Lab 6: Path Planning and Trajectory Tracking

RSS Team 8
Kevin Chan, Alexander Galitskiy, Josh Hellerstein,
Trevor Henderson, Caroline Hope, Katy Muhlrad
6.141/16.405J Robotics: Science and Systems

Spring 2017

Submitted: April 12, 2017

1 Introduction

Lab 6 focuses on implementation of an effective path planning algorithm for the RACECAR to navigate the MIT tunnel system underneath Stata Center. The team decided to test implementations of PRM/A* and RRT algorithms for safe and efficient path planning, to be discussed in detail below, and then utilize pure pursuit to execute the found path with the RACECAR. Successful completion in this lab will add the ability to plan and execute a path (completely precomputed) in order to navigate the tunnels and avoid obstacles, in addition to previously achieved functionality such as visual servoing and localization, which will set the team up well for the RSS challenge to come.

[Sasha]

1.1 Objectives

The objectives of Lab 6 include developing and demonstrating the capability to plan trajectories in a map from a start pose to goal pose, to program the RACECAR to follow the planned path in the basement using particle filter localization and pure pursuit, and then execute this on the real RACECAR in the Stata basement.

1.2 Theory

There are numerous algorithms for path planning that each have their own strengths and shortcomings. The two main types of path planning algorithms are ones that (1) utilize the configuration space, and (2) sample search algorithms.

Configuration space algorithms include potential fields, exact and approximate cell decomposition, and state-space search. State-space search is then further divided into depth-first and breadth-first searches. While both of these utilize the idea of tree search, meaning that they search adjacent vertices from the current location, the fundamental difference between them is just how they search through the discretized vertices to find a path. Depth-first search means that the algorithm will leave from the start point and search adjacent vertices in the same segment of the tree until it either finds reaches the goal point or is unable to find a path to the goal. Breadth-first search instead builds the tree from the start point, outwards to all adjacent vertices, then again to the next group of adjacent vertices, and so on until one of the entire tree reaches the goal point. While depth first search is generally quicker and does not take the high computing power that is required by breadth-first search, breadth-first is guaranteed to find the shortest path. A variation of breadth-first search, called Dijkstra's algorithm, is to find the most efficient path based on "cost" of the path steps rather than just the number of steps required. One example that is implemented during this lab, and is often used to guarantee shortest path finding is the A* algorithm. A* combines Dijkstra's algorithm of calculating cost of path with a heuristic function to essentially weight future path motions towards the goal. In this way, the A* function is calculating the optimum path at each current point, n, to find $f(n) = c(n) + h(n)$, where $c(n)$ is the exact cost incurred from the start vertex to the current vertex *n*, and *h(n)* is the estimated cost from any vertex *n* to the goal point. Commonly this heuristic function *h(n)* is a straight line between the current point and goal point.

Sampling-based algorithms, on the other hand, do not have to explicitly construct or occupy configuration spaces, avoiding issues with complex configuration spaces and rapidly increasing run-time with increased degrees of freedom. The two that are discussed here and tested for this application to the RACECAR in the Stata tunnels, are Probabilistic Road Maps (PRM) and Rapidly-Exploring Random Tree (RRT), because of wide popularity for applied path planning.

Implementation of the Probabilistic RoadMap algorithm involves sampling *N* points randomly from the configuration space, connecting all points with a straight trajectory, and then deleting any point or line that is through an obstacle. In this way, the algorithm can return a path including set vertices on the path for tracking and set edges to navigate that are known paths without obstacles. This returned set is known as the roadmap graph, and is most simply described as a discretization of the configuration space that can then be used for another path planning algorithm. An increased number of initially sampled points will increase the probability that this discretization is a complete algorithm, but also depends heavily on openness of the space (or rather, how many obstacles versus open paths large enough for the robot), as well as on computation time that an application is willing to run. Possible roadmap paths can sometimes dynamically-infeasible, especially if they are built under coarse discretizations. In order to mitigate this, we would apply not only the Manhattan distance heuristic to the PRM and A* implementation, but also a heuristic favoring small required turning angles to ensure dynamic-feasibility.

The Rapidly-exploring Random Tree algorithm works exactly as the name suggests; a tree is generated from the start point and then random vertices are generated that rapidly extend from the source vertex. Each vertex is connected by an edge as they are generated but the branch is discontinued if the next vertex/edge pair is in the obstacle space. This process continues until a path is found from the start point to the goal point. This method is referred to as "rapidly-exploring" because it generates up to a certain acceptable number of child nodes once the newest best node (closest to goal) is found and appended to the current list of nodes in the path. This means that once it determines the best path so far, it will test a new direction a few times but then moves on when it cannot find a better path to the goal in that direction. In this way it is almost as quick as depth-first search because it is not getting side-tracked by breadth of adjacent nodes. It is also better than breadth-first search because it does not require the same high amount of computation time while still avoiding paths that take it further away from the goal.

[Caroline, Josh]

2 Methodology

In order to complete the objectives, the team needed to implement a path planner algorithm to know points to feed into the RACECAR on the path. The RACECAR then executes this path by using Pure Pursuit. The information for each component is discussed below. We tried two different path planning algorithms: PRM with A* and RRT. We found that RRT, while slower, produced a better path for the robot to follow, so our final path was generated using RRT.

2.1 Inputs and Output of Functions

This section will describe the inputs and outputs of the various functions, as many intermediate classes have been written that are required to interface with each other in order to complete functionality from taking in a map with a start point and goal point, plan an optimal path, and then execute the actions, either in simulation or with the actual RACECAR.

The first steps are drawn from PathPlanner.py, which introduces standards for all methods that will be upheld for all path planning implementations through @abstractmethod. It also includes the following methods for reference by the planners.

__init__(*self, startState, goalState, mapMsg*) : Takes in a map, a start point, and an end point. The map is taken in as a ROS map message that contains dimensions, resolution, and occupancy information to use for obstacle avoidance. The start and end points are given as [x, y, theta] pose values for interface with the localization methods.

computeBestPath(*self*) : Abstract method that returns the (x, y) waypoints of the path as computed by the path planner algorithm. This list is to be used by pure pursuit to execute the path, but the 2D tuples will first be input into a different data structure, a KD-tree, for path implementation based on the robot's local position.

getAllStates(*self*) : Abstract method that returns a numpy array of all states of the search for debugging.

generateRandomPoint(*self*) : Uniformly generates and returns a random point. This point is guaranteed to be within the map and within an ellipse formed around the start and end states as foci. The radius of the ellipse is calculated by half of the distance between the foci plus the predetermined constant planning search radius, as defined in the `__init__()` function.

generateRandomPointGlobal(*self*) : Uniformly generates a random point on the map. This is the same as `generateRandomPoint()` as described above, but does not limit the point generation to the defined ellipse.

isOccupied(*self, points*) : Returns True is the point on the map is occupied by an obstacle, as necessary for path planning in order to not create a path that is included in the obstacle space. Units are in meters.

metersToMapCell(*self, meters*) and **mapCellToMeters**(*self, mapCell*) : Unit conversion methods for calculation of path and use in localization.

The class `PathPlanner_Writer` is used to actually calculate the optimal path to be used, save it to a map, and visualize the goal and points along the path. This class takes in any of the path planning algorithms in order to generate the output. Most of the functions are small, self-explanatory modules, but further detail is provided below for class initialization. This class is what actually passes a planning implementation into `PathPlanner` and then publishes to a `PoseArray` for use by pure pursuit. It is also the file that is called in PathPlanner.launch to run all path planning functions.

> **__init__**(*self, outputFile, stateFile, indexFile, goals, PathPlannerImplimentation, path, states, startIndex, mapMsg*) : The initialization method publishes each the calculated goals (waypoints), states, and the path to a `PoseArray` ROS topic. Note that the goals are kept as pose values in [x, y, theta], but theta is negligible because the path planner does not project a goal theta, only x,y-points, and theta is dealt with in the pure pursuit trajectory. The path planning algorithm to be used is passed into the `__init__()` function. The function calls `self.visualize()` to see the path and progress on the map real-time, and calls `self.computePath()` to call the path planner with any start and end points.

Rapidly-exploring Random Tree methods are implemented between the classes `RRT` and `RRT_Nodes`. While `RRT_Node` takes the randomly generated point from `PathPlanner` and generates a new node for the path if the point is not occupied by an obstacle. The class `RRT` then takes this node and appends it to the current list of nodes. `RRT` then checks if this node is closer to the goal than the current path, and if so the `bestNode` and `bestNodeLength` are updated to this most recent generation. Methods are discussed in further detail here, beginning with class `RRT`. The class `PathPlanner` is passed into `RRT`.

> **__init__**(*self, startState, goalState, mapMsg*) : Initializes the start node, a list `self.nodes` to bookkeep all nodes that are generated and check to be within unoccupied space. Also bookkeeps the current `self.bestNode` for a `self.bestNodeLength` that is closer to the goal than the previous path.
>
> **computeBestPath**(*self*) : Calls `self.iterate()` function for the predetermined number of iterations for the planner to execute. Returns the pose of the best node, `self.bestNode`. Enforced by abstract method in PathPlanner.py.
>
> **getAllStates**(*self*) : Iterates through list of nodes to create state array. Enforced by abstract method in PathPlanner.py.
>
> **iterate**(*self*) : Calls `generateRandomPoint()` from `PathPlanner`. Calls `self.getNearestNeighbor()` to get the nearest node in the discretization, and continues to iterate from that parent a predetermined number of times per planner iteration, called `RRT_NUM_CHILDREN_PER_ITERATION`. For each newly generated point the node is appended to the `self.nodes` list, and then the best path is updated if it is currently closest to the goal.

getNearestNeighbor(*self, searchPoint*) : Takes the randomly generated point as argument and finds the closest node to the current point. Function called in `self.iterate()` in a for loop to continue adding children to parent nodes until an obstacle is hit and ends the branch.

updateBestPath(*self, node*) : Calculates distance to goal and necessary orientation change to reach goal point. If the distance to the goal point is shorter than the previous best path, and the required turn angle is smaller than the maximum dynamically-possible angle, the best node is updated to the found path. Path length is also updated for bookkeeping. The function is called in `self.iterate()` in order to check and potentially update the path for each generated point.

[Caroline]

2.2 Path Planners - RRT

RRT[1] stands for rapidly-exploring random trees. From a high level, it it randomly draws feasible paths in the search space until one reaches the goal. The paths are generated so that they tend to explore areas that are not yet explored - hence being rapidly-exploring.

The reason we chose to use RRT is because the paths it generates take into account the geometry and physics of the car. In particular there will be no sharp corners and any path found with be drivable by the car. We are basically just applying pure pursuit in reverse (randomly)!

RRT works as follows:

- Generate a random point in the search space
- Choose state in the tree that is the nearest neighbor of the random point
- Randomly perturb that state according to the vehicle model and add the new states to the tree

2.2.1 Random Point Generation

We define our search space on a path between start and end nodes p_0 and p_1 to be the intersection the set of unoccupied points and a circle which encompasses p_0 and p_1. This circle is centered at the midpoint of p_0 and p_1 and has a radius large enough to encompass both points plus a constant buffer radius. Points are first randomly generated within the circle and then rejection sampling is used to produce points in the map.

2.2.2 Nearest Neighbors

Choosing the nearest neighbor of a random point is the bottleneck in the RRT algorithm. Linearly searching through the list of states for the nearest neighbor becomes slower and slower as the number of states increases. We solved this issue by randomly sampling a constant number of states and choosing the nearest neighbor from those. We realize that the smaller the sample size, the less exploratory the tree becomes. With a single sample the RRT algorithm would simply be a random walk, which tends to stay isolated to a single space. However we found we found with the right sample size we found a balance between the exploratory and efficiency. We also considered using 2D data structures for querying nearest neighbors, however most common structures like KD-Trees are generally static and would not be useful for a set of points that continues to be updated.

2.2.3 State Perturbation

After choosing a particular state, we perturb that state with a random driving command. We assume the car to be moving at constant velocity, so our commands only consist of a random steering angle in the car's steering range which varies from -0.34 to 0.34 radians. Using the bicycle model of the car, we can calculate the new position of the car after traveling for a constant time Δt. For a particular steering angle δ, the curvature, κ, of the arc the car travels along is

$$\kappa = \tan(\delta)/L$$

Where L is the length of the car wheelbase. For a car traveling at constant velocity v, the new position with be at an angle α and at distance d relative to the current position:

$$\alpha = \kappa v \Delta t/2$$
$$d = 2 \sin(\alpha)/\kappa$$

Therefore the new position (x', y') and new angle, θ', of the state will be

$$x' = x + d \cos(\theta + \alpha)$$
$$y' = y + d \sin(\theta + \alpha)$$
$$\theta' = \theta + 2\,\alpha$$

As we add states we reject (but do not replace) states that intersect with obstacles.

After many iterations of the RRT algorithm are performed, we check for states that are within a certain threshold of the goal state. We backtrace from those states to find the best path to the goal state and return it.

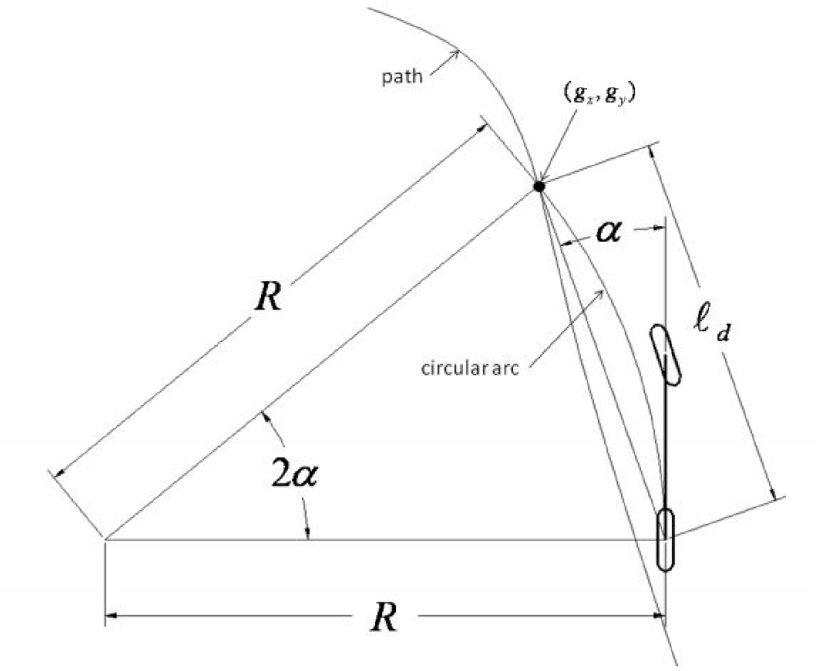
[Trevor]

2.3 Pure Pursuit

In order to make the robot follow the generated path, we implemented pure pursuit control[2]. We adapted the controller from lab 4 to follow points on an imaginary line instead of a bright orange one.

First, we put all of the generated points on our path into a KD Tree for nearest neighbor searching in logarithmic time[3]. Then we search the tree for the nearest point to the robot's current location, determined from the localization. To get a point for pure pursuit to track, we walk up the list of points until we find two points on either side of our look ahead distance, then take the linear interpolation of those points to find one that's exactly the look ahead distance away from the robot.

The point we choose to track starts out in map coordinates. In order for our controller to work, we need to transform it into the coordinate system of the robot.



We define the goal point in the car's reference frame as *(g_x, g_y)*. The lookup to the list of points described above gives the goal points in the global frame of the map. To translate the goal point in the global space to the local coordinate space of the robot, we find the difference in the global space coordinates of the robot's position and the goal point. The norm of this difference gives us the measurement l_d in the figure above. We then apply a rotation matrix to the difference vector based on the robot's current orientation relative to the map coordinates to translate the difference into the car's reference. Given the goal in the relative frame of the car and other constants like the wheelbase of the car, we apply the normal pure pursuit equations to find the required steering angle.

$$l_d = \sqrt{{g_x}^2 + {g_y}^2}$$
$$sin(\alpha(t)) = \frac{l_d}{g_x}$$
$$\delta(t) = arctan(\frac{2Lsin(\alpha(t))}{l_d})$$

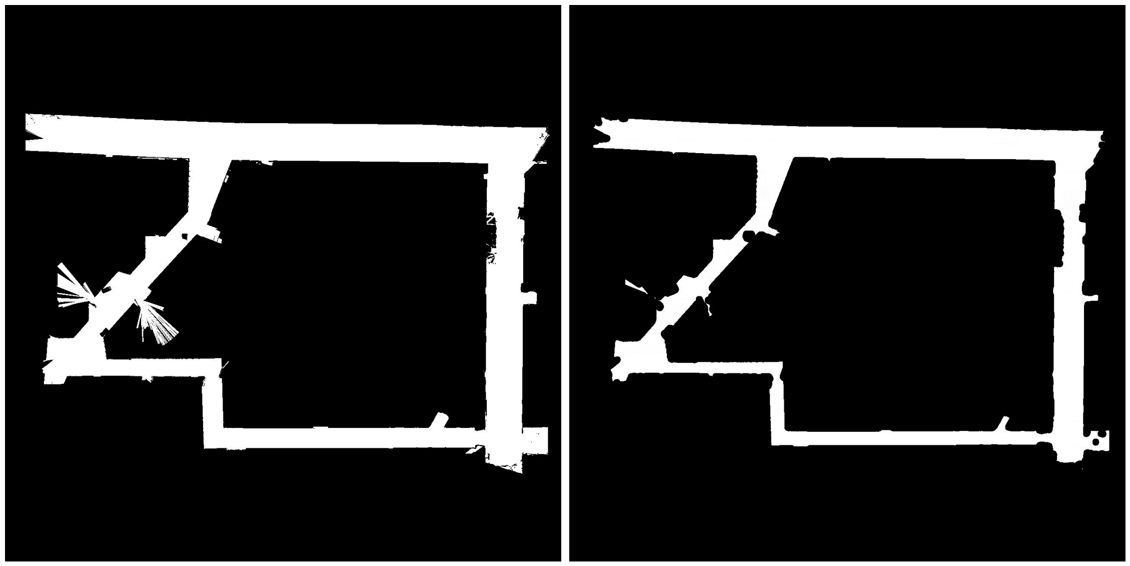
As in Lab 4, we set the lookahead distance of the system to be linearly proportional to the current speed of the car. This factor is referred to as the reaction time and reflects how long it takes the car to adjust to some command.

[Katy, Kevin]

3 Testing and Implementation

3.1 Dilation

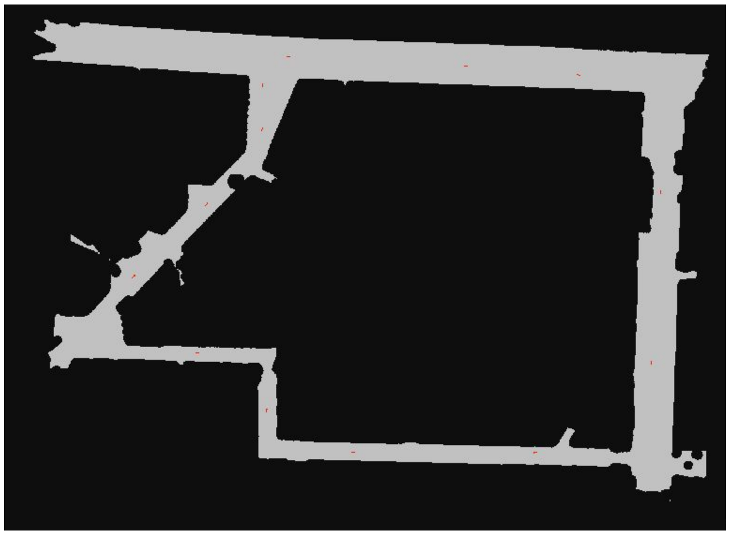
Before applying any path planning algorithms we first directly modified the map. We dilated the occupied spaces in the map, to account for places the robot could not travel because of localization error and its finite width. This was done by convolving the map image with a circle of radius equal to the width of the car plus an error term. This dilation also reduced the search space for path planning algorithms which helped with efficiency. Here we see the map before and after our dilation:



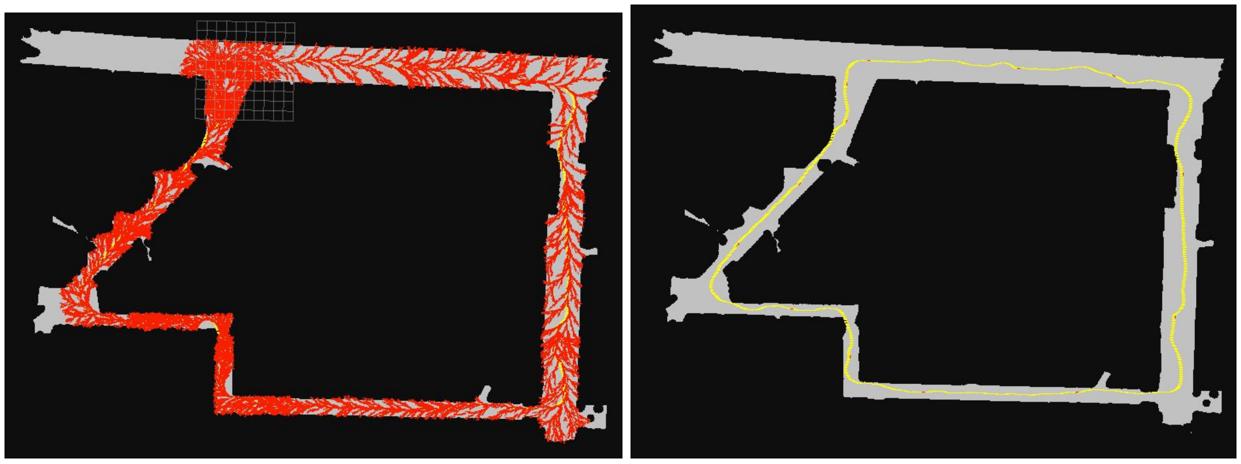
[Trevor]

3.2 RRT

Our RRT implementation generated a good path given a very small set of points to interpolate between. We wrote an interface to RViz which collected poses drawn by the user as goal states to move between. Our path generated by RRT used 12 hand drawn pose states. We placed the states in the middle of passageways so that the job of turning corners was left to RRT. Below we display the goal poses as (very) small red arrows:



After the goal points were found, RRT would search for paths from one state to the next, trying to match positions and angles at the intersections as closely as possible. This process was fairly slow, and we dynamically updated the number of iterations used for a given search depending on the complexity of the passage being traversed. Straight lines were much easier than bendy passages. We set a very small time step size so that the path would be fine enough for pure pursuit. After about an hour of simulation a full path was produced.



In hindsight we should have implemented a bidirectional search. It is very unlikely for the start state to randomly walk to the goal and we had to make our search particularly dense (and therefore slow) to compute the best path. What we should have done is both a forwards search from the start state and a backwards search from the end state. We could then form paths from start to end states wherever the two trees intersect and select the best path from those.

The randomness of RRT also adds jitter to the path. It would be relatively easy to add postprocessing that smooths the path to travel in a direct line between points in straight sections of the map. This would fix things like the extreme dip on the right hand side of the curve which happens because of a discontinuity between path segments.

[Trevor]

3.3 Debugging

The majority of the time was spent on developing and implementing the various path planning algorithms, as the pure pursuit trajectory tracker and localization had been used before in previous labs. The debugging for the planning algorithms were able to be done independently from the full code implementation because it is generic enough to be used on various discretizations.

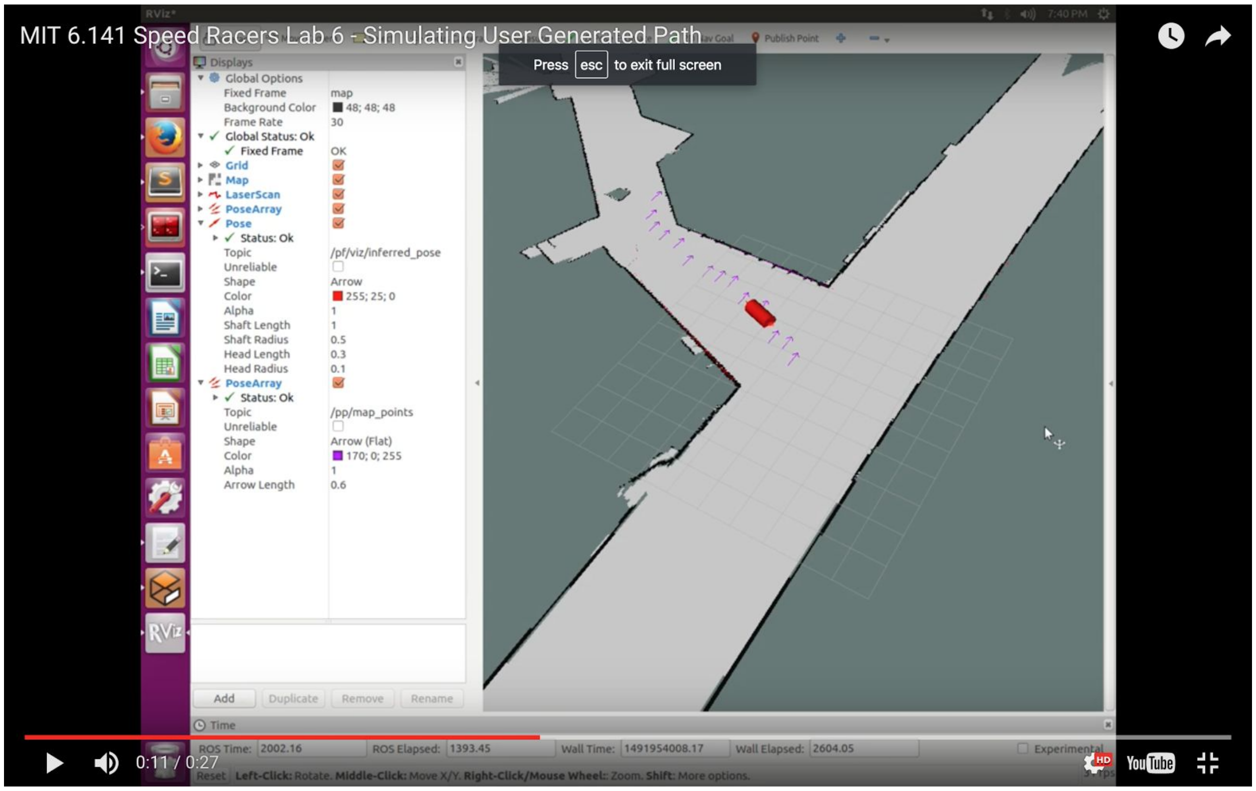
A greater percentage of specific debugging time was spent on integration of the path planners with the trajectory commands from pure pursuit. The first bugs fixed were in the calculations of the turn angle required to reach the next point, but this was mitigated by simply going through the mathematical process and trigonometry. The next issues arose with how the trajectory tracker was choosing its goal point. It always keeps track of the next X number of points in the given path rather than all points, but continually updates these kept points as it moves through the list. Nominally X = 10 points are kept but this can change as necessary from the configuration file. Regardless of the number of points tracked, the goal point for pure pursuit is the next point within 1 meter (the look ahead distance). The issue came that the method was updating its next point to be the very last point that was tracked, often far past the look ahead distance so that the assumptions of pure pursuit were broken. This was fixed by changing how the points were updated, and the tracker worked after that.

3.4 Simulation

Testing the pure pursuit code in the simulation was a long process. At first there was much confusion about what units all of the coordinates were in. Originally we were under the impression that we had to convert points from the map system into some global system, and then those global coordinates into local robot coordinates. Fortunately, we were able to go directly from map coordinates to local robot coordinates.

Once the units all matched up, we had to debug the trigonometry to compute and follow the goal point. This also started out badly, because y and x were flipped on the map. We ended up refactoring our code with more descriptive and clear calculations, which reduced any confusion about the directions of the axes.

After these bugs were resolved, the car nicely follows the manually created path in Gazebo, as shown in the video below.


MIT 6.141 Speed Racers Lab 6 - Simulating User Generated Path
Press esc to exit full screen
Displays
Add
Duplicate
Remove
Rename
Time
0:11 / 0:27

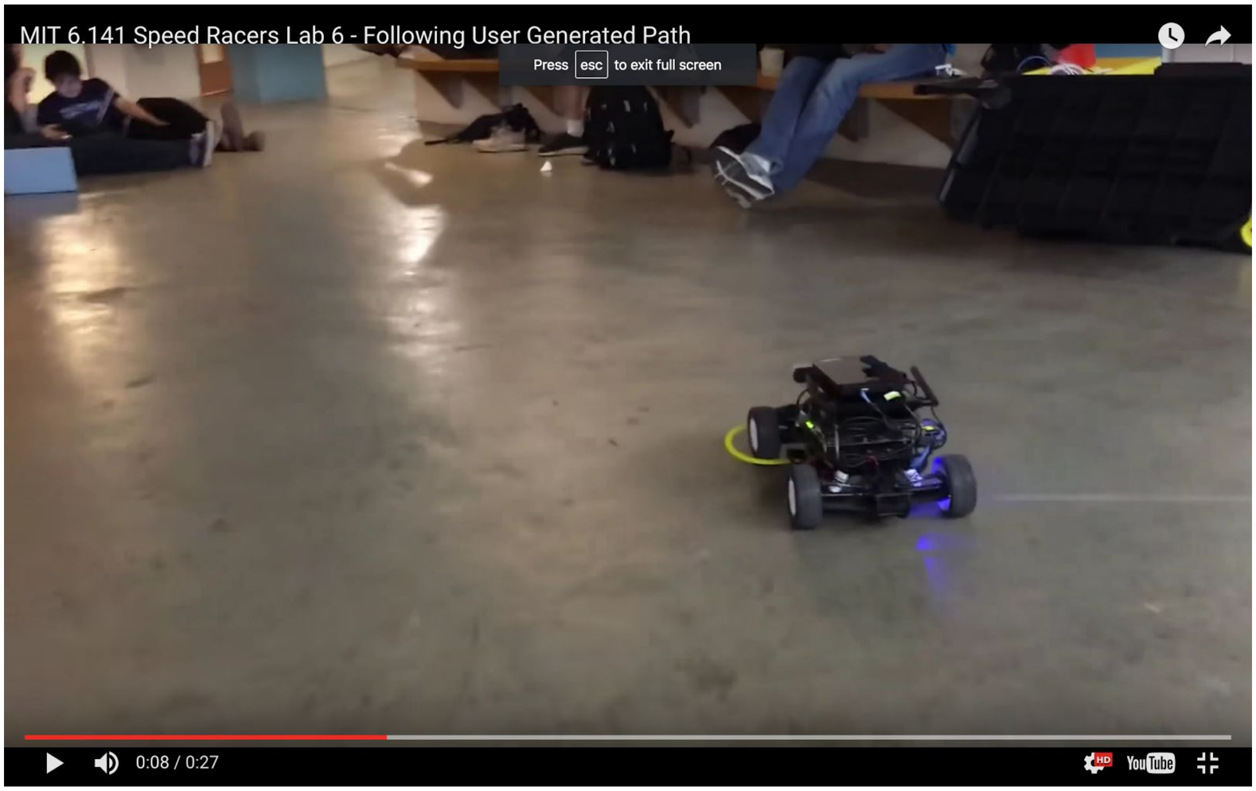
```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
  Fixed Frame          OK
Grid
Map
LaserScan
PoseArray
Pose
  Status: Ok
  Topic                /pf/viz/inferred_pose
  Unreliable
  Shape                Arrow
  Color                255; 25; 0
  Alpha                1
  Shaft Length         1
  Shaft Radius         0.5
  Head Length          0.3
  Head Radius          0.1
PoseArray
  Status: Ok
  Topic                /pp/map_points
  Unreliable
  Shape                Arrow (Flat)
  Color                170; 0; 255
  Alpha                1
  Arrow Length         0.6

ROS Time: 2002.16   ROS Elapsed: 1393.45   Wall Time: 1491954008.17   Wall Elapsed: 2604.05   Experimental
Reset   Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
```

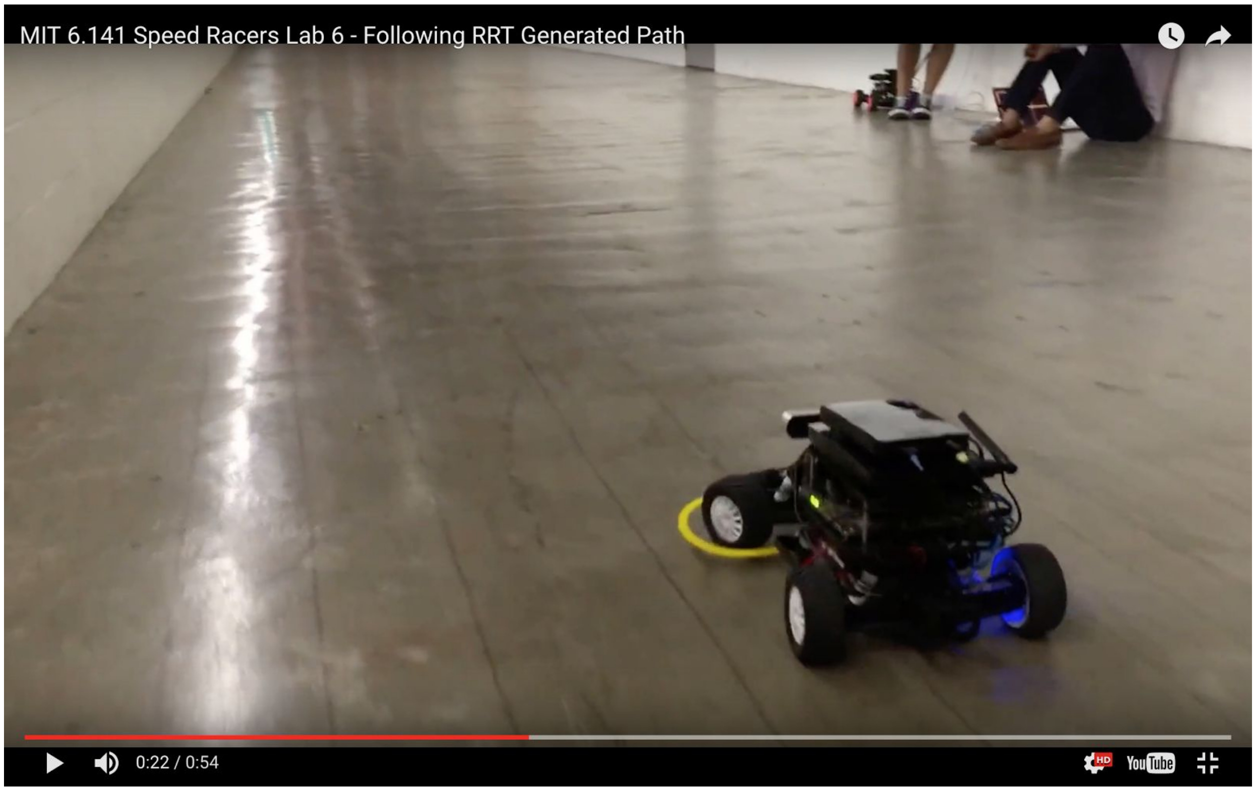
[Katy]

3.5 Tunnel Testing

The first tunnel test was following the manually created path that was used for testing pure pursuit in the simulation. It went very well, following the path almost exactly as it did in the simulation. Towards the end it brushed up against the wall, but that was due to poor point choices rather than the robot missing the path. A video of this run is shown below.


MIT 6.141 Speed Racers Lab 6 - Following User Generated Path
Press esc to exit full screen
0:08 / 0:27

The next test was following the path returned by our RRT algorithm. This test didn’t go as well. The car quickly diverged from the path and oscillated back and forth rather than following a tighter line. We discovered our pure pursuit algorithm had a trigonometry error that was not caught during the testing of the smaller path. After that was fixed and the look ahead distance was increased, the car still did not perform well. We think this had a lot to do with dying wheel motors, so we didn’t have good control over the steering angle. A video of the original test before fixing the bug is shown below.



[Katy]

4 Teamwork

Everybody on the team had various levels of experience with path and motion planning. At the very beginning we decided it was most important to conduct further research into the various algorithms we could potentially use. Shortly after the team split into groups working on writing the RRT and A* algorithms simultaneously, as well as creating the dilated map and working on implementing the pure pursuit algorithm. Although splitting our efforts to work on competing tasks once again created a time crunch towards the end, we believe it was in our best interests to do so, as this would allow us to choose the best working solution in the end.

[Sasha]

5 Conclusion

We managed to successfully complete the majority of this week's objectives. We managed to implement an RRT algorithm that generated a path from a starting to a finish pose in the dilated map of the tunnels. Furthermore, we demonstrated the ability of our RACECAR to follow a short, user generated path in both the simulation and physical tunnels using the particle filter localization and pure pursuit algorithms from previous labs. However, the last objective was not met to the standards we feel are sufficient. While we believe this is due to issues with the RACECAR itself, it highlights our need to plan for such issues moving forward. In the future we hope to be able to avoid road bumps such as these, or at least be able to deal with them in time. Before we continue with the next lab, we will make sure that we can enable real-time path planning and execute it successfully in the tunnels.

[Sasha]

6 Sources

[1] RRT: http://msl.cs.uiuc.edu/~lavalle/papers/LavKuf99.pdf
[2] Pure Pursuit:
https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf
[3] KD Tree: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html