Path Planning for Autonomous vehicle in Dynamic Environment

Prasham Patel
Robotcs Enginering Department
Worcester Polytechnic Institute
Worcester, USA
pspatel@wpi.edu

Purna Patel
Robotcs Enginering Department
Worcester Polytechnic Institute
Worcester, USA
pspatel@wpi.edu

Purvang Patel
Robotcs Enginering Department
Worcester Polytechnic Institute
Worcester, USA
pppatel@wpi.edu

***Abstract*—Path given by many sampling-based planners are not generally drivable as they have sharp turns. While algorithms like RRT family and Hybrid A* provide drivable path but have no real control on the type of path generated i.e., minimum acceleration or minimum jerk path cannot be generated. In this project we developed an algorithm to generate a minimum acceleration path for an Ackerman Drive vehicle using RRT* and Velocity Obstacle. A novel method for obstacle detection is also discussed.**

Keywords—Path planning, obstacle detection, motion planning, Carla, Autonomous vehicle

I. Introduction

Non-holonomic constraint is an inherent feature for most of the wheeled mobile robot. One such system is the Ackerman Drive which is used in most of the autonomous vehicles. The constrained steering angles (due to mechanical design) implies a minimum turning radius. Algorithms like Non-Holonomic RRT and Hybrid A*[13] are used for solving motion planning problem of such systems. However, they consider a fixed steering angle-based motion primitive which can increase computation time. Not only do we have to explore more nodes due to non-holonomic constrains but also, we need to perform collision check for the full path generated by the motion primitive.

Our approach uses knowledge of the path generated without the non-holonomic constrains to generate the path with constrains. While generating the path with constrains we do not use fixed steering angle and velocity motion primitive. Rather we generate a minimum acceleration trajectory which if followed will lead us to desired goal position and orientation. As a side effect our vehicle controller must be able to track the changing steering angle and velocity to follow the desired trajectory to reach goal.

Rest of the article is as structured as follow: part II discusses the related work. Next, part III discusses our proposed methods. Part IV discusses the Platform which is Carla and testing scenarios. Finally we discuss the limitation of our approach.

II. Related Work

Local path planning is important in the development of autonomous vehicles since it allows a vehicle to adapt their movements to dynamic environments, for instance, when obstacles are detected. The paper Peralta, Federico, et al 2020 [10] evaluates the local planning techniques such as the A*, Potential Fields (PF), Rapidly Exploring Random Trees* (RRT*) and variations of the Fast Marching Method (FMM).

A* produces the routes with a smaller number of nodes in the path compared to other algorithms but the computation time increases enormously in the dynamic environment which makes it slow to adapt the avoidance of obstacles. Potential Field calculated the most direct route (lowest difference in distance); however its computation time was much higher than A* and RRT*[8]. The path generated with Fast Marching Method is curved which makes it much more traversable by autonomous vehicles without any further smoothing requirements. But the extremely high computation time makes it unsuitable to act as a path planner. RRT* performs comparable in terms of path produced however when it comes to dynamic handling of obstacles, it outperforms other local planning algorithms.

Decomposition of geometry into primitive shapes can decrease collision checking time significantly. The paper by J. Ziegler and C. Stiller [1] decomposes the square shape of the vehicle into multiple circles. As collision checking for circles is essentially just finding distance from the centre and checking if it is greater than a minimum required value.

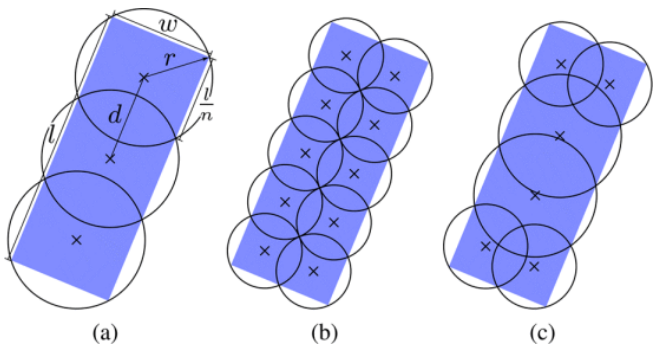

Figure 1: Collision checking circles.

The control algorithm is also an intricate part of the motion planning problem. As the path generated by the motion planner must be drivable in the sense that the controller can track the path within certain error bound. For Ackerman Drive Vehicles many controllers do exist. It should be noted that kinematics of such systems is generally simplified by using a bicycle model. Out of three mainly used geometric controllers [2], [3], we use Stanley controller/ front wheel feed-back control. However, such geometric controllers are not stable in high speeds and feed-back linearized control and MPC are go-to solution in real world applications[4], [5].

The idea of minimum acceleration trajectory is not new, however [6] shows how motion primitives can be generated such that they satisfy acceleration and velocity constrains. Such constraints arise frequently in real world scenarios due to actuator's limitations or limitations due to mechanical design. However, our system is not holonomic in nature thus our approach changes as discussed in later sections.

III. Proposed Method

a. Minimum acceleration trajectory

A cubic trajectory is required for satisfying minimum acceleration condition. However, unlike the holonomic systems our autonomous car cannot control its position and orientation simultaneously. So rather than developing a time dependent trajectory, we just generate the X and Y co-ordinates which if followed will result in minimum acceleration along X and Y axis. The below given equation represents the motion model for the bicycle model [8], these equations are used to calculate the desired orientation and steering angle for a given time.

$$\dot{x} = Vr \cdot cos(\theta) \tag{1}$$

$$\dot{y} = Vr \cdot sin(\theta) \tag{2}$$

$$\dot{\theta} = Vr \cdot tan(\theta) \tag{3}$$

$$\theta = Vr \cdot tan^{-1}\left(\frac{\dot{x}}{\dot{y}}\right) \tag{4}$$

We first approximate the time to reach the end point by dividing the linear distance between start and end point by the vehicle velocity.

$$Ti = 0 \tag{5}$$

$$Tf = {}^{d}/_{v} \tag{6}$$

$$\begin{bmatrix} 1 & Ti & Ti^2 & Ti^3 \\ 0 & 1 & 2Ti & 3Ti^2 \\ 1 & Tf & Tf^2 & Tf^3 \\ 0 & 1 & 2Tf & 3Ti^2 \end{bmatrix} \begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} xi \\ vxi \\ xf \\ vxf \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} 1 & Ti & Ti^2 & Ti^3 \\ 0 & 1 & 2Ti & 3Ti^2 \\ 1 & Tf & Tf^2 & Tf^3 \\ 0 & 1 & 2Tf & 3Ti^2 \end{bmatrix} \begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} yi \\ vyi \\ yf \\ vyf \end{bmatrix} \tag{8}$$

However, this approach does not work when initial or final velocity is zero. For that, we apply the same method but with straight line distance to travel and the initial and final velocity of the car. We match the time step for corresponding X, Y co-ordinate and velocity to get the desired velocity of the vehicle for the current waypoint.

Other better approach is to give a very low velocity instead of zero so we can define the end orientation of the vehicle which is not possible if velocity is zero as ${}^{0}/_{0}$ is indefinite. Figure 2 and 3 shows the path generated for velocity of 5 Km/hr (it is in m/s in figure) with initial and final orientation of ${}^{\pi}/_{2}$ and 0.

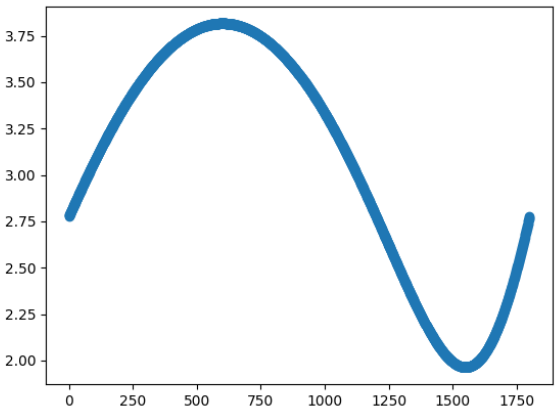

Figure 2: Plot Time (s) vs Velocity(m/s)

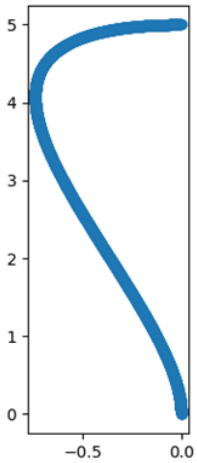

Figure 3: Path generated for 5 km/hr. velocity

b. Collision detection

Normally for mobile robots, the easiest way for collision check is to consider a circular bounding area for both the robot and the obstacle. But in the case of cars driving in narrow lanes

this method won't work. To check accurately for collision, a collision checking algorithm was to be developed which more accurately considers the actual shape of obstacles and the ego vehicle itself. The algorithm must also be simple to compute so the computation time can be minimized.

The initial idea was to use ellipsoidal bounding area instead of circular. Though it accurately maps the rectangular geometry of a car compared to circular bounding area, the computation becomes way more complicated.

To overcome this issue a novel collision check algorithm was developed which uses two different radius coaxial circles to bound the area. The bounding area geometry is shown in the Figure 4.

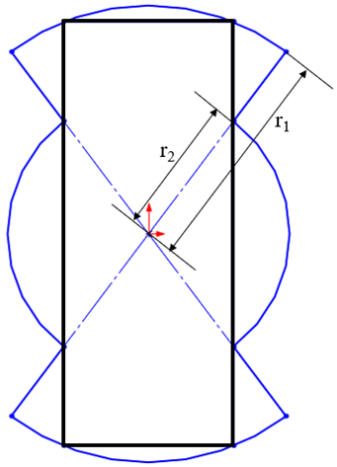

Figure 4: Bounding area geometry

This bounding region is fairly simple to program as it only needs to consider the orientation of the ego and the obstacle vehicle to select the respective collision circle radius. For more safety, the radii of both the circles can be increased to provide more buffer. The following example in Figure 5 demonstrates how the collision checking algorithm works.

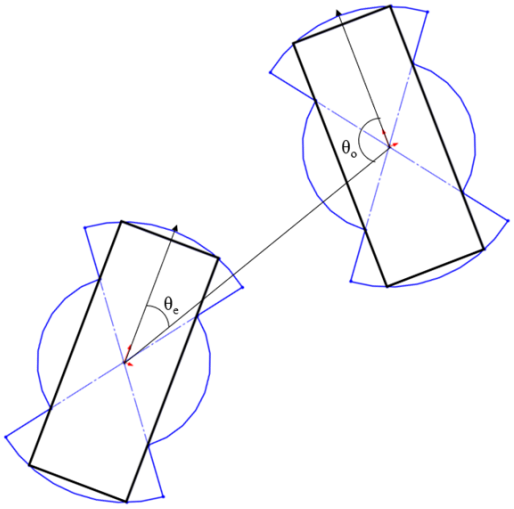

Figure 5: Demonstration of enlarging the radii to provide more buffer

The above example demonstrates how the algorithm checks for collision. Initially it calculates the angles between the vehicles and the line connecting the centres of the vehicles. According to the angles θ_e and θ_o the respective collision radius is decided. In the above-mentioned case, the collision radius for the ego vehicle would be r_1 while for the obstacle vehicle, the collision radius would be r_2. Adding these two radii will give the collision distance between two vehicles. If the actual distance between the centres is smaller than the collision distance, collision is detected.

The advantage of using the above-mentioned method is that there are no complicated equations involved. The algorithm can be implemented using simple if-else statements and still the algorithm works great.

c. RRT* with trajectory generation

RRT* with trajectory generation

```
1  V ← {x_init}; E ← ∅
2  for i = 1,…,n do
3      x_rand ← SampleFree_i;
4      x_nearest ← Nearest (G = (V, E), x_rand);
5      x_new ← Steer(x_nearest, x_rand);
6      if ObstacleFree(x_nearest, x_new) then
7          X_near ← Near(G = (V, E), x_new, min{γ_RRT*(log(card(V)))/card(V)^1/d, η});
8          V ← V U {x_new};
9          X_min ← x_nearest; C_min ← Cost(x_nearest) + c(Line(x_nearest, x_new));
10         foreach x_near ∈ X_near do
11             if CollisionFree(x_near, x_new)^cost(x_near) + c(Line(x_near, x_new)) < c_min then
12                 X_min ← x_near; c_min ← Cost(x_near) + c(Line(x_near, x_new))
13         E ← E U {(x_min, x_new)};
14         foreach x_near ∈ X_near do
               if CollisionFree(x_new, X_near) ^ cost(x_new) + c(Line(x_new, x_near)) < Cost(x_near); then x_parent ← Parent(x_near);
16             E ← (E \ {(x_parent, x_near)}) U {(x_near)}
17 G = (V, E)
18 foreach (x_parent, x) ∈ E
       T = GenerateTrajectory(x_parent, x)
       if not CollisionFree(T) then
           foreach x_near ∈ (X_near − x_parent) do
               if CollisionFree(x_near, x) ^ cost(x_near) + c(Line(x_near, x)) < c_min then
                   X_min ← x_near; c_min ← Cost(x_near) + c(Line(x_near, x_new))
       E ← E U {(x_min, x)}
19 Return G = (V, E);
```

Given the initial and final position and orientation of the vehicle, we perform a simple 2-d RRT* search. The lines 1 to 17 of the pseudo code are same as that for RRT*. Once we get the path from start to end generated without any constrains. We can start generating our trajectory.

Generate trajectory step works by operating on each edge of the path generated. For each edge the required orientation of the vehicle at the end is defined by the orientation of the next edge. This starts from the start node for which the orientation is defined thus the required orientation of the vehicle at the end of first node will be θ1 (as shown in the figure), which is the angle of first node form the x-axis.

After the trajectory is generated, collision check is performed on it, if collision is detected we pop the edge. And rewire the child node to parent of neighbour with least cost other than its original parent node. We again trace the path, generate trajectory and check collision till we reach the goal node.

d. Planning for Dynamic Velocity Obstacles

The major hurdle in path planning of autonomous vehicle is planning for dynamic environment. Simple path planning algorithms like A*, RRT* etc. fails in the task as the position of the obstacles are constantly changing. In such cases, rapid replanning algorithms like D* and RRTX can be used as they are quicker in replanning. But these algorithms cannot be directly applied as they would replan every time the position of the obstacles are changed.

In our case, it is most likely that the obstacles are in motion. Thus, their positions change continuously and so the algorithms would most likely replan on every time step. To overcome this situation, the future setup of the environment must be predicted, and the path must be planned for that predicted future environment. There are many methods that are used for this task.

One such method is using Velocity obstacles [11][12]. This method predicts the set of velocities for which the ego vehicle will undergo collision with constant velocity obstacle. This method helps plan the path in velocity dimension too and thus helps avoid obstacles just by changing the velocity. For example, at a cross section, an obstacle vehicle approaching from the side can be easily avoided by slowing down the ego vehicle. If there are no ways for avoiding collision by simply changing the vehicle velocity, the path must be planned considering the predicted position of the vehicle.

In this project, a function is developed which checks for velocity obstacles and updates the obstacle position and ego vehicle velocity accordingly to avoid the obstacle in motion with constant velocity. For the project purpose, the function is currently able to compute for a single obstacle, but the same concept can be applied for multiple obstacles.

The problem is divided into two parts, 1) Velocity obstacles in the same lane, 2) Velocity obstacle approaching from other direction.

1) Velocity obstacles in the same lane:

When the obstacles are in the same lane and are moving in the same direction, initially the algorithm will check for velocity of the obstacle. If the velocity of the obstacle vehicle is greater than the minimum set velocity of the ego vehicle, it will set the ego vehicle velocity same as the obstacle vehicle velocity and set the obstacle list to None (i.e. it will not approach any obstacle when moving at that velocity). If the velocity of the obstacle vehicle is larger than the max velocity limit of the ego vehicle, the ego vehicle will not change its velocity and will set the obstacle list to none.

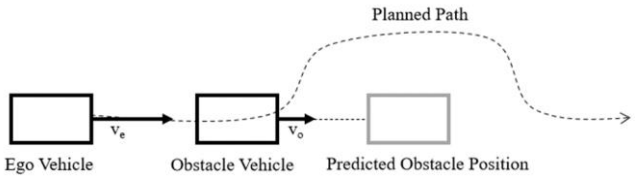

Figure 6: Obstacle is in the same lane with velocity less than ego vehicle's velocity

If the velocity of the obstacle is smaller than the minimum set velocity of the ego vehicle, the algorithm will calculate the time of intersection considering a constant ego vehicle velocity. Then taking the velocity of the obstacle into account, it will predict the position of the obstacle at that time. This position will be appended to the obstacle list and will be used to plan the path.

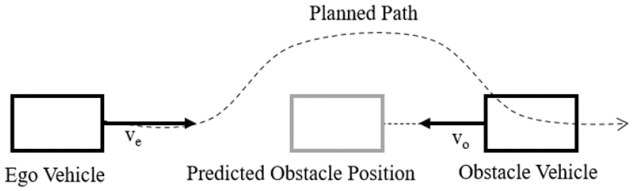

Figure 7: Obstacle is approaching from the opposite lane

Similar steps are followed when the obstacle vehicle is approaching the Ego vehicle. The only difference is the direction of V_o is flipped 180 degrees.

2) Velocity Obstacles approaching from different directions:

Velocity obstacle approaching from random direction makes the problem little more complicated, because to make predictions, along with the intersection point, the collision radius is also to be considered. To make the problem a little less complex, the bounding areas are considered circular.

The first step is to find the intersection point of the ego vehicle and the obstacle vehicle. By solving for equation of lines, the intersection point can be found using equation (10), (11), and (12).

$$x = \frac{cos\psi_e cos\psi_o (y_o - y_e) + x_e \sin\psi_e cos\psi_o - x_0 sin\psi_o cos\psi_e}{sin\psi_e cos\psi_o - sin\psi_o cos\psi_e} \quad (10)$$

$$y = tan\psi_e (x - x_e) + y_e \quad (11)$$

or

$$y = \tan\psi_o (x - x_o) + y_o \quad (12)$$

There are some extra steps need to be implemented in the code to deal with tangential infinities. Due to this limitation, x is defined in sine and cosine.

After the point of intersection is found, velocity range for which the ego vehicle will undergo collision with obstacle vehicle is decided. To find the upper limit of the velocity range, following case is considered.

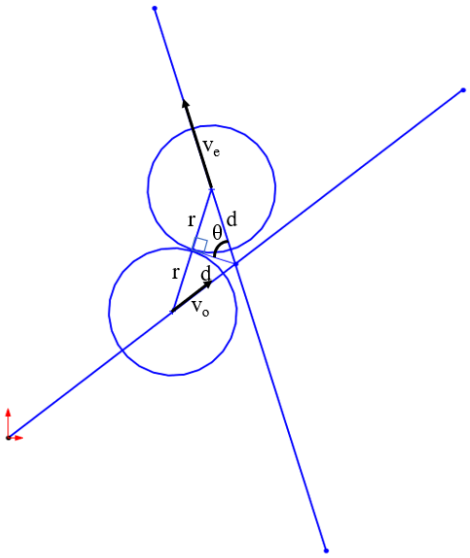

Figure 8: Finding upper limit of velocity of ego vehicle

Here it is considered that the ego vehicle is fast enough to surpass the intersection point but still collided with the obstacle vehicle. From the above figure,

$$d = {}^{r}/_{sin\theta} \quad (13)$$

The maximum distance travelled by the ego vehicle would be,

$$d_{max} = d_{inter_{ego}} + d \quad (14)$$

The time taken by obstacle vehicle to reach collision distance would be,

$$v_{max} = \frac{d_{max}}{t_{min}} \quad (15)$$

Similarly, we can find the minimum velocity for which the ego vehicle will undergo collision as follows.

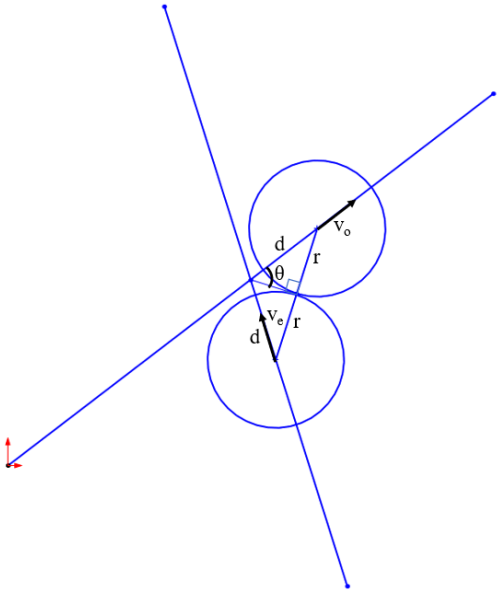

Figure 9: Finding lower velocity limit of ego vehicle such that it will undergo collision

From the Figure 9,

$$d = {}^{r}/_{sin\theta} \quad (16)$$

The minimum distance travelled by the ego vehicle would be,

$$d_{min} = d_{inter_{ego}} - d \quad (17)$$

The time taken by obstacle vehicle to reach collision distance would be,

$$t_{max} = \frac{d_{inter_{obs}} + d}{v_o} \quad (18)$$

The minimum velocity for which the ego vehicle would undergo collision will be,

$$v_{min} = \frac{d_{min}}{t_{max}} \quad (19)$$

Now of the minimum allowable velocity for ego vehicle in less than the lower limit, the algorithm will decrease the velocity to avoid the obstacle. If not, then it will check for the maximum allowable velocity. If it is greater than the upper limit, the algorithm will increase the speed to avoid the obstacle. In both the cases, it will set the obstacle list to none.

If the above both cases are not satisfied, then the algorithm will calculate the time for the ego vehicle to cross the intersection and for that time, it will predict the position of the obstacle vehicle. This position will be appended to the obstacle list and the path will planed accordingly.

e. Vehicle Control

1. Longitudinal Control

Longitudinal controller controls the vehicle velocity. We use a simple PID controller to maintain the velocity as given by the minimum acceleration trajectory.

2. Lateral Control

Lateral controller controls the orientation of the vehicle and it distance error from the reference trajectory. We have used Stanly Controller/ from wheel control in our case. It is the path tracking approach used by Stanford University's DARPA Grand Challenge team[3], [8]. It uses the mid-point of the front axle as its reference point.

Stanly controller has two parts first one accounts for the heading angle of the car to reach the next way point. This simply done by taking the difference between the desired headings and current heading of the car represented generally by ψ.

Next, is the part which accounts for the cross-track error represented by 'e'. The cross-track error is defined as the distance between the closest point on the path with the front axle if the vehicle. Final, formula is as follows:

$$\delta(t) = \psi(t) + tan^{-1}\left(\frac{k \cdot e}{ks+v}\right) \quad (20)$$

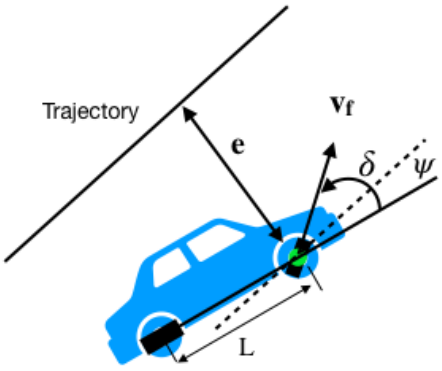

Figure 10: Demonstration of the orientation control and distance error from a reference trajectory

IV. Simulation And Testing

The experiments were simulated in Carla environment [9]. Different test scenarios were simulated, and the developed algorithm was able to guide the vehicle through different situations.

First scenario included a static environment with three obstacles along the route. The Algorithm was successfully able to find a path avoiding all the obstacles. The controller efficiently guided the vehicle through the environment avoiding the obstacles. Following plot represents the path generated.

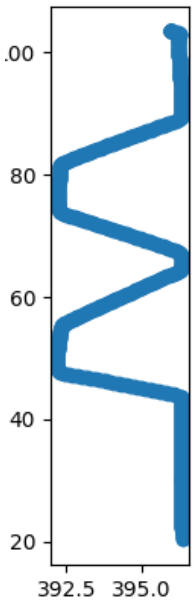

Figure 10: Generated path through the static environment

Second scenario included a moving obstacle in the same lane. This scenario tests the algorithm's ability to deal with velocity obstacle in the same lane. In the first case, the vehicle avoided the obstacle just by decreasing the velocity below the obstacle's velocity. In the second case, the obstacle's velocity was set low and thus the algorithm planned an overtaking maneuver by considering the future position of the obstacle. The path traced during the overtaking maneuver is stated below. Here the green dot represents the initial position of the obstacle, and the red dot represents the predicted position of the obstacle. It should be noted here that the start position of ego vehicle is at x = 5m, y = 165m (i.e. top point in the graph).

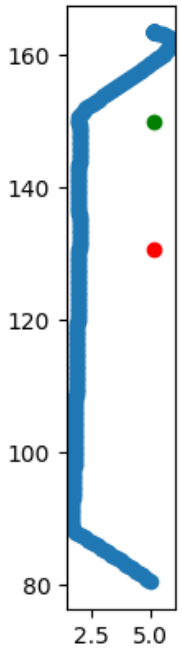

Figure 11: Overtaking an obstacle in motion

In the third case, the obstacle is approaching the ego vehicle and thus the algorithm must plan the path avoiding the collision predicting the future position of the obstacle. Following figure represents the traced path. Here the green dot represents the initial position of the obstacle, and the red dot represents the predicted position of the obstacle. It should be noted here that the start position of ego vehicle is at x = 5m, y = 165m (i.e. top point in the graph).

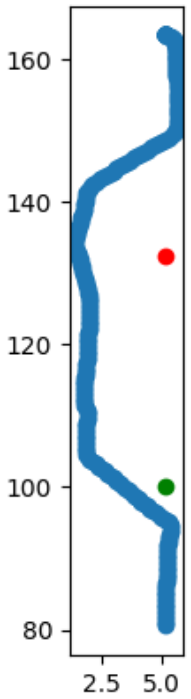

Figure 12: Avoiding obstacle approaching towards ego vehicle in the same lane

Third scenario included a moving obstacle approaching from the side. This scenario tests the algorithm's ability to deal with the moving obstacle approaching from random direction. In the first case, the algorithm successfully avoided the obstacle just by decreasing its velocity. In the second case, the there was no scope of decreasing the velocity further due to the lower speed limit in the code, in this case the algorithm accelerated the vehicle to pass the collision area before the obstacle approaches and avoided collision.

In the third case, there was no scope to decrease or increase the velocity to avoid the obstacle. In this case, the algorithm considered the future position of the obstacle and planned accordingly to avoid the collision. Following figure represents the traced path. Here the green dot represents the initial position of the obstacle, and the red dot represents the predicted position of the obstacle. It should be noted here that the start position of ego vehicle is at x = 5m, y = 165m (i.e. top point in the graph).

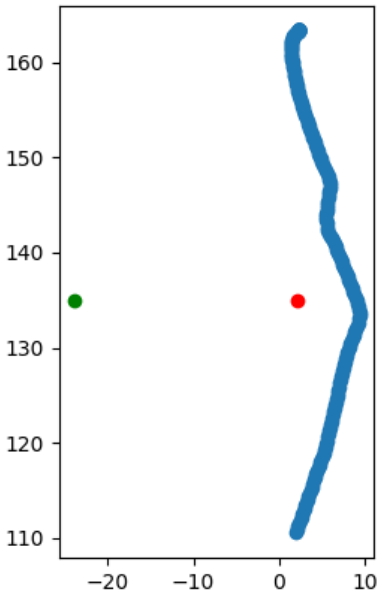

Figure 13: Avoiding obstacle approaching from side

Code for the same can be found on this GitHub page.

V. Limitations

Our work as of now has 2 major limitations, first one being that the planning does not account the reverse maneuver of the car. Thus, planning for situations like reverse parking or parallel parking in narrow gap is not possible. To solve this, changes must be made into trajectory generator as well as in the controller to give reverse velocity as when required.

Secondly, the velocity obstacle can only account for 1 dynamic object moving at constant velocity. If possible, in future we would like to implement RRTx [7] which is dynamic planner. We do account for dynamic obstacle but compute our trajectory offline, with RRTx we would like to make our approach real time.

VI. Task Division

Name	Task
Prasham	• Trajectory Generator • Stane Lee Controller • Collision Check
Purna	• RRT* • Velocity Obstacles • Testing
Purvang	• Environment Setup • Testing

REFERENCES

[1] J. Ziegler and C. Stiller, "Fast collision checking for intelligent vehicle motion planning," 2010 IEEE Intelligent Vehicles Symposium, 2010, pp. 518-522, doi: 10.1109/IVS.2010.5547976.

[2] C. Samson, “Path following and time-varying feedback stabilization of a wheeled mobile robot,” in 2nd Int. Conf. on Automation, Robotics and Computer Vision, 1992.

[3] M. D. Ventures, “Stanley: The robot that won the DARPA Grand Challenge,” Journal of field Robotics, vol. 23, pp. 661–692, 2006.

[4] d’Andréa Novel, G. Campion, and G. Bastin, “Control of nonholonomic wheeled mobile robots by state feedback linearization,” The International journal of robotics research, vol. 14, pp. 543–559, 1995.

[5] P. Falcone, F. Borrelli, H. E. Tseng, J. Asgari, and D. Hrovat, “Linear time-varying model predictive control and its application to active steering systems: Stability analysis and experimental validation,” International journal of robust and nonlinear control, vol. 18, pp. 862–875, 2008.

[6] Liu, Sikang, Kartik Mohta, Nikolay Atanasov, and Vijay Kumar. "Search-based motion planning for aggressive flight in se (3)." IEEE Robotics and Automation Letters 3, no. 3 (2018): 2439-2446.

[7] Otte, Michael, and Emilio Frazzoli. "RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning." The International Journal of Robotics Research 35, no. 7 (2016): 797-822.

[8] B. Paden, M. Čáp, S. Z. Yong, D. Yershov and E. Frazzoli, "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles," in *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33-55, March 2016, doi: 10.1109/TIV.2016.2578706.

[9] Dosovitskiy, Alexey et al. “CARLA: An Open Urban Driving Simulator.” ArXiv abs/1711.03938 (2017): n. pag.

[10] Peralta, F., Arzamendia, M., Gregor, D., Reina, D. G., & Toral, S. (2020). A comparison of local path planning techniques of autonomous surface vehicles for monitoring applications: The ypacarai lake case-study. Sensors, 20(5), 1488.

[11] 1. Fiorini P, Shiller Z. Motion Planning in Dynamic Environments Using Velocity Obstacles. The International Journal of Robotics Research. 1998;17(7):760-772. doi:10.1177/027836499801700706

[12] D. Wilkie, J. van den Berg and D. Manocha, "Generalized velocity obstacles," 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009, pp. 5573-5578, doi: 10.1109/IROS.2009.5354175.

[13] Dolgov, Dmitri A.. “Practical Search Techniques in Path Planning for Autonomous Driving.” (2008).