SC 627 - Motion Planning and Coordination for Autonomous Vehicles
Assignment III: Collision Avoidance

Gagan Jain | 180100043

Implementation Details

The overall idea is to avoid dynamic obstacles while maintaining the aim of reaching the goal point. We use a heuristic based planning approach in order to do this. The code can be broadly segmented into four parts, each of them explained below.

We need a goal condition which checks if we have reached the goal. While the condition is not met, we iteratively follow this procedure. First, we generate the Reachable Velocities region incorporating the constraints provided and then sample points within this region. Then, we construct collision cones corresponding to each of the obstacles and filter out the points that lie within any of the collision cones. Once done, the optimum feasible point is chosen according to a heuristic that leads us closest to the goal direction. Finally, we choose this as our velocity plan and take a step by publishing the velocity data for our robot. Each of these steps have been explained in detail below.

The goal condition simply checks whether the robot position is within a certain threshold distance of the goal position. If true, we terminate the planning algorithm. Note that the threshold here has been conservatively set to 0.5, because the heuristic as it will be explained in detail below along with the constraints on the velocity and heading direction does not allow the robot to reach very close to the goal. But this threshold does ensure that the robot avoids all the obstacles and tries to head towards the goal as much as possible.

The first step of the iterative code is to generate feasible samples. We have a maximum velocity constraint and a maximum orientation change constraint. Combining these two, the reachable velocity region becomes a sector of a circle with radius equal to the maximum allowable velocity, the sector angle equal to twice the orientation deviation allowed and the center line of the sector along the current velocity direction. Now, in order to sample points, we linearly discretize along the radial and angular directions, and store these in the form of tuples.

The next step is to filter out the points which lie in any of the collision cones. To construct these cones, we iterate over all the points for all the obstacles and then check collision of each point with each obstacle's collision cone. For the cones, we compute the cone angles by using simple geometry as we know the obstacle radius as well as the distance of the obstacle from the current position. To check collision, we check whether the angle of the vector made by the sampled point from the current pose point with the center line of the collision cone is less than the half angle of the cone. If this is the case, then the point lies within the collision cone and we discard it. If not, we append the point to a list of feasible points.

Now, once we have these feasible points, we need to find the optimum velocity point to stick to. For this, we take a hybrid version of the heuristics mentioned in the paper. We try to take the point which is closest to the goal direction. For this, we find the feasible point which has minimum deviation from the current position to goal line and choose that as our next velocity target. This is done so that we will be moving even though the goal direction is blocked and still we will have some notion of moving in the direction of the goal. As mentioned before, this does not allow us to reach completely to the goal but still the robot is able to move to a point that is away from all obstacles and reasonably close to the goal.

Finally, we take a step using the computed optimum velocity point by publishing it. Before publishing, we convert the cartesian coordinate velocity to linear and angular components so as to be able to work with the publisher. The velocity convert function is used as it is. On every iteration, the subscribers call the callback functions for odom and obstacles. These have been filled in by storing and updating the data in appropriate data structures on every iteration. The odom callback also logs time and the robot trajectory so as to be able to plot them later.

To run the code, we just instantiate the class and call the main algorithm function and generate the X vs T and the path plots.

Simulation results

We get the following results upon simulation:

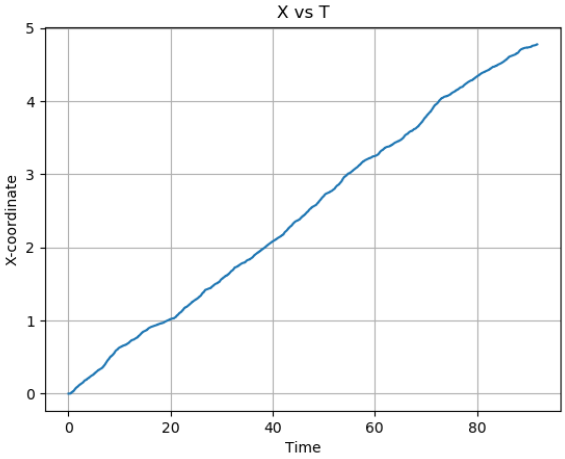

Figure 1: Robot 2: X vs Time

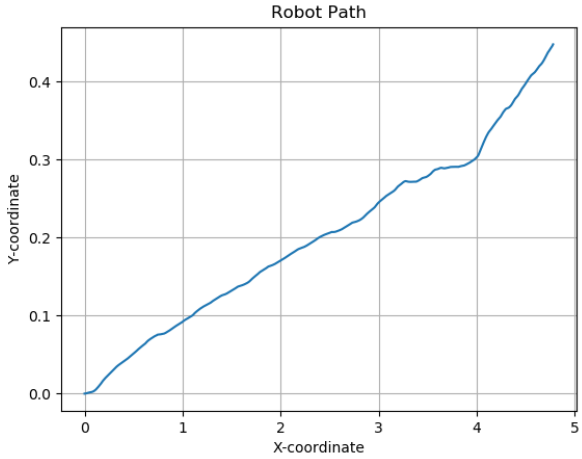

Figure 2: Robot 2: Path Plot