Team 5

Navigating the Stata Basement: *Path Finding and Trajectory Following Algorithms*

04/24/2019

Hector Castillo
Mitchell Guillaume
Claire Traweek
Daniel Wiest
Franklin Zhang

1. Overview and Motivations *(Authored by Franklin Zhang)*

At a low level, robots need very precise commands to navigate their environment. However, inputting such commands is tedious and can be challenging, or in the case of very complex systems, nearly impossible. Automating these commands via a decision making algorithm becomes desirable because decision making saves time for the roboticist, enables non-experts to control the robot, and allows for complex and high speed maneuvers not possible by direct control.

Robots move around, but who decides where and how they move? Manually inputting commands to guide robots through simple tasks is tedious and time consuming. Automating these tasks becomes desirable . As we move higher up in the robot architecture from low level control towards higher level decision making processes, we allow the robot to start making more and more of its own decisions. Robots can then decide for themselves what to do when they reach a red light, or which directions to turn when they are trying to get out of a maze.

For robot cars, one of these decisions is to decide how to get to a destination quickly and without collisions. This problem is called the path planning problem. This week we explored a couple of these path planning algorithms. We implemented a search based and sample based planning algorithm, evaluated the performances of both and integrated these algorithms with the rest of the robot.

1.1. Lab Overview *(Authored by Franklin Zhang)*

This week we built off what we did in our previous labs. We used our particle filter localization algorithm and improved upon our steering and speed controllers with a pure pursuit controller. The particle filter will give us a state estimate of the robot and that will be used to plan and drive a trajectory that we will create using path planning algorithms. We divided up the lab into three distinct modules: search based path planning, sample based path planning, and pure pursuit.

The first path planning algorithm is a search based algorithm called A*. We created a graph that contains nodes, which represent possible states of the robot. After the graph is created, A* searches the graph, beginning from a start point based on the state estimate from the localization algorithm to find the best graph traversal that will give the best route to a goal. A route is considered good if the distance required to traverse through all the states in minimized.

The second algorithm we explored was a Rapidly Exploring Random Tree or (RRT*). Instead of searching through a calculated set of states like A* does, RRT* randomly generates states and searches over those. RRT* also considers the kinematic constraints of the physical car when generating trajectories.

Another part of this lab is the pure pursuit controller. The pure pursuit controller uses geometry to directly figure out what steering angle and velocity is required for the car to get to specific point.

The performance of these modules were evaluated. The path planning algorithms were then integrated with the pure pursuit and tested in simulation and in real life.

1.2. Motivations *(Authored by Franklin Zhang)*

Path planning is important because the robot needs to make high level decisions about where to go and how to get somewhere autonomously. The planned paths found using these algorithms are shorter and more kinematically feasible than paths that a human can draw.

Efficiency and autonomy are important especially when thinking about the challenges that our team will have to complete in the following labs. Programming a robot that is able to escape a maze on its own or navigate through an obstacle course really fast requires a robust path planning algorithm.

1.3. Goals *(Authored by Franklin Zhang)*

The goal for this week was to implement path planning algorithms that find efficient trajectories that allow our robot to drive from its current position to a manually set final position in the real world.

1.3.1. Implement High Level Path Planning Algorithms

Specifically we aimed to implement the A* and RRT* path planning algorithms. The algorithms needed to be be efficient to enable a fast computation of a path, which should be optimized to be the fastest path towards a goal. The algorithms should account for obstacles and optionally, the kinematics of the car.

1.3.2. Design a Pure Pursuit Controller

Implement a pure pursuit controller that is able to calculate exact values for a steering angle and velocity that will navigate the robot to a desired goal state without too much overshoot or error and without getting lost or colliding with obstacles.

2. Proposed Approach

We first implemented an RRT* and an A* path planner for our sampling and searched based algorithms, respectively. After optimizing and evaluating both, we compared our results and chose the best algorithm to put on the robot. We controlled the

robot with a pure pursuit controller, which traverses planned paths by moving from point to point.

For both of these algorithms, we found paths in the Stata basement. Because the paths our algorithms found had no width dimension, but the car traversing them did, we ensured that the algorithms did not return paths that would cause the car to cut corners or side-swipe walls by dilating the map to a little over half of the width of the car. This value was adjusted with testing. We settled on an erosion radius of .3 meters. The dilated map was saved in our repository and used as an input for both of our algorithms.

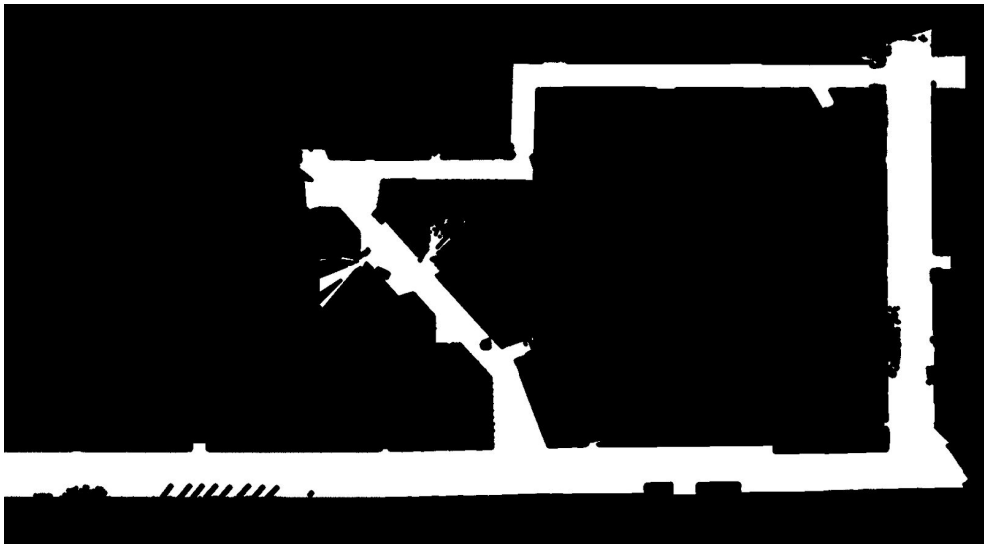


Figure 1: The map provided by the course staff of the Stata basement is offset by 0.3 meters (approximately the width of the vehicle) around all obstacles in order to allow planning algorithms to treat the robot as a point object. Performing this dilation greatly simplifies the process of determining whether or not the robot will collide with the wall because only the robot's center of mass location must be checked for collision, allowing for the motion planning algorithms detailed in sections 2.1 and 2.2 to run operate more efficiently.

2.1. Search-Based Planning Algorithm *(Authored by Mitchell Guillaume and Franklin Zhang)*

For our search-based algorithm, we chose to implement A*. This algorithm works by continuously checking pixel neighbors in the direction of the goal until the goal is reached. A* is advantageous because it will always return a solution if the solution exists, but is disadvantageous because it is not guaranteed to return the shortest or an easily drivable solution.

2.1.1. Technical Approach

There are many different ways to implement a search algorithm. However, regardless of the algorithm used, the search space is often

represented in the same, mathematical construct known as a graph. A graph consists of a series of two objects--nodes and edges. A node represents a discrete element of the search space. Things like location, items, states, could all be examples of nodes. Edges represent how the nodes are interconnected. Edges link two nodes and can be bidirectional, unidirectional, and/or weighted. Figure 2 shows a visual representation of a simple graph with unweighted bidirectional nodes. Graphs are incredibly useful because they can be used to represent a wide variety of situations. For the implementation of our search based algorithm, we explored a few different ways that we could represent a map of the Stata Center basement as a graph.

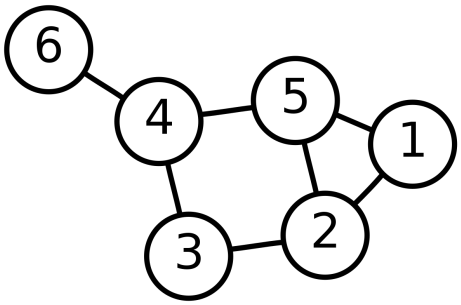


Figure 2: A visual representation of a simple graph. Nodes are represented by circles and edges are the lines that connect them. Points on the map can be represented as nodes, and potential connections between them as edges.

Algorithms like breadth first search, depth first search, and Dijkstra's algorithm, are all examples of ways to traverse or search a graph. The A* search is one of many in this class of algorithms. We chose to implement A* because it is a "smart" algorithm. Instead of advancing all possible paths at the same rate, it prioritizes the path that currently has the lowest cost. The idea is that by advancing only the most promising route, the goal can be found faster.

The cost of path is determined by a cost function that is specific to the application of the search algorithm. In our case, the cost function was the length of the path. The cost function for each node in the graph is the sum of two components. The first is the length of the path that was traveled to reach the node. This component is easily defined-- the algorithm knows how far it has traveled to reach each node. The second component is a little more tricky. The algorithm doesn't know exactly how long a path from a given node to the goal will be. To know the length of the path to the goal, the algorithm would have to know the actual path.However, because the exact path is unknown, the algorithm needs to make a "best guess". This is called the heuristic function, and is specific to the context in which the search is being implemented. Since we're trying to minimize path length, our heuristic function is simply the euclidean distance from the node to the goal.

After we've defined our cost function, it's possible to implement the A* search algorithm. In our graph representation of the Stata Center basement, each pixel is a node, and the edges connecting them are weighted by the euclidean distance between them. A starting point is defined as an endpoint is given as a command from the robot's user in Rviz. For computational simplicity, we defined each node's neighbors as the eight pixels surrounding it. Other options for determining node neighbors were explored and are discussed later in this section.

The algorithm maintains two sets of nodes. The first list is the "frontier" of the search and is also known as the open list. Nodes in this list represent the endpoints of all currently explored paths. As the algorithm works, this frontier pushes forward towards the goal. The second set of nodes is called the closed list. This consists of nodes that have been visited but are not on the frontier. It includes all nodes behind the frontier, including points that are in the trajectory.

The algorithm starts with just the starting point in the frontier, and no nodes in the closed list. The algorithm chooses the node with the lowest cost from the frontier (simply the starting point on the first iteration) and then finds all of that node's neighbors. Each neighbor is then added to the frontier, unless the node is already in the frontier and has a shorter path from the start, the node is in the closed list, or the node is obstructed by an obstacle. If none of those three conditions are true, then the node must be in clear , unexplored space and it is added to the frontier. After all of the neighboring nodes have been checked, the parent node is removed from the frontier and added to the closed list. The process repeats until the frontier reaches the goal. Once the goal is reached, the algorithm works back up the graph, following the nodes' parent pointers. This is our trajectory. Figure 3 below shows how the A* algorithm moves through a simple map.

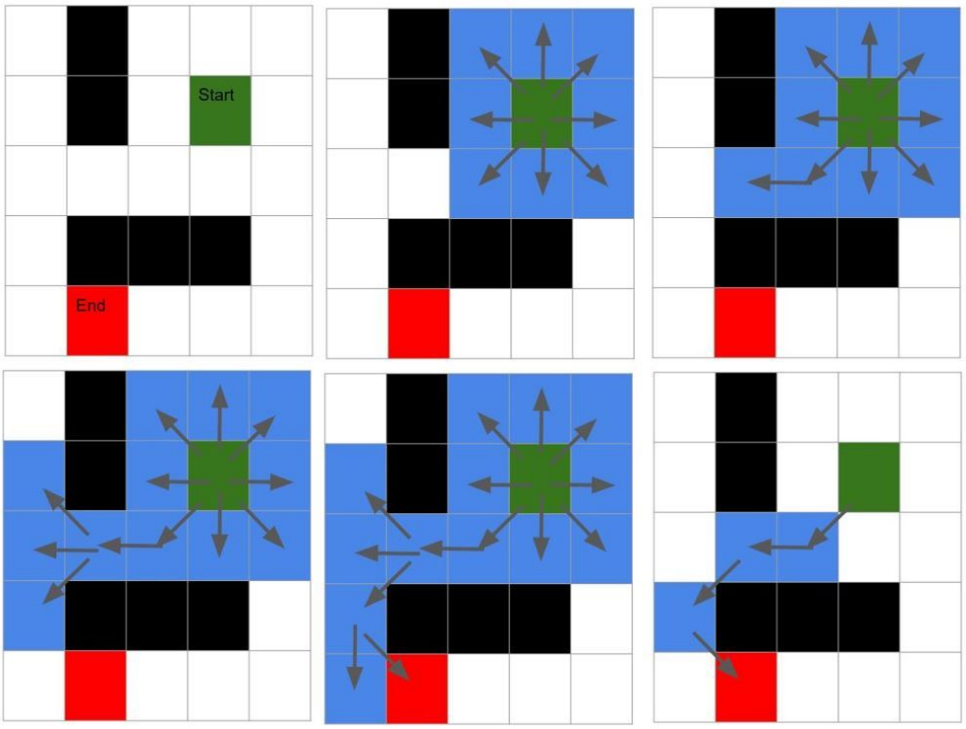


Figure 3: A visual representation of the A* algorithm on a simple map. From top left to bottom right, the algorithm pushes towards the goal. Each arrow points from a parent node to its neighbors that are valid.

The nodes of the A* algorithm are each of the individual pixels in the image. There are over 2 million pixels in the image and of those there are 250 thousand possible states to consider if we decide to use every single pixel. A* will run too slow on a graph of this size. In order to increase the speed, we can reduce the number of nodes that A* has to look through. We explored 2 ways of reducing nodes: corner point search with a gift-wrapping algorithm and down scaling to reduce the number of pixels in an image..

The gift-wrapping algorithm was implemented first. Imagine that the environment the robot is in is filled with small obstacles. We can gift-wrap an obstacle by drawing a convex polygon around the obstacle that contains all the the points of the obstacle. Corner point search with gift-wrapping assumes that the shortest path between any 2 points in this environment is a connected set of points that forms a continuous line segment, where each point is the starting point, end point, or one of the vertices of the convex polygons.

Walking through the algorithm for doing Corner Point Search, the first step is edge-detection. To reduce the size of the input, we can do edge-detection to find the pixels corresponding to the boundary of obstacles. Since we are running a gift-wrapping algorithm on it, having only the boundary points wouldn't affect the output that much (There are some imperfections in the edge detection

that have some effect on the output. However, this effect doesn't have a large negative effect on the rest of the algorithm).

Next, a recursive block finding algorithm is run on these points to cluster each pixel into groups, or blocks with each block being one distinct object / obstacle, shown in Figure 4. Then the gift-wrapping algorithm is run on each of these blocks. The gift-wrapping algorithm finds the points that form a bounding box, shown in right in Figure 4. The algorithm starts with the leftmost point and draws a line to the next point so that the rest of the points all fall to the right side of the line. In order to do this, we have to iterate through all of the points to figure out which one of them is "most left". We can compare if a point is on the left side or right of each each other by considering the sign of the cross product of the vectors formed by connecting two candidate points for the bounding polygon to the most recent point in the bounding polygon list. Lists of bounding polygon vertices are found for each block and all those points are returned as the set of nodes that will be considered in our graph.

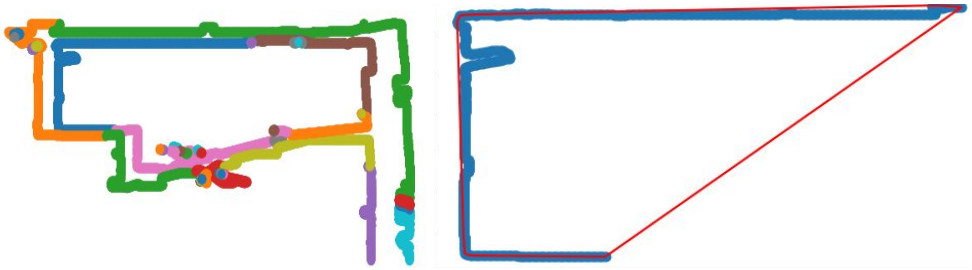


Figure 4: The original image was run through an edge detection algorithm to find the edges of obstacles, shown by the different colors on the left. On the right, each individual obstacle is run through a gift-wrapping algorithm that finds the set of points that draws a convex wrapper that contains all the points within the obstacle.

This process does a reducing the size of the graph significantly, shown in Figure 5. However, this process of reducing nodes to only consider the most relevant states only works in certain cases. There is a condition that the bounding polygons can't overlap. When these polygons overlap, we exclude some important states that are necessary to have in order to have all the nodes necessary to generate the most optimal path.

Figure 5: Above is a picture of the output of the gift-wrapping run on each identified obstacle. The blue dots represent the points that from the convex polygons in the gift-wrapping algorithm. This algorithm misses some key feature points such as the ones that should be on the edges of the a protruding piece of the wall located on the bottom right. Further improvements on this algorithm are necessary for it to function correctly. However, it can potentially speed up the A* algorithm because it can reduce the number of nodes to around 500.

Corner point search can be effective in some cases, but in this application there are too many weaknesses. There is another approach we can take that doesn't throw out all the necessary data, but is not as efficient as corner point search potentially is. We can down scale our image by a scale factor to decrease the number of searchable nodes, as shown in Figure 6. Multiplying the image by a scale factor of 1/10 reduces the number of nodes from around 250 thousand in the original image to around 66 thousand nodes. The nodes then represent potential robot states in the rescaled image. The found trajectory is then scaled up and the points are converted into the pixel space of the original image by multiplying by the same scale factor. The drawback of this approach is that it trades resolution for speed. When the image is rescaled, there is information lost in the process because a original pixel index is divided or multiplied by a scale factor and then rounded in order to be an integer. These floating point values are thrown away during rounding, even if they represent a significant distance in the real world, potentially causing collisions. Another tradeoff is that rescaling to extremely small values causes the hallway to appear blocked, meaning that trajectories that exist cannot be found.

Although both algorithms have their strengths and weaknesses, we decided to implement the down scaling method to reduce nodes because it is easier to integrate with the rest of the A* algorithm and corner point search doesn't reliably return the optimal solution.

Figure 6: The original image (left) contains over 2 million pixels. It can be downsized by scaling the width and height of the picture by 1/10 to create a smaller image that has 2 thousand pixels instead.

2.1.2. ROS Implementation

The ROS implementation of the A* search based algorithm is almost identical to the ROS implementation of the RRT* algorithm. See section 2.2.2.

2.2. Sampling-Based Planning Algorithm *(Authored by Daniel Wiest)*

The idea underlying sampling based approaches to motion planning is to reduce the computational cost associated with exhaustively searching the environment (in this case, the map of Stata basement seen in Figure 6) by relying on random samples of the environment instead. This may sound trivial, but the consequences of this shift in methodology is that we give up the assumption that the algorithm will find the *most ideal path* in all cases, and instead arrive at a *good enough* solution. Much of the difficulty associated with the creation of paths using sampling-based planning was in defining quantitatively what a "*good enough*" solution meant. Significant effort was put into defining a metric for success through the use of iterative parameter searches and an evaluation map, a topic covered in detail in section 3.2.

2.2.1. Technical Approach

The first step in implementing a sampling-based motion planning algorithm is to decide which algorithm was best suited for the task of navigating the Stata basement. Most significant among the factors that dictated how well suited the algorithms were was whether or not they could be adapted to take into account the motion-related constraints imposed by the racecar platform. Unlike other robots such as the roomba, which can rotate and move in any direction at any location, the racecar can only make turns corresponding to its current speed. Additionally, the racecar can only move directly forward with respect to a

reference frame located at its rear wheels (base_link frame) at any given instant in time and as such is unable to turn without also moving forward.

With the kinematic constraints of the racecar in mind, an approach based on rapidly exploring random trees (RRT) was adopted both because of the flexibility it afforded in the motion capabilities of the robot and also the incremental pathway towards functionality that it presented (RRT → RRT*). Because the Ackerman steering geometry of the racecar is best suited to follow arcs, our implementation of RRT* was designed to construct paths entirely out of circular sections and hermite splines with radiuses of curvature defined to be greater than the minimum turning radius of the racecar.

The process of constructing an arc connecting two points given the tangency constraint of the first point is as follows and is pictured in Figure 8. First, the points are translated to the origin and rotated such that the tangency constraint of the first point is aligned with the x-axis. Next, the y location of the radius of the circle is calculated with Equation 1. Because of the tangency constraint, it is known that R_x is zero. At this stage, it is ensured that the radius calculated meets the turning constraints of the racecar platform.

$$R_y = \frac{{x_{end}}^2 + {y_{end}}^2}{2y_{end}} \tag{1}$$

Next, the angle swept (theta) by the arc is trigonometrically formulated and a parametric equation for the circle is created and evaluated for each pixel along the arc. The result of these operations is a list of points which are then rotated and translated back to the global reference frame. Finally, the validity of the path is assessed by checking that each point in the trajectory found lies in an unoccupied region (A white pixel) of the dilated map shown in Figure 6. If the path does not cross any invalid pixels, then it is returned to the RRT* algorithm.

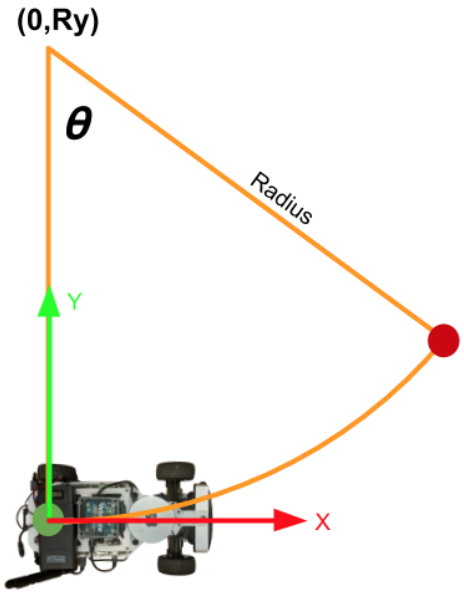


Figure 7: Visual interpretation of circle based trajectory calculation. Given the start point in green and end point in red, the circular section connecting them is found and discretized into points that are then checked to ensure that they fall in unoccupied regions of the map.

Cubic hermite splines are used in cases where both the start and end points of the requested paths have tangency constraints associated with them (see Figure 8). The rewiring phase of RRT* (covered in a subsequent section) requires that tangency constraints are preserved and accordingly, relies entirely on spline-based trajectories.

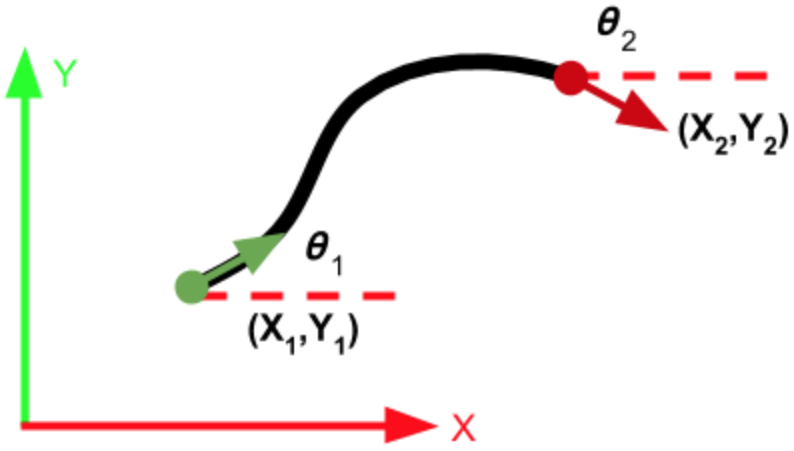


Figure 8: A cubic hermite spline is shown connecting the green starting point and red ending point. The important capability of the hermite spline is that it enables the path to match the tangency of both the starting and ending points.

Equation 2 is the mathematical representation of a parametric two dimensional cubic hermite spline. By solving the systems of equations presented for the starting conditions (X_1,Y_1,$\boldsymbol{\theta}_1$) at t = 0 and the ending conditions (X_2,Y_2,$\boldsymbol{\theta}_2$) at t = 1, the values of the spline's constants are calculated. The angle constraints prescribed by theta are satisfied by ensuring that the derivatives of the spline's the x and y components are equal to the cosine and sine of the starting and ending angles $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ respectively.

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y \end{aligned} \tag{2}$$

Next, iteration over the parametric equation of the spline checks that each point along its length lies in the unoccupied region of the map. If all points are valid, then the spline is passed to another function which uses the first and second derivatives of Equation 2 to determine the radius of curvature at all points along the path and ensure that they all exceed the minimum turning radius of the car. Finally, once the path is verified to comply with the map and motion constraints, the trajectory is passed back to the planning algorithm.

Once paths that comply with the kinematic constraints of the racecar between any two points can be calculated, the rest of the planning algorithm can be developed relatively simply. The RRT*-based planning algorithm implemented by our team begins by first choosing either a random point in the unoccupied regions of the map or the ending location of the desired trajectory with probabilities 0.875 and 0.125 respectively. The end point is chosen frequently enough so that if a path to the end location is available, it will be found quickly and iteration can be terminated. All unoccupied points are initialized into a Python set when the algorithm trajectory planner is created to allow for quick query of point validity.

$$Radius = \gamma \sqrt{\frac{\log N}{N}} \tag{3}$$

Next, a new point is calculated by finding the location that is a step size towards the randomly chosen point (see section 3.2 for a discussion of selecting the step size and γ). At this time, the path distance to connect the new node with all pre-existing nodes within a radius defined by Equation 3 is found and the neighboring node that can be connected to the new node using a valid circular

section and also minimizes the distance of the path from the starting point to the new point is chosen as the new point's parent. This process is depicted visually in Figure 9.

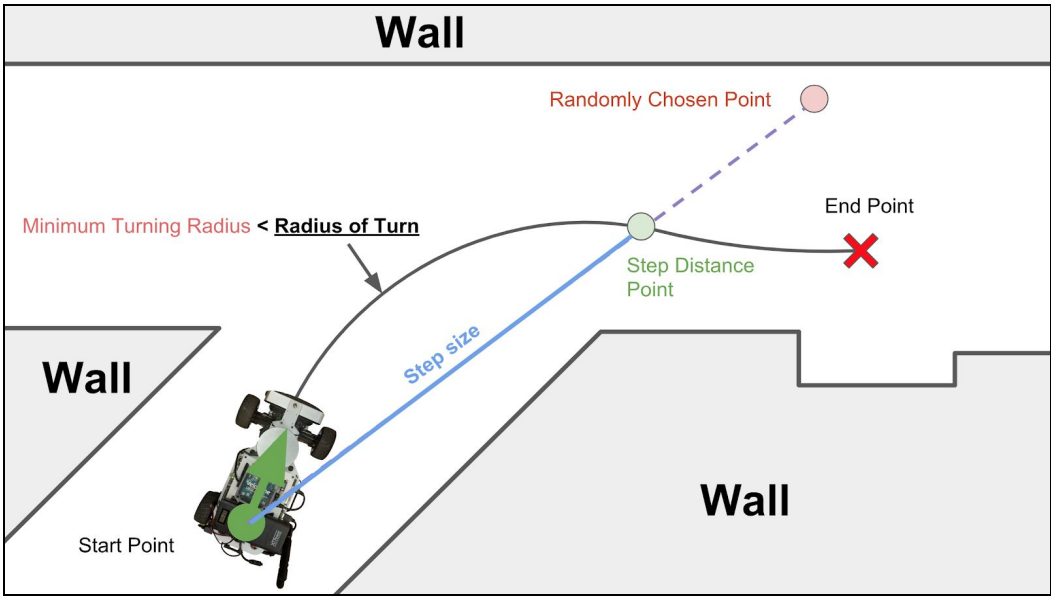
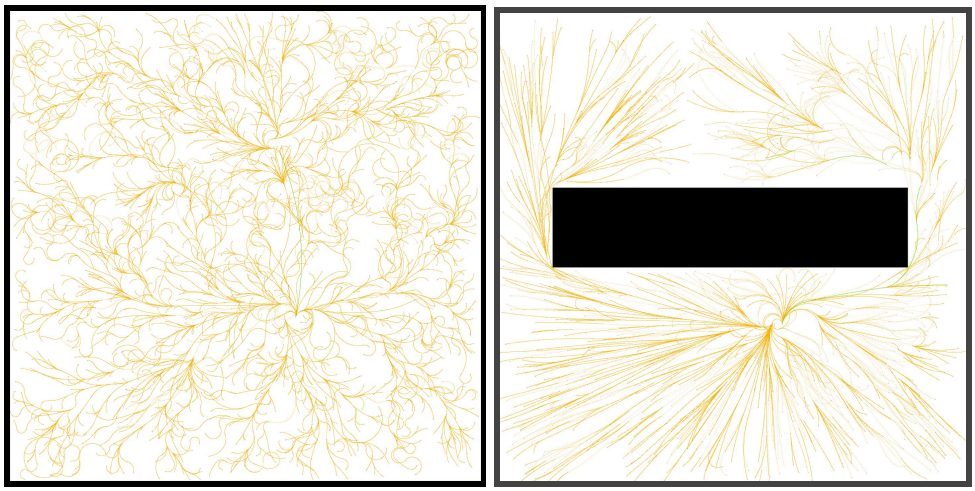


Figure 9: The process of selecting a parent node given the randomly selected point in red is shown. First, a point a step size distance towards the random point is found, then the start point is selected as its parent and a valid circular trajectory is found between the two points, adding the new point to the tree. This process repeats with the end point chosen as the new point, terminating the path.

After the new node is added to the tree the rewiring process is initiated. The validity and distance of a path between the neighboring nodes within the radius defined by Equation 3 and the new node is calculated. With this information, and the distance of the path from the starting node to the new node, our algorithm checks whether the path to each neighboring node could be shortened by passing through the new node. Because the path from the new node to the near node must obey the tangency constraints for both points to be drivable, a hermite spline trajectory is used. By adjusting the configuration of the nodes in the tree to minimize the overall distance from the starting point to each node, the solution is made more optimal with additional iteration.

To reduce the length of an entire valid path, a set number of new nodes are randomly added again. The determination of the number of new nodes to add involves a tradeoff between computation time and path directness, which is discussed in detail in section 3.2.

As the process outlined above is performed repeatedly, the number of nodes in the tree grows and the tree expands to fill the available unoccupied



regions. Figure 10 shows two examples of the tree's expansion in cases with no obstacles and a single large obstacle.

Figure 10: Tree expansion across an open and partially obstructed map are shown. Notably, the parameter gamma is much larger for the map on the right, resulting in paths that are very linear in nature, whereas the smaller gamma value in the map on the right leads to curvier paths.

2.2.2. ROS Implementation

Our team concluded that that it would be best to implement the RRT* planner as an entirely self contained Python class. Making the planner into a class was a good choice because it enabled it to be developed entirely in isolation and in parallel with the other portions of the lab. Additionally, because it has no dependency on ROS for execution, the planner can be called from any python script, allowing for the extensive evaluation, tuning, and profiling presented in section 3.2.

As shown in Figure 11, our path planning algorithm takes in only three arguments: the current robot pose to start planning the path from, the desired robot location that we would like it to reach, and the map of the environment being navigated as a two dimensional numpy array. Given this information, a drivable path is calculated and optimized then returned.

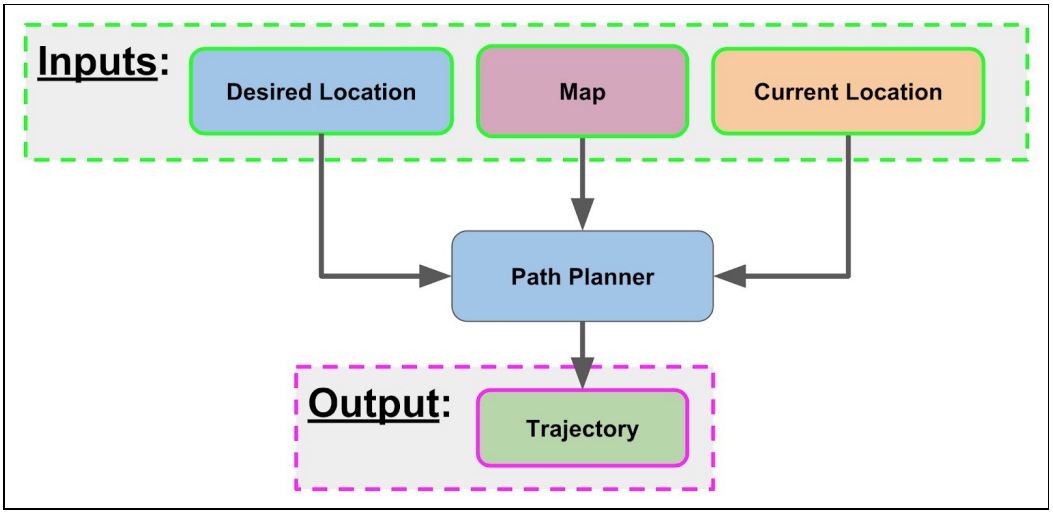


Figure 11: Simplified view of the path planner showing its three inputs and trajectory output. The path planner is a self-contained class that can take any valid input without code changes.

For integration with ROS, a wrapper node was created to translate the desired location published to "/clicked_point" by RVIZ and the current location published to "/base_link_pf" by our particle filter from lab 5 to coordinates in the map coordinate frame that can be interpreted by the sample-based trajectory planner. Once a trajectory is calculated, it is published by the path planner node as a polygon message and also published for visualization in RVIZ. Inside the ROS environment, the structure of nodes is shown in Figure 12.

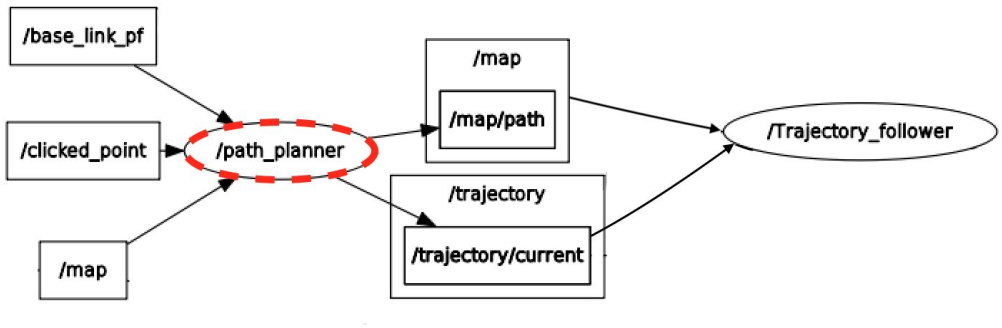


Figure 12: An image captured from rqt_graph showing the relationships between the input and output topics of the path_planner node.

Additionally, if more than one desired location is specified, the subsequent paths are calculated to begin at the end of the previously computed trajectory. This allows for the creation of highly complex trajectories with multiple waypoints. By setting a series of consecutive desired locations around the Stata basement map in simulation, the racecar was able to calculate and follow the loops pictured in Figure 13 and Figure 14.

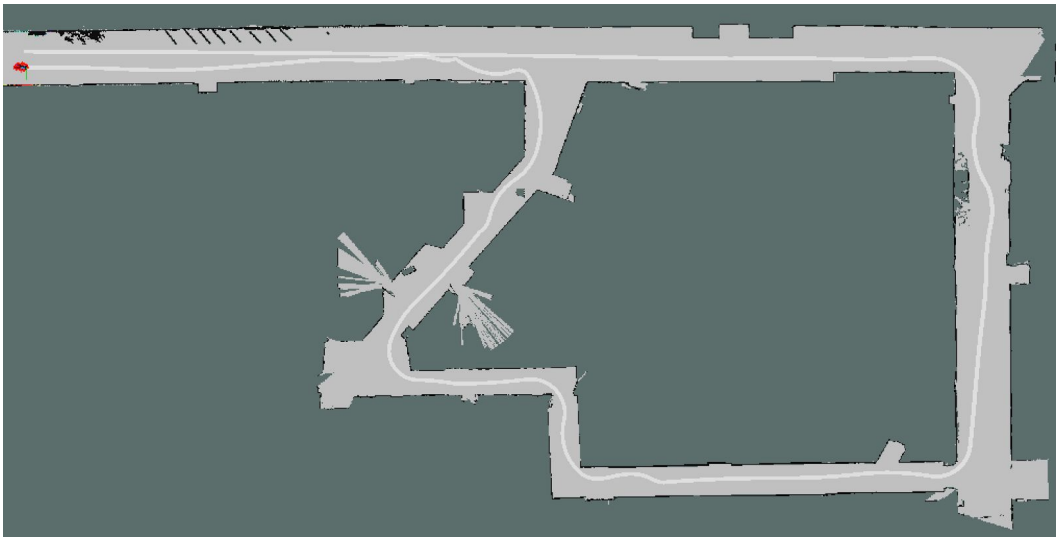


Figure 13: A path completing a loop around the entirety of the Stata basement was calculated and followed in simulation. The loop was achieved by placing four waypoints sequentially throughout the map then allowing the sample-based path planner to calculate the drivable trajectories between them.

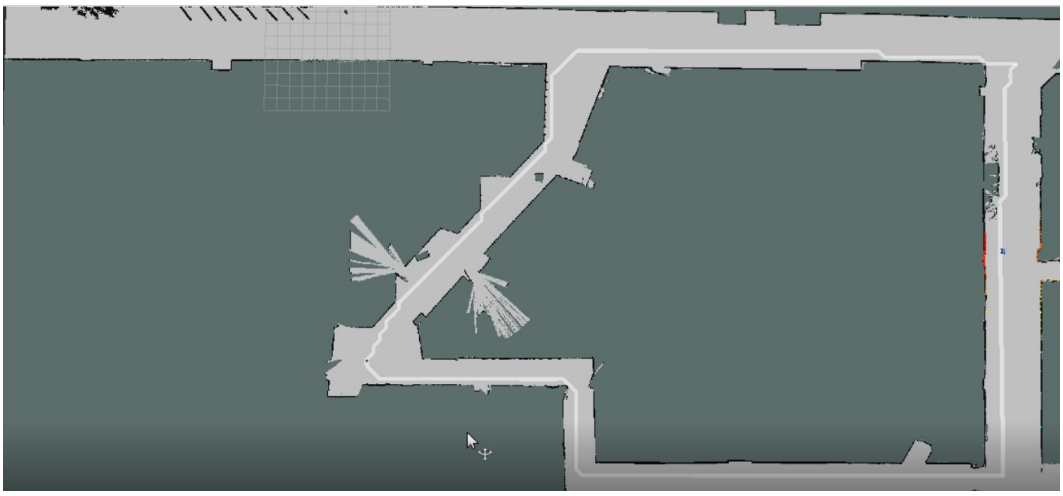


Figure 14: A path completing a loop around the entirety of the Stata basement was calculated and followed in simulation. The loop was achieved by placing four waypoints sequentially throughout the map then allowing the search-based path planner to calculate the drivable trajectories between them.

2.3. Pure Pursuit Trajectory Following *(Authored by Hector Castillo)*

Once a path has been defined for the robot to travel to the desired location, the robot needs a way to follow this path closely and quickly. The robot uses a pure pursuit controller in order to accomplish this goal.

2.3.1. Technical Approach

The pure pursuit controller operates based on a lookahead distance that defines a virtual circle around the robot. The robot finds the intersection between this lookahead circle and the path it is trying to follow, and it uses this point as it's current driving target. Naturally, this intersection point moves down the path as the robot drives towards it, until the end of the path.

The start and end conditions differ from the normally operating algorithm, but are relatively simple. When the robot is first given a new trajectory via the */trajectory/current* topic, the robot checks the distance from itself to every point in the trajectory list, finds the closest point, and sets that point as the first driving target. This makes the algorithm robust to cases where the trajectory doesn't start right in front of the root, and the robot starts somewhere further along the path. The normal operation of the algorithm, discussed in the next paragraph in more detail, operates by looking at the next point in the trajectory outside of the look ahead circle. When the robot detects that the point it is looking at is the last in the trajectory, will set that point as its target until the robot reached a defined *goal_distance* away.

As mentioned in the previous paragraph, the normal operation of the algorithm works by looking at the closest point on the trajectory, p_1, that is outside of the lookahead circle such that the previous point, p_0, is inside or has recently passed through the circle. The algorithm does not require recalculating the distance to several points on the map in order to determine this point; it simply steps the index up each time the point of interest passed within the lookahead distance of the robot. Once this point is determined, the algorithm looks at the line segment defined by this point and the previous point in the trajectory. Naturally, this line segment intersects the lookahead circle. Figures 15-17 detail the calculations and geometry used to find a steering angle for pure pursuit.

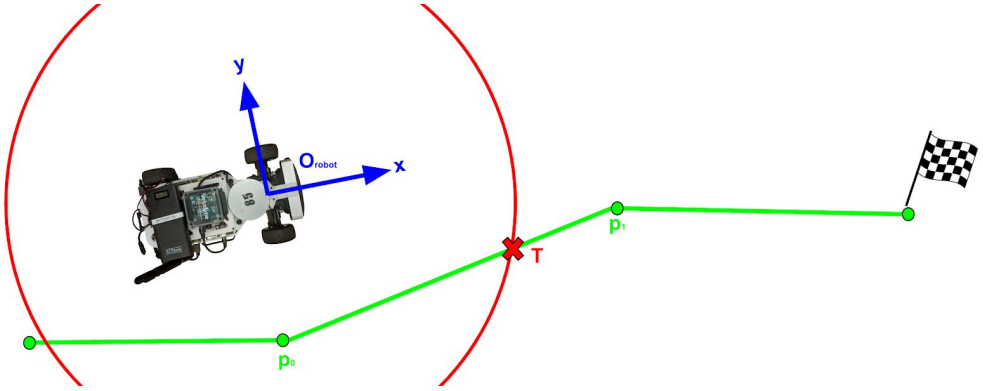


Figure 15: This diagram shows a generic case in which the robot is trying to follow the green trajectory. The lookahead circle, shown in red, intersects the trajectory path at the target point, T.

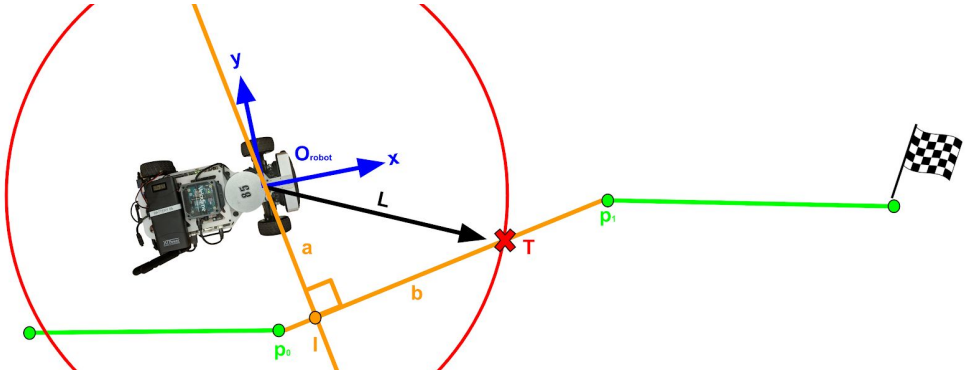


Figure 16: The pure pursuit algorithm isolates the intersecting line segment, calculates the location of the intersection point between the line segment and a perpendicular line passing through the robot's origin. The algorithm calculates the position of the target point using simple right triangle geometry.

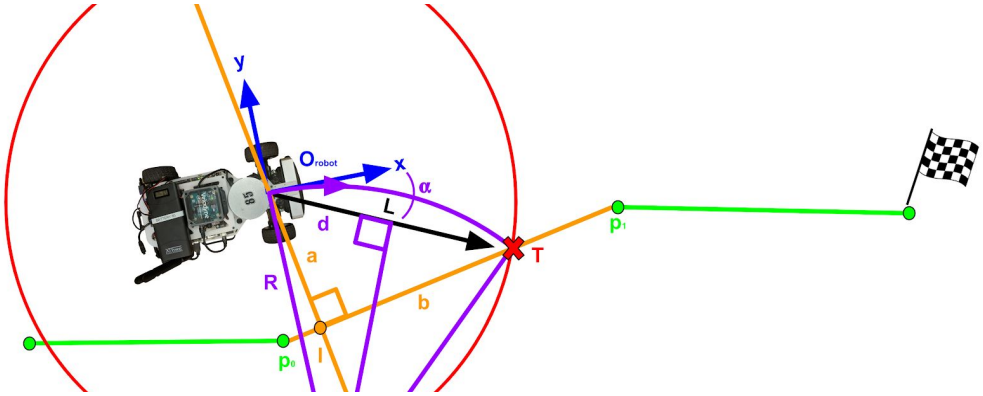


Figure 17: The next step is to calculate the turning radius of the circular path from the robot to the target point, tangent to the robot's x axis. Distance d is half the distance from tehr robot to the target and angle α is the angle of the vector from the robot to the target with respect to the robot's reference frame.

The following equations 4 and 5 are used to calculate the turning radius R and associated turing angle θ, where l is the length of the robot's wheelbase.

Before the steering angle is passed to the publisher, it is clipped within the bounds of the minimum turning radius for the car, 0.34 rad.

$$R = \frac{d}{sin(\alpha)} \tag{4}$$

$$\theta = tan^{-1}(\frac{l}{R}) \tag{5}$$

2.3.2. Performance and Robustness

Through evaluation discussed in section 3.4, we determined that high speeds around turns resulted in larger error in tracking the trajectory. In order to go faster on straightaways and slower on turns, the pure pursuit controller calculates the robot's driving speed as a function of its turning angle θ with parameter α as a smoothing factor.

$$v_n = \alpha v_{n-1} + (1-\alpha)\left[v_{max} - \frac{|\theta|}{\theta_{max}}(v_{max} - v_{min})\right] \tag{6}$$

Additionally, the algorithm models lookahead distance as a piecewise function with a linear regime designed to scale the lookahead distance with driving speed. This model was further evaluated to optimize a tradeoff between stability and minimizing error.

$$L_{fw} = \begin{cases} 0.825, & v \leq 1 \\ 1.5(v - 0.55), & 1 < v \leq 1.88 \\ 2, & v > 1.88 \end{cases} \tag{7}$$

2.3.3. ROS Implementation

Pure pursuit was implemented into ROS in its own dedicated node called *pure_pursuit.py*. This node subscribed to the topic */trajectory/current* to receive the latest trajectory either produced by a path planning algorithm or drawn manually in rviz. The trajectory building tool proved to be helpful in debugging pure pursuit while the rest of the team was working on their path planning algorithms. The pure pursuit node then published an Ackermann Drive Stamped message containing the calculates speed and turning angle. Figures 18 and 19 show the the robot using the pure pursuit algorithm to take a turn around a tight corner using a hand-drawn trajectory and an RRT*-planned path, respectively. Quantitative results regarding the performance and tuning of the controller can be found in section 3.4 under the evaluation of the controller performance.

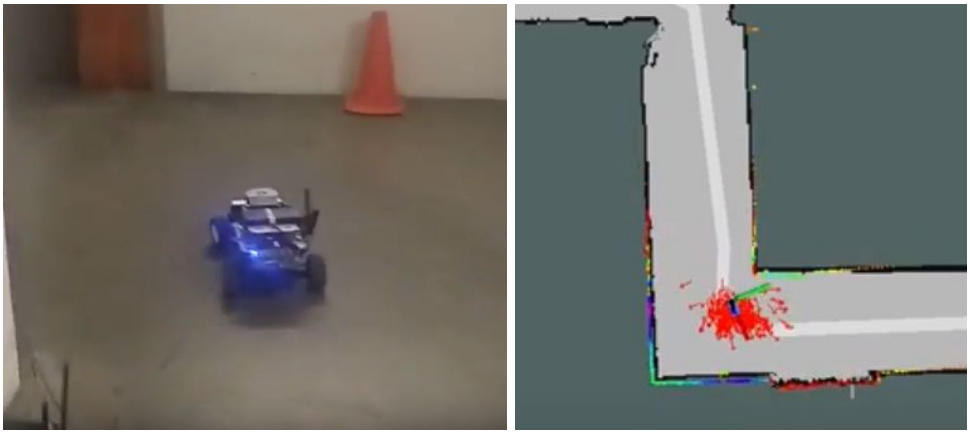


Figure 18: Pure pursuit successfully follows a hand-drawn trajectory and guides the robot around tight corners. The image on the left is the robot driving autonomously through the stata basement. The image on the right is the visualization of the robot in real time as it follows the light grey path.

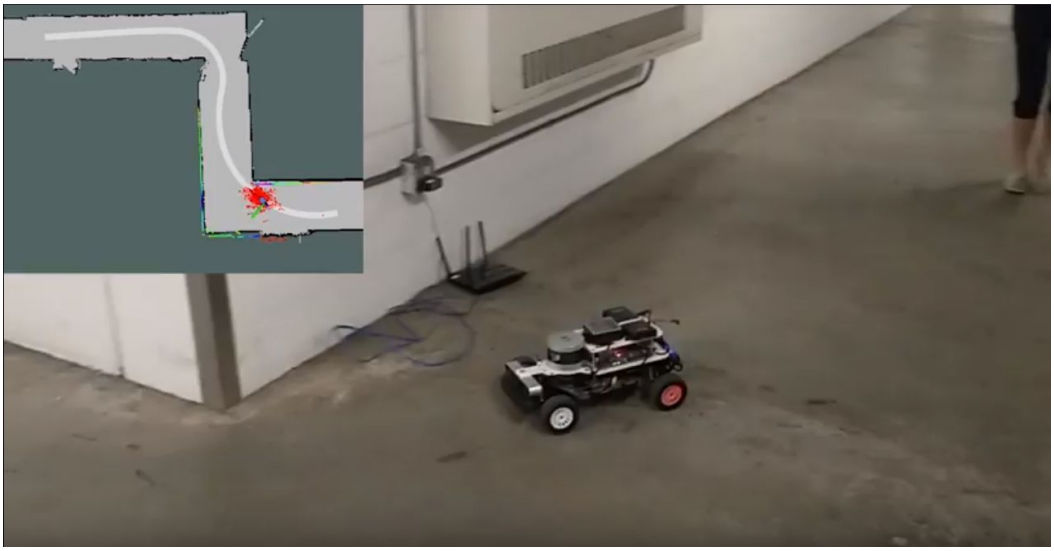


Figure 19: Pure pursuit successfully follows a planned path generated by RRT* and guides the robot around tight corners. The main image shows the robot driving autonomously through the stata basement. The image in the top-left corner is the visualization of the robot in real time as it follows the light grey path.

3. Experimental Evaluation

Our team had to decide which one of our two algorithms we were going to implemented on the racecar. To make as informed of a decision as possible, we performed extensive evaluation of our algorithms. Because lower computation times meant that the robot spends a larger percentage of each run driving, we spent most of our efforts lowering the runtime of our algorithms and evaluated our parameter choices off of this metric.

3.1. Evaluation of Search-Based Method *(Authored by Mitchell Guillaume)*

The A* search algorithm has one major trade off--runtime versus map resolution. Scaling the map down reduced the number of pixels, and therefore the number of nodes, drastically. However, by reducing the number of pixels, were also eliminating our ability to navigate some of the finer details with great precision. In fact, there is a hard limit to the amount that the Stata basement map can be scaled down. At 1/10 scale, one of the major corridors has a constriction in which there is only a one-pixel wide path. If we were to scale the map down any further, then that pixel would be lost and the corridor would be impassable to the search algorithm.

To evaluate how map resolution affects trajectory computation runtime, 100 random start and end points were used to create 100 random trajectories to be found. These 100 trajectories were computed at ⅜ scale, ¼ scale, ⅛ scale, and 1/10 scale. The final length of these trajectories was also recorded. Figures 20 and 21 show the results of the ⅜ scale and 1/10 scale tests respectively.

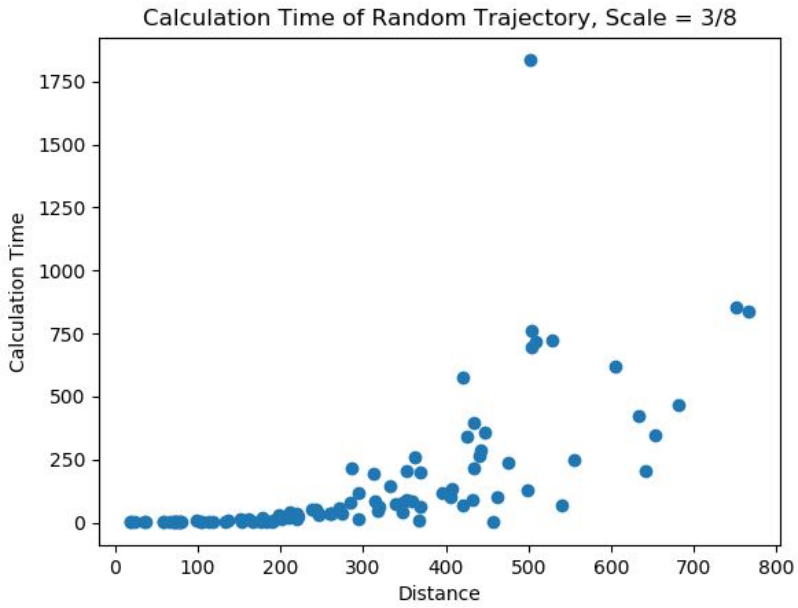


Figure 20: The time required to compute 100 random trajectories of random lengths at ⅜ scale resolution. The calculation time is lower than that of a full-sized map, but much higher than the scaled down map in Figure 17. Trajectories were planned on a Stata basement map.

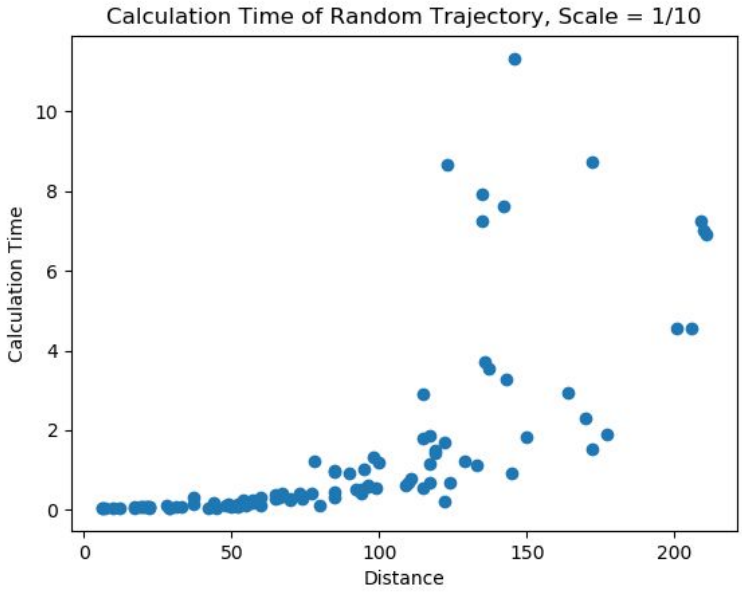


Figure 21: The time required to compute 100 random trajectories of random lengths at 1/10 scale resolution. Lowering the resolution has greatly decreased the calculation time, even as the distance grows. Trajectories were planned on a Stata basement map.

The results indicate that runtime increases exponentially with path distance. More importantly, we can see that in some cases, the runtime for a 1/10 scale map is more than 100 times faster than a ⅜ scale map! To get a clearer picture of this, the mean runtime for the 100 trajectories was plotted versus map scale. The result is seen in Figure 22 .

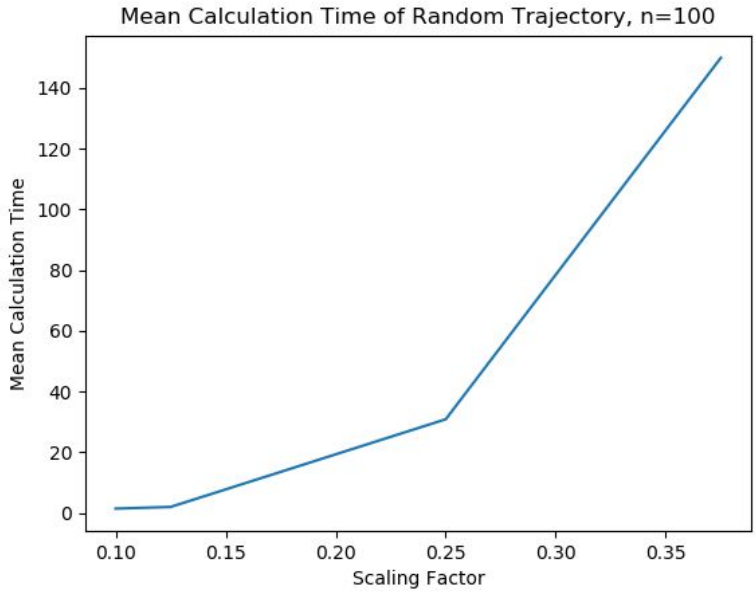


Figure 22: The mean time to calculate a trajectory at a given map scale. The mean time increases roughly exponentially, so calculating a trajectory from a full map would be nearly impossible. Trajectories were planned on a Stata basement map.

Because running the algorithm at 1/10 scale was so much faster than ⅜ scale, and even ¼ scale, our team decided that 1/10 would be our default scaling parameter for running the A* algorithm. We felt that any issues with loss of path optimality due to lesser resolution were not significant compared to the savings in runtime.

To directly compare the A* algorithm to the RRT* algorithm, an obstacle course seen in Figure 25 was created. This map is described in much greater detail in section 3.2. To see how the A* algorithm compared to the RRT* algorithm, the runtime to calculate a trajectory through the obstacle course at various scales was recorded. Figure 23 shows the results of this test. Figure 24 shows the resulting trajectory calculated through the obstacle course.

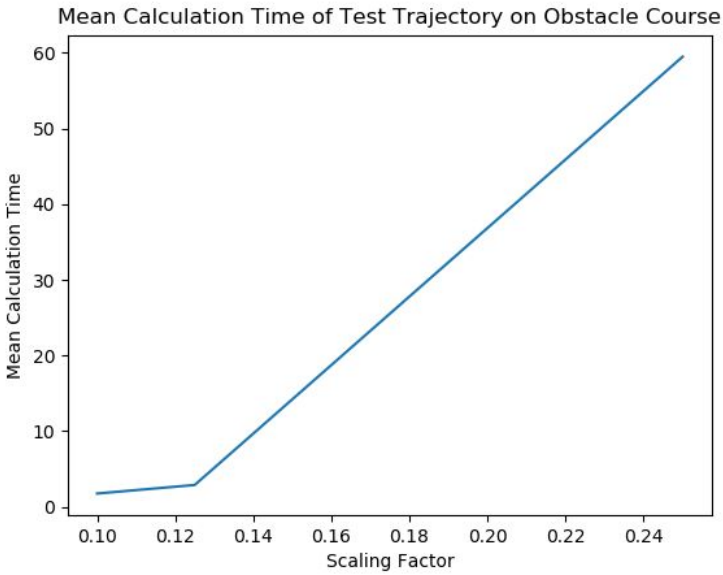


Figure 23: The runtime required to calculate a trajectory through the obstacle course shown in Figures 24 and 25 at a given scale using the A* search algorithm. Just like in the Stata basement, runtime increases exponentially with scale.

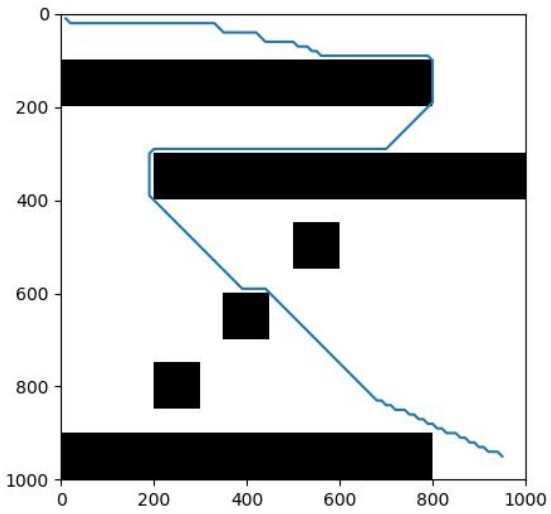


Figure 24: The resulting trajectory through the obstacle course as calculated by the A* search algorithm. The trajectory is the same regardless of scale because there are no small features in the map that are lot when the map is scaled down.

3.2. Evaluation of Sampling-Based Method *(Authored by Daniel Wiest)*

In order to evaluate and tune the performance of the RRT* based algorithm discussed in section 2.2, a number of steps were taken to develop a metric on which performance could be quantitatively measured. As the first step in this process, we created an evaluation map that was representative of the sort of challenges that the sampling-based algorithm would encounter.

The first section of the evaluation map pictured in Figure 25 consists of a thin passage that performs a 180 degree bend and is designed to evaluate the algorithm on its ability to quickly plan paths through regions with limited space. Successfully finding paths through narrow spaces is critical in environments like the Stata basement because as is evident in Figure 1, the hallways connecting the main corridors of the basement are quite narrow and have tight bends.

The second section of the evaluation map consists of a relatively open region with three blocks separating the exit of the first section from the end. This section is designed in this way because another critical function of the path planning algorithm is to find a path between the start and end goal location that minimizes the distance the robot has to travel to traverse the path. Because the second section has a variety of potential paths to the end goal, it enables differentiation between planning parameters that promote the generation of short and direct paths versus those that meander through the available space. The first section alone does not adequately evaluate the directness of the path because there is only one correct way through the obstacle and accordingly we cannot expect large amounts of differentiation in the distance of the paths taken through this region.

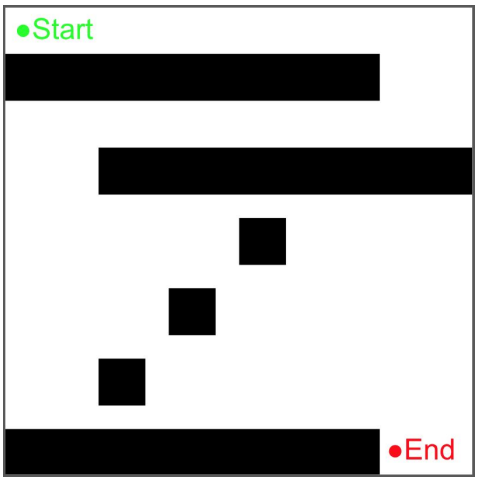


Figure 25: An image of the evaluation map used for the comparison of different algorithms and the tuning of their associated parameters. The map consists of two regions, the first of which tests the algorithm's ability to traverse narrow passageways and the second of which evaluates the algorithm's ability to find the most direct path between the start and end positions.

Parameters are tuned iteratively on the map pictured in Figure 25. Each parameter is swept through a large range of potential values and is evaluated 25 times for each. The distance of the trajectory that the algorithm calculates with its specific parameters as well as the time taken to calculate this trajectory is recorded for every one of these iterations. After all parameter variation finishes (this process took over 5 hours and ran more than 1400 different tests), the resulting mean values for the distance and time taken to calculate the path for all parameters are returned with their corresponding 95% confidence intervals. The data is then shown graphically using the Python graphing library, matplotlib, allowing for insight into the effects of various changes on the parameters. The results of the evaluation of the RRT* sampling-based algorithm discussed in section 2.2 are shown in Figures 26 through 28.

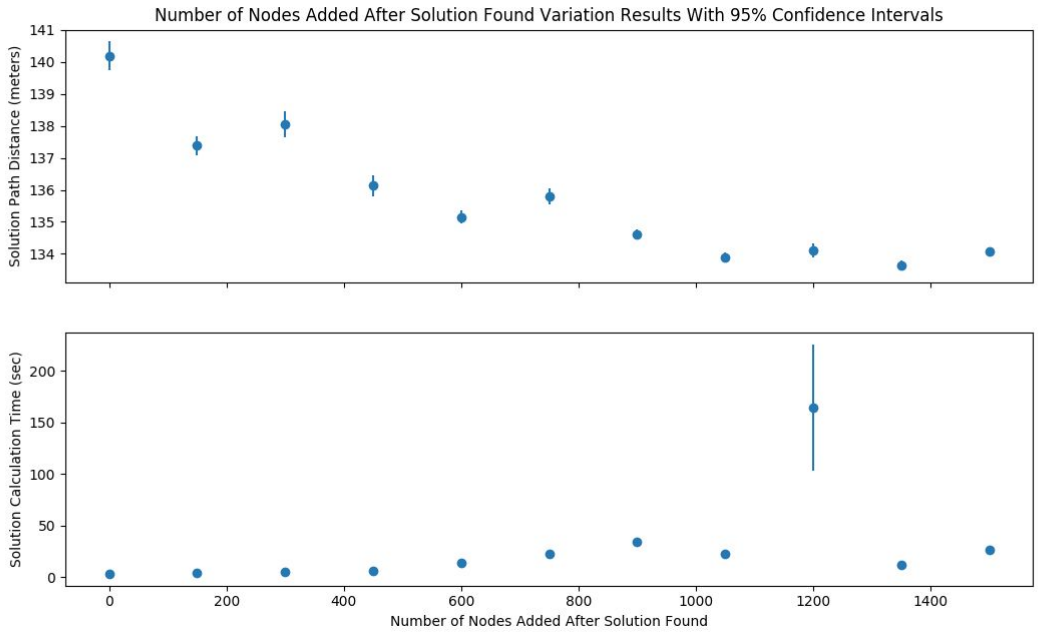


Figure 26: A comparison of the number of new nodes added after an initial path is found and the time and distance of the resulting path is shown. The decreasing trend in path distance is expected because adding more nodes allows greater opportunity for optimization, but this decrease comes at the cost of increased calculation time. The large amount of uncertainty in the 1200 node value is the result of the algorithm not finding a solution for the map after adding 10,000 nodes.

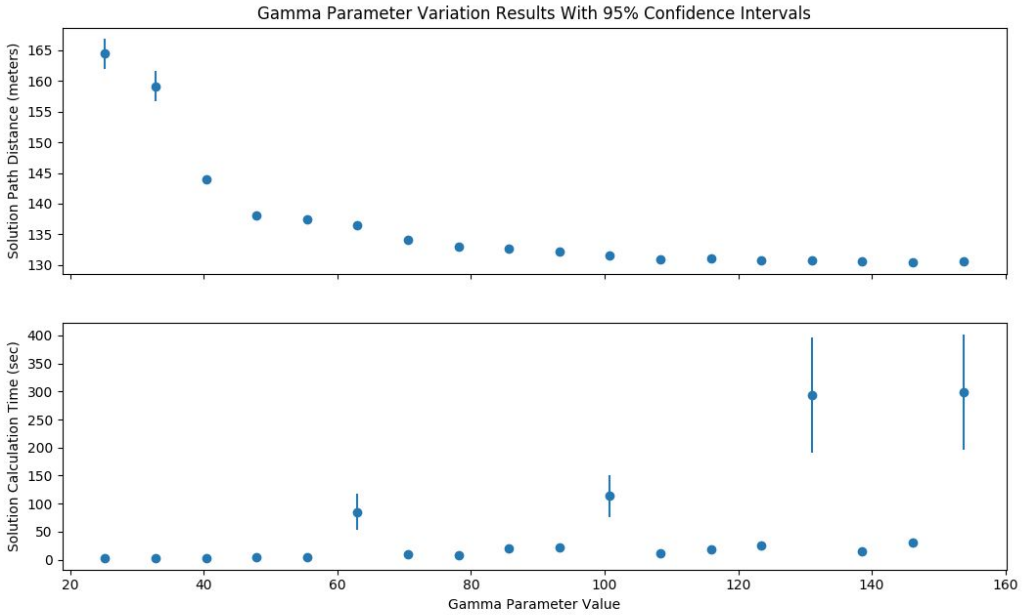


Figure 27: The results of the variation of the parameter gamma, discussed in section 2.2 and used in Equation 3 is pictured. Path distance decreases significantly with increasing values of gamma, but this is also accompanied by an increase in computation time. Spikes in calculation time represent cases were no solution was found by the algorithm after 10,000 nodes were added.

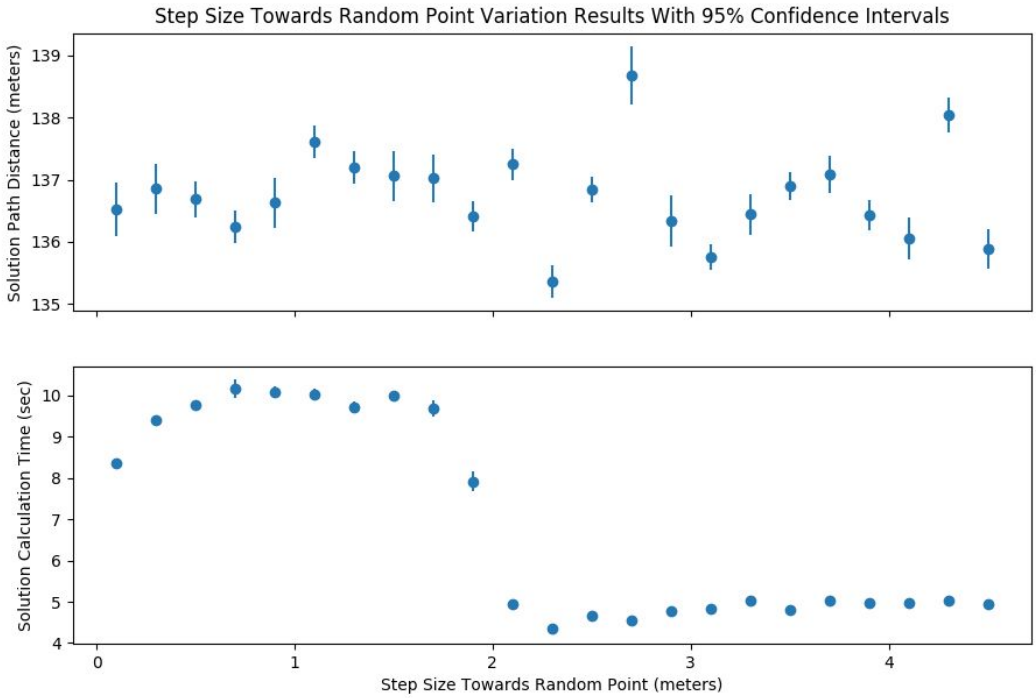


Figure 28: The results of the variation of the step size taken towards each randomly chosen point are shown. The plots above show significant evidence of an ideal computation time range for the step size parameter for values around 2.3 meters. This represents an excellent tradeoff between path optimality (which is relatively unaffected by step size) and the time required to compute the trajectory.

By taking advantage of the insight we gained with the iterative parameter search, our team was able to better assess the tradeoffs between speed of calculation and optimality of the path calculated and ultimately settled on the parameter values specified in Table 1 by leveraging the data collected.

Table 1: Parameters for RRT* found using through iterative parameter search that weighs the tradeoff between computation speed and path optimality.

Parameter	Value
Gamma	110
Step Size	2.3
Number of Nodes to Add After a Solution is Found	165

3.3. Comparison of Sample-Based and Search-Based Path Planning Algorithms *(Authored by Claire Traweek)*

Both of these algorithms successfully found paths between provided points virtually all of the time. However, we found great differences in both runtime and driveability. The A* algorithm had a runtime of around 2 seconds with a 1/10 scale factor during our benchmark, while RRT* had an average runtime of around 5 seconds, despite intensive optimization. However, the RRT* returns a path with greater resolution, at about half of that of the map. A* returns a path that's 1/10 of the resolution of the map. Theoretically, RRT* may not find a valid path, even if one exists, though due to the density of random points, that never happened in our tests on the Stata basements. RRT* does account for the minimum turning radius of the car and curvature constraints, meaning that in practice in the basement, it returns a drivable algorithm 100% of the time, while A* routinely returns tight corners that cause the car to overshoot on turns and can potentially cause collisions.

After testing, we decided to use RRT* on our racecar because it returned a valid, drivable path more often than A*.

3.4. Evaluation of Pure Pursuit Controller *(Authored by Hector Castillo)*

In order to optimize the pure pursuit controller, the next step was to evaluate the effects of different parameters on the controller. The main parameter that was evaluated was lookahead distance. Speed was chosen more or less for safety considerations considering that this lab did not require speed optimization, only safe and accurate maneuvering through the stata basement. Further optimization of speed will be performed in preparation for the final race.

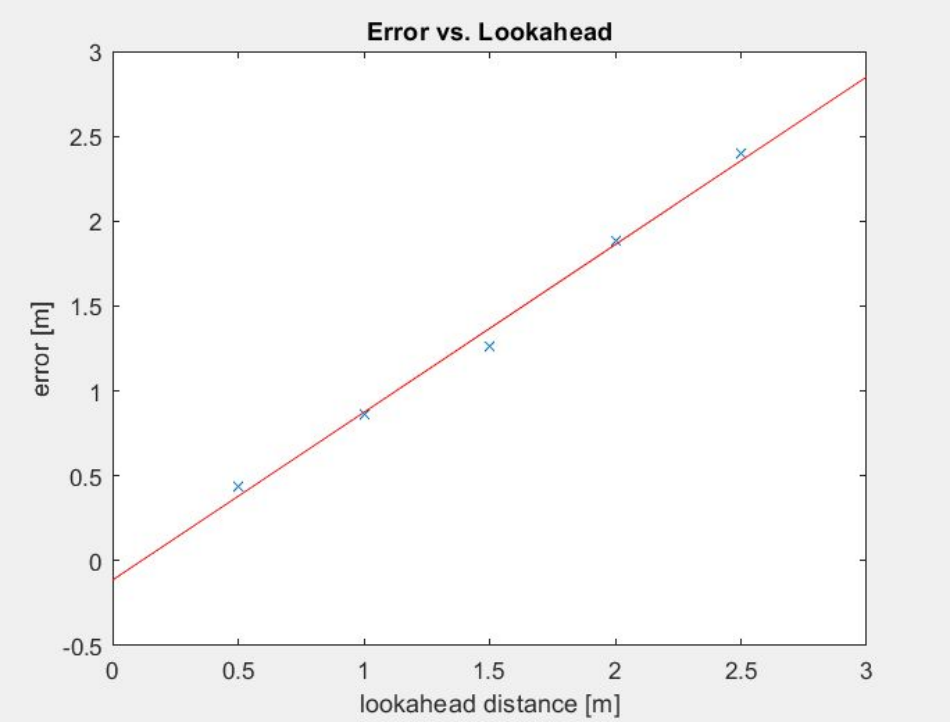


Figure 29: The robot took a 90 turn at five different lookahead distances using the same speed controller. The resulting peak error for that turn is plotted over lookahead distance. A linear regression shows a positive proportional relationship between lookahead and error. This means minimizing lookahead is favorable.

An experiment was performed in order to determine a relationship between lookahead distance and error. Figure 29 shows the proportional relationship that was found. From this plot it seems that in order to minimize error, lookahead distance should be minimized effectively to zero. This would ensure that the robot never deviated from the prescribed trajectory. However, as one decreases lookahead distance, one also sacrifices stability. A lookahead distance that is too small can result in unwanted oscillations and overshoots as shown in Figures 30 and 31.

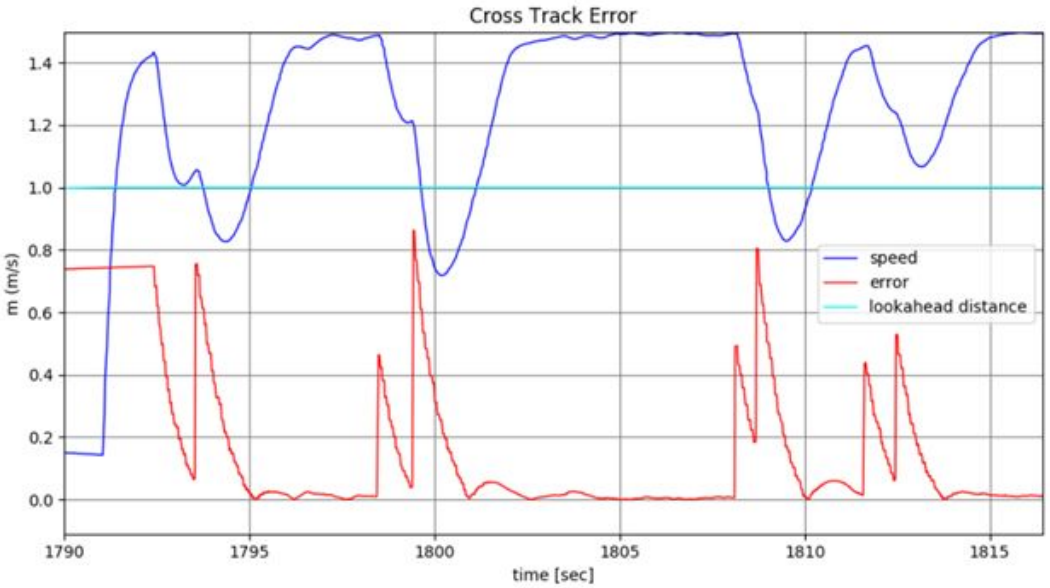


Figure 30: This plot shows the cross track error for the robot as well as the speed and lookahead distance when the robot was given a fixed look ahead distance of 1.0 m. Oscillations are visible on the red error curve near 0.0 error between 1800 and 1805 seconds. The first small peak reaches about 0.05 m of error. Note that the error shown is an absolute value, but the robot actually oscillate across the trajectory line.

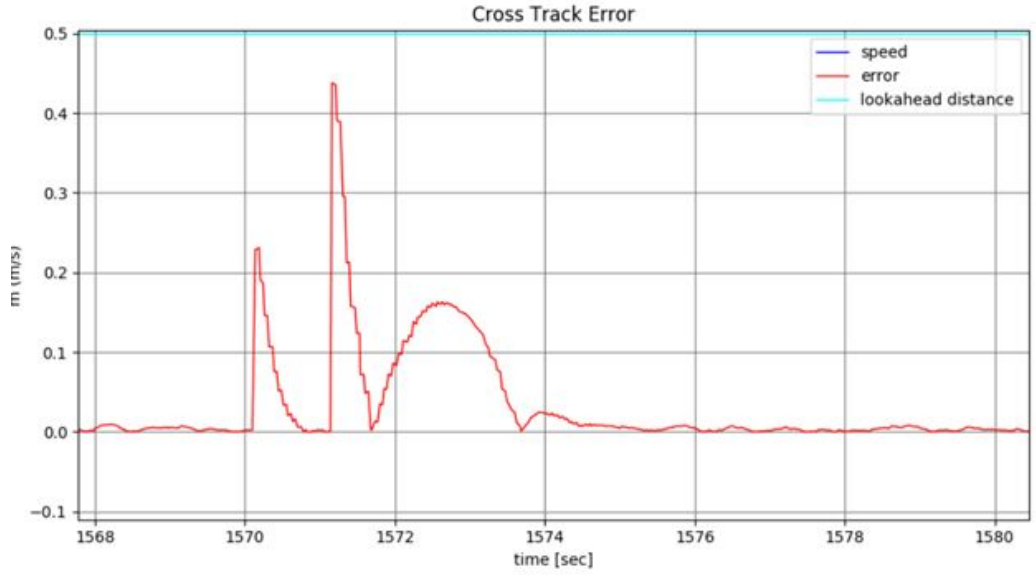
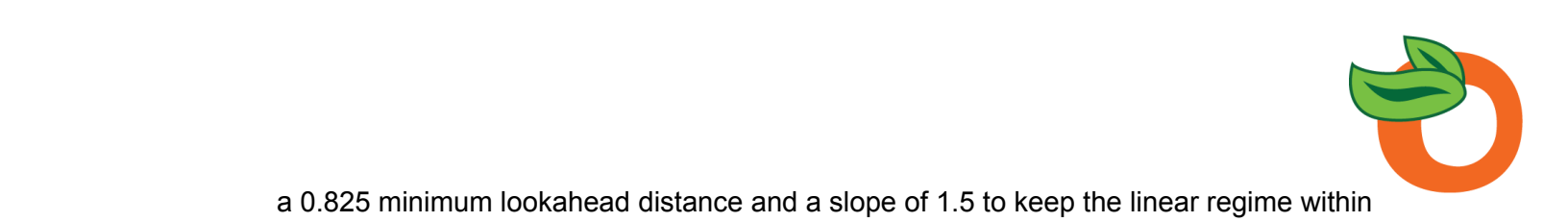


Figure 31: This plot shows the cross track error for the robot as well as the speed and lookahead distance when the robot was given a fixed look ahead distance of 0.5 m. Notice that the first small speak (1572 s) measures around 0.16 m, over three times the same peak with a look ahead distance of 1.0 m.

Both figures show the existence of these oscillations for small lookahead distances. Figure 31 shows a significantly larger oscillation peak than the plot in Figure 30. Thus, a minimum lookahead distance of 0.825 was selected to minimize error and prevent instability. Figure 32 shows the lookahead distance model as implemented with



a 0.825 minimum lookahead distance and a slope of 1.5 to keep the linear regime within a range of reasonable speeds for this lab. Naturally, this model will change for the final race in order to be more optimized for speed. Finally, Figure 33 shows an example of the full cross track error for this model. Note that the spikes are quite tall but fall rapidly. This is due to the inherent error associated with the algorithm switching its focus to a different line segment in the piecewise hand-drawn trajectory.

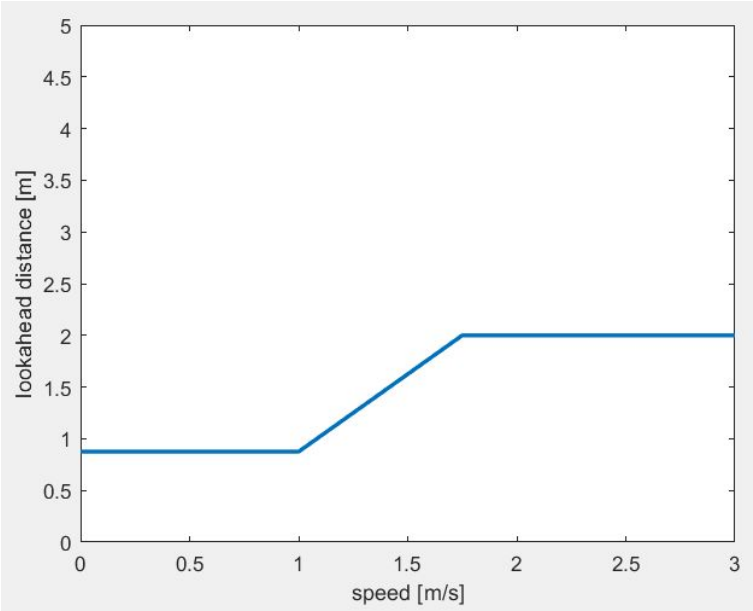


Figure 32: This is the associated lookahead distance model associated with the trial run that generated the cross track error in Figure 27. It shows minimum and maximum lookahead distances at 0.825 m and 2 m, respectively. It also shows the linear regime with a slope of 1.5 that spans from speeds between 1 m/s and 1.88 m/s.

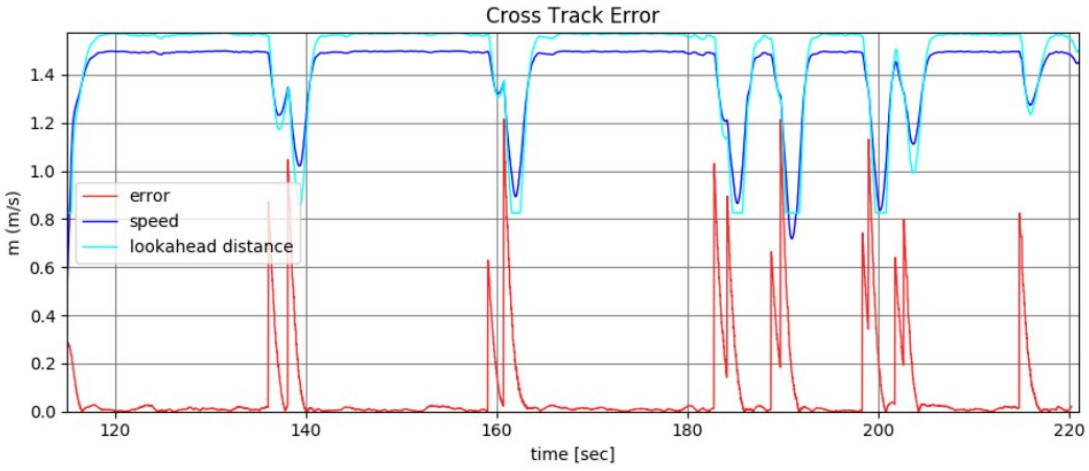


Figure 33: This is an example of the cross track error collected for one run of a hand-drawn trajectory through the stata basement loop. Each spike represents a turn, and the larger spikes signify sharper turns. These turns consist of piecewise line segments with non-continuous curvature. Thus, there will be an inherent error spike when the robot switches focus to another line segment.

4. Integration *(Authored by Claire Traweek)*

In this lab, a path planning algorithm of our choice had to be integrated with pure pursuit and localization nodes. We chose to use our localization algorithm from last lab with our RRT* algorithm (see section 3.3). To ease this process, each node communicated via an agreed upon set of topics using a standard notation for paths.

4.1. Path Notation

Paths were notated by a series of (x,y) points. Because a line with too many points slows down the entire system, and points too far apart create blocky and hard to follow path, it was important to find a balance. For the RRT* algorithm, the path was found using each map pixel as a point, but only every other point was returned. In A*, each pixel was considered as a point, but the initial map was shunk, so each point was 10 pixels on the full sized map. This helped smooth some of the sharper turns when this algorithm was run in simulation.

4.2. Pure Pursuit Integration

Pure pursuit subscribed to the /pose_array topic, which is published to by localization's particle filter and contains and x coordinate, a y coordinate, and an angle, theta, which notates the direction the car is facing. Pure pursuit also subscribes to /trajectory/current, which contains the last path generated by RRT*.

5. Lessons Learned *(Authored by Claire Traweek)*

This week's lab required our team to work on parallel on different components that had to integrate. On top of that, all of these components were constantly being revised to produce the fastest paths possible with the least computation. This lab challenged our team to work together under stressful conditions that required team consensus.

5.1. Technical

Our team faced a number of difficult bugs this week, most of which could have been avoided if the contents of arrays were more closely monitored. Most of these bugs were resolved with debug statements and visual outputs. Comprehensive commenting helped us pinpoint the issues

5.1.1. Search Based vs Sampling Based Path Planning

We explored the benefits and costs of different algorithms as we implemented them in simulation. The A* sample based algorithm will find a direct and valid path to the goal if it exists. There are some caveats in our implementation. Due to the loss of information when the map is down scaled in the A* implementation that is optimized for time, the output trajectory may not be the best trajectory. The information lost can translate into a systematic offset of the trajectory, and in some cases this might make the robot drive a bit too close to a wall. The RRT* sample based algorithm doesn't have this systematic error built in. RRT* also adds a condition to make sure that all the trajectories are drivable. A* could create a path with sharp turns that are difficult to drive due to kinematic constraints of the car, but RRT* considers them. A test path was planned on both RRT* and A* and it found that A* was able to generate the trajectory faster with similar route optimality as one that RRT* creates in a longer span of time. As a result, we choose to use RRT* for future path planning.

5.1.2. Smoothing Using Splines

We found that it was important to connect paths when they were connected by consecutive waypoints, which was difficult to do when a waypoint was placed behind a car. This problem also arose in our sampling based algorithm, when more efficient paths were inserted but required turns that were possible but not if two points were connected by a simple arc. The elegant solution was to implement splines. We explored several types before settling on hermite splines, because it is easy to control their slopes. We learned how to implement and adjust the parameters of these splines to control the end path.

5.1.3. Methods of Optimization

As our algorithms were computationally intensive, we explored several methods of optimization to decrease their calculation time. In the case of A*, we used corner point detection to decrease the number of nodes, but we encountered some problems in that method. We switched to decrease the map size to decrease the number of points searched instead. Several optimizations were applied to RRT*, including reducing the number of nodes placed and the number of nodes optimized. We also learned the the TX2 has several modes, and switching to mode 0 (Max-N) sped some of our computation, at the cost of battery life.

5.1.4. Evaluation Methods

Both planning algorithms and the pure pursuit algorithms had a lot of variable parameters that affected speed at the cost of performance. To decide which parameters to use and which algorithm to use in the end, we established a benchmark we

repeatedly tested. In the case of pure pursuit, this required a lot of repetitive on the car testing.

5.2. Team and Communication

This lab required our team to split tasks among ourselves. The three fairly discrete technical tasks made it difficult to keep track of the progress of the entire team, so using git properly and communicating our progress was important.

5.2.1. Using github effectively

While merges are possible, they can grow messy quickly, especially if the same function was being edited simultaneously. For this reason, some coordination beforehand was necessary. Large functions were broken up, and distinct subsections of the lab were put into their own files. Before any code was written, standards for publishing and subscribing to topics were established. While code was being written, it was committed frequently with descriptive commit messages, so that the next time a team member pulled they could pick up where the code was left.

5.2.2. Updating the team

After a few avoidable merges had to be performed, the team made it a habit to update our each other when major changes were made. This way, team members knew when to pull, and knew about open issues. Team members could also help each other with issues or bugs that were taking a long time to resolve.

5.3. Individual Lessons Learned

The multitude of technical and teamwork related challenges faced during this lab contributed to the development of all members' individual skills.

"Sometimes the work you are doing might not get you to the correct solution, it's important to know when to reevaluate and take a step back. The gift-wrapping algorithm was a good idea to help increase speed, but in our case it was good to scrap that and explore a different approach. The new approach worked and was much easier to implement! " -*Franklin*

"This lab put our safety controller to the test, especially as we gradually increased speed. It was important to test each iteration of our code before running it at full speed on our car." -*Claire*

"The process of exploring a multitude of different implementation strategies for each technical subsection helped me to better understand the various metrics by which solutions can be quantitatively compared then downselected from." *-Danny*

"This week I gained some valuable experience with debugging, and realized how critical it is that the integration process is not taken for granted. Just because your code works well on your machine does not mean it is ready to run in an integrated implementation!" *-Mitch*

"I learned a lot about using visualization techniques in rviz to debug the pure pursuit controller. Being able to see what the robot sees makes all the difference. Although it may seem a bit arduous to go and program in the visualizations, it is definitely worth the effort, and it will save time in the long run." *-Hector*

6. Future Work *(Authored by Claire Traweek)*

This week, we implemented both sampling based and search based path planning algorithms, but only our sampling based algorithm was tested on the car. In the future, we can make improvements to both of these algorithms and test both on the car.

6.1. Making A* Drivable

A* is very fast, and would be the obvious choice to run on the car if it didn't return impossible angles. Limiting the abrupt turns a path can make would make the algorithm workable for the entire basement.

6.2. Refining Minimum Turning Radius

Our planning algorithms and the pure pursuit controller all assume that the minimum turning radius is constant. However, we found in tests that the minimum turning radius gets a lot smaller when the turning speed is reduced, and becomes very large at high speeds. When speed becomes more important, we will need to optimize for this. RRT* can plan tighter (and more efficient) paths if it's integrated with a controller that is smart enough to slow the car.

This document was revised and edited by Claire Traweek and Franklin Zhang