall. We believe to fully eliminate this problem we need to adjust the dilation we do. We tried different sized summation filters that just made the walls strictly bigger, but we would still get holes in the wall or if the filter was too big, we would loose pathways and cause the paths to randomly curve in 'free'-space. To eliminate we suspect we could use a smaller summation filter combined with a variable sized averaging filter and even possibly more complex filters like hyperbolic tangent that would weight 0's and 1's more heavily than values in between causing the average to diverge to being certainly obstacles or certainly not. This further experimentation would definitely help us eliminate holes in the graph and allow

us to path plan better.

4.2 Sample-Based Algorithm

Author: Rowan Cheung

Originally the exploration distance used by RRT was much smaller than the look-ahead distance implemented in our Pure Pursuit. This led to many difficulties integrating RRT with pure pursuit, as the pure pursuit had trouble finding the trajectories produced by RRT. To solve this problem, we increased the exploration distance of the RRT which also decreased our ability to navigate succesfully around obstacles, like the pillar in Stata basement. Further tuning of the exploration distance and amount of map dilation is needed for RRT to be more robust to all map locations and easily integrated with pure pursuit.

In the future - especially for our final race - we would like to experiment with adapting our RRT algorithm to RRT*. While we did not have time to add the rewiring step of RRT* during this lab, this could help us improve our planning efficiently for the final project. As the Stata Basement map is very narrow with little open space, we had some trouble getting RRT to explore quickly because it could get stuck not finding valid points, or remain clustered in the same place. Ideally, the addition of a cost metric to adding points could help us spread more efficiently throughout the map.

4.3 Pure Pursuit Algorithm

Author: Eric Chen

Our original Pure Pursuit algorithm did not determine the nearest point on the path. Instead, a running index was kept as the robot progressed along the path, and the look-ahead point was determined by intersecting the look-ahead distance directly with line segments in the vicinity of the current index in the path. This worked nicely and the car was able to accurately track a given path, however one main issue arose - the car was only able to track the path if located near the segment of the path at the running index. This meant that if the car lost the path or was started at a point in the middle of the path while the index was at the start or the end, it would not able to track the path. By calculating the nearest point on the path first and then using this information to subsequently inform the look-ahead point and steering angle calculations, the car is now more robust to losing the path and is able to handle the "kidnapped robot" scenario in which it is placed at a random location anywhere along the path.

In terms of future improvements, implementing a variable look-ahead distance that changes based on the curvature of the path could greatly increase the accuracy of the Pure Pursuit path tracking algorithm along a wider variety of paths. When a path has a steeper curvature, a shorter look-ahead distance allows the vehicle to make smaller more incremental changes to its steering instead of attempting to navigate a large sharp angle all at once. This results in more accurate behavior along sharper bends. In the opposite scenario where

the path remains fairly straight and smooth, a larger look-ahead distance could benefit the computational efficiency if the look-ahead point were calculated less frequently. Implementing a variable look-ahead distance could be done by determining the curvature of the path and appropriately tuning a controller.

Lastly, the capacity to adjust the velocity of the car based on the curvature of the path could greatly increase accuracy and the speed at which the car completes its trajectory. By increasing velocity along relatively straight portions of the path and decreasing the speed along sharper turns, the car can maintain high path tracking accuracy while achieving a faster execution time for the entire path. This could be implemented using the same general methodology as the variable look-ahead distance - by computing path curvature and then implementing a controller to adjust the velocity of the car.

Editor: Carlos Cuevas