Execution of Motion Planning algorithm in different scenarios using CARLA Simulator

Aravind Swaminathan

***Abstract*—Currently, autonomous driving is one of the most discussed and researched topic in the automotive industry. In this paper, I have proposed algorithms that can be utilized to make ego vehicle traverse in different scenarios. The algorithms include Clothoid based local planning, rule based decision making , combined graph optimization for velocity profiling, PID controller for longitudinal control and Pure-Pursuit for lateral control. These methods are used to effectively compute a trajectory for different scenarios. These scenarios were simulated in carla environment with the corresponding perception algorithms obtained from the simulator API.**

***Index Terms*—Carla , autonomous driving, motion planning, control, path planning, Clothoids, Simulator, graphic user interface , control**

I. Introduction

Autonomous vehicles are considered as a solution to reduce the road accidents, improve efficiency and convenience for the users etc.[1] The market of autonomous vehicles is expected to be around 58 million vehicles by the year 2030. [2] It is also estimated that 1 out of 10 vehicle will be autonomous by that time. [3].This technology raises relevant controversies, especially with recent deadly accidents. Nevertheless, autonomous vehicles are still popular and attractive thanks to the improvement they represent to people's way of life (safer and quicker transit, more accessible, comfortable, convenient, efficient, and environment-friendly). In the context of this article, motion planning denotes decision making, path generation , velocity profiling and control . Lane change, obstacle avoidance, Stopping for static obstacle are the situations addressed in this paper. After a brief introduction of context of AGV's, the detailed conditions for motion planning are described.

Motion Planning for autonomous on-road driving is a challenging problem. The optimal trajectory exists in high-dimensional space, yet real-time constraints must be met in finding it. Trajectory solutions must adapt to complex and unpredictable traffic. Perception data, which are critical to high-speed driving, are partially observed, noisy, and lagging.To inspire the development of autonomous driving technologies, DARPA organized the Urban Challenge in 2007. To deal with on-road driving, many teams performed lane-based trajectory generation by rolling out trajectories based on lateral shifts from the lane center line. This scheme worked well in the low density, low-speed (up to 30 mph) competition environment, but was too naive for realistic on-road driving in complex dynamic environments.[4]

A. Carla Simulator :

The boom for Autonomous vehicles rapidly exploded after the DARPA challenge and the need for a simulation environment was more prominently required to verify the working of algorithms. One such simulator is CARLA. CARLA has been developed from the ground up to support development, training, and validation of autonomous driving systems. In addition to open-source code and protocols, CARLA provides open digital assets (urban layouts, buildings, vehicles) that were created for this purpose and can be used freely. The simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors, maps generation and much more.[5] The most important features of carla include flexible API, Autonomous Driving sensor suite , Fast simulation for planning and control, Maps generation, Traffic scenarios simulation, Traffic scenarios simulation

B. Problem Statement

Today's autonomous cars will have to undergo lot of complex scenarios which are creating a nightmares for existing algorithms. Also these algorithms needs to tested for different use cases for their accuracy and robustness. Generating the path for these complex dynamic environments is really challenging. But for the path to be generated, there needs a separate decision making module which should act as a brain to instruct the ego vehicle to generate path appropriately. With this path, the velocity computations needs to be smoother in order to avoid jerk while vehicle is in movement. To support that, we would combine the trajectory with a smooth controller to perform the control action on longitudinal and lateral movement. In such cases, we need to have different blocks operating effectively to solve the problem. Without compromising the computation time, one needs to perform the perception-control pipeline of the Autonomous driving framework.

II. High Level Motion Planning Framework

The autonomous vehicle motion planning problem can be summarized as providing a reference trajectory, in continuous domain, that meets the specified requirements of safety, comfort, progress and energy efficiency, and which can be executed by the vehicle hardware, taking into account the vehicle's dynamics.The task is typically decomposed into three subtasks: (i) behaviour planning; (ii) path planning(Global and local) with Collision avoidance; and (iii) velocity planning and control. Behaviour planning is the process of making the highest level decisions that the vehicle should undertake in this current environmental conditions, such as to do maintain the lane, follow a lead vehicle or do a lane change. These decisions are also called as high level control actions in different articles.

These manoeuvres are to be followed up the subsequent modules, path and velocity planners. BP acts as a brain to the entire AD framework and so the input to this system is very crucial. Perception is, therefore, a crucial ingredient of behaviour planning with many of its own challenges, including noise, occlusions and sensor fusion. In this work, these errors due to perception is neglected by obtaining the perception related information from the simulator API's and the focus is mainly on the developing the logic and algorithm for Motion planning framework[6].

The objective of path planning is to devise a safe and smooth trajectory from the autonomous vehicle's current position to a goal position, by avoiding obstacles, satisfying comfort requirements and generally respecting kinodynamic constraints. The abstract actions chosen by the BP must therefore, take into account the actual state of the ego vehicle and environment [6]. To ensure a comfortable jerk less movement, Velocity planning plays an important role in smoothening the trajectory with appropriate velocity. This ensures minimal longitudinal and lateral jerk during the vehicle transition.Both the Path and velocity planning are collectively called as low level controller in this article. The detailed framework in block is given in Fig 1. Once the trajectory is generated by the low level controller, the 2D controller(lateral and longitudinal) controller will chose one point based on the lookahead distance and compute the commands for lateral and longitudinal action.

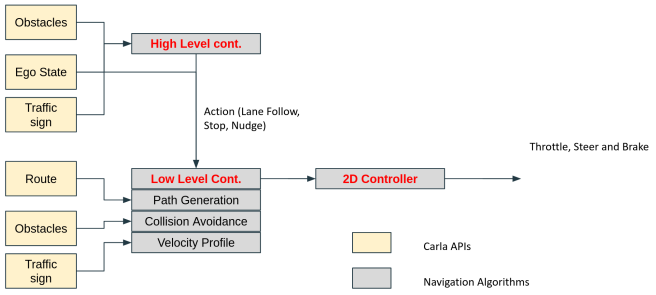

Fig. 1. High Level block

A. High level controller:

The integration of the BP within the motion planning framework is critical since this constrains the other elements and the overall architecture. In this framework, the behaviour planning process is done on a planned Global path. . The BP restricts the complexity of the local planning algorithm by providing a single high-level manoeuvres at each step of the planning process. This makes the planning process simple but it also has a disadvantage. For example, BP needs to generate a path that is executable by LP and controller, in this case, BP has to solve partial problem related to LP which makes computation redundant. Although there are other learning based algorithms that can be implemented to solve the BP, this work is mainly concerned with the rule based approach. A simple way to avoid conflicting trajectory decisions by LP and BP, new methods involve generating a set of goal points for the LP to generate local paths[7]. In this way, LP can choose one best path out of all possible goals based on cost computed for each path. The cost computation can be based on some heuristics. This approach reduces the interaction between LP and BP where there maybe a case that the path generated by LP is not feasible enough to execute the maneuver provided by BP. The high level behaviour planner pipeline is given in in Fig 2

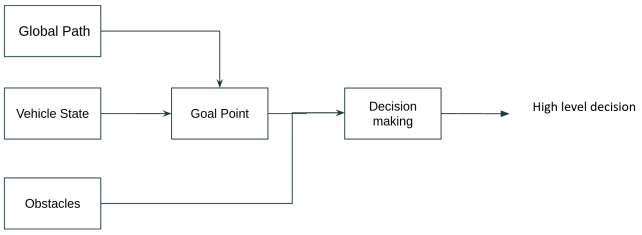

Fig. 2. High Level block of behaviour planner

1) High level Action space selection: The action spaces that are chosen in this work are majorly three, LANE_FOLLOW, LANE_CHANGE and STOP maneuvers. The lane follow is a case, where ego vehicle will have to continue in the lane and there are no other actors involved in this scenario. This action provides the instruction to LP and VP to maintain lane center and also to maintain the speed limit set by the user/ max lane speed limit. LANE_CHANGE in this work is considered only for static obstacle. When a static obstacle is detected on the current lane. Ego vehicle will sense the other lane and check if there aren't any collisions to transition on to the other lane and does a lane change. A graphical representation of this is shown in Fig 3

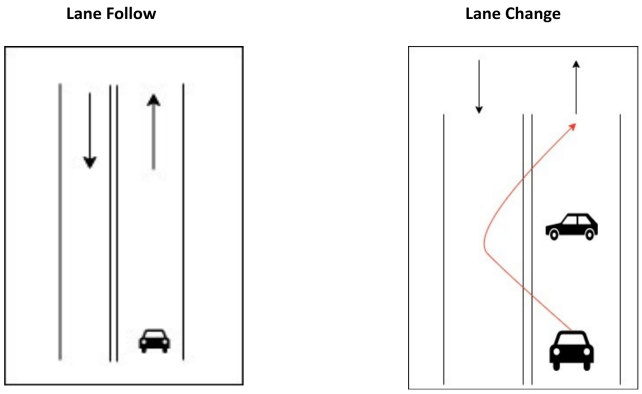

Fig. 3. Lane follow and Lane change maneuvers

The next maneuver is STOP in which a partial decision is taken by the LP based on the collisions. This cannot be considered as a High level decision with respect to this work, but should be considered as a maneuver that needs initial input from the BP. After the decision of LANE_CHANGE, the LP will check for its path collisions on other lane with dynamic obstacles. This adds up the constraint of collision avoidance and so all the paths on other lane will be rejected. The scenario is explained in Fig 4

2) Mathematical formulas: The environment is populated with lot of actors, $N = \{N_0, N_1, ...N_k\}$ with N_0 being the

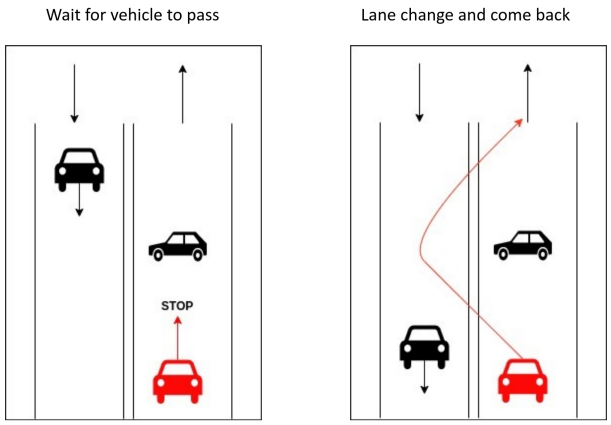

Fig. 4. Stop maneuver

ego vehicle. Other agents $N_k, k\epsilon\{1..,K\}$ has either static or dynamic motions which has a specified position/path $x_k, k\epsilon\{1,..K\}$. The path of ego vehicle x_0 which will be computed by the ego vehicle is supposed to get an action input from the High level controller . As seen from the Fig 2, the input to this module is obstacle positions/paths, denoted by $x_k, k\epsilon\{1,...K\}$, global path which the vehicle has to follow g_k , vehicle state x_0 . The output of this module is goal point g_p and the high level decision R_d. The computation of goal index is very simple, as it involves computing the closest point of ego vehicle onto the global path g_k and use the lookahead distance to compute a goal point g_p. The closest distance of waypoints is given by

$$P_c = min\{|g_k^* - x_0|\} \tag{1}$$

Once the point index with closest distance is obtained the goal point can be computed from the closest point by computing the arclength along the global path. In such case, the global path needs to have close points for the generation of accurate goal point, as arc length is computed over its abscissa. The distance at which the goal point should be considered is given by

$$g_{p0} = P_c + \sum_{n=m}^{T} arclength(g_k^{(n)}), (m,T)\epsilon\{P_c, P_c + lookahead\} \tag{2}$$

The decision is computed by the rule based method. If the global way points between P_c and g_{p0} has any intersection by any static obstacle, then the decision to do a lane change will be initiated. This initiation will provide a exclusive input to local planner to reject the paths that are available on the current lane. Then the LP will compute the paths, that are more effective to do a lane change. The intersection of an actor N_k with state x_k and the global path G_k is given by determining the line intersection formula. To conclude an intersection of obstacle with a path, the obstacle position is converted to a line with the lane boundary information, this acts as stop line fence in which the vehicle should be stopped. Lets assume the stop line fence equation is given by $S \hookrightarrow \{a1x + b1y + c1 = 0\}$ and the line defined by two consecutive path points is given by $P_t \hookrightarrow \{a2x + b2y + c2 = 0\}, t\epsilon\{m,T\}$ The intersection $I = \{I\_x, I\_y\}$ of fence points with global path is computed by

$$(I_x, I_y) = \left(\frac{b1c2 - b2c1}{a1b2 - a2b1}, \frac{c1a2 - c2a1}{a1b2 - a2b1}\right) \tag{3}$$

If the intersection point (I_x, I_y) is successfully obtained, the decision of the BP output will be Stop and the new goal point will be the Intersection point in which the vehicle will need to stop.

B. Local Planner:

The local planner is responsible fro generation of local path along with collision avoidance. The main idea of this module to get the action and local goal provided by the BP and utilize it in best possible way to compute a set of points that leads to the local goal. The path has to efficiently computed and it shouldn't lead to oscillations in the controller . The path points are to be obtained smoothly so that the velocity profiling is gradual on acceleration and deceleration profile. This will help the reduce the longitudinal jerk which is more of high frequency brake-thorttle transition. The pre processing unit of local planner developed will first generate a new set of goal points $g_{\{}p1...n\}$which are perpendicular to the local goal provided by the BP. These goal points are then provided to some solvers which provides a smooth path from the current state x_0 to individual points in the new goal set $g_{\{}p1...n\}$, here n presents the number of paths. This parameter is tunable one, as it depends on the size of road in which the ego vehicle will be traversing from source to destination. This can be made dynamic with the help of simulator API's through which we can get the lane width or road width. The local planner high level diagram is given in Fig 5

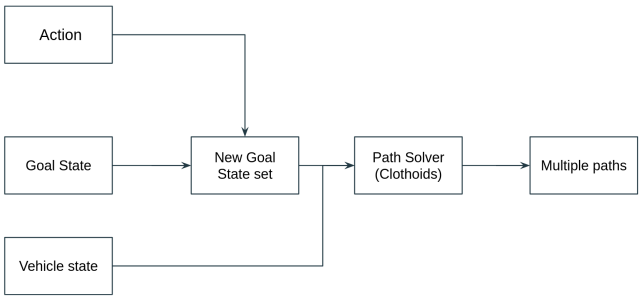

Fig. 5. Local Planner high level block

As seen from the figure, the first step is to get a new goal subset that is normal to travel direction. The new goal set

$$g_{p1..n} = \{g_{pi}^{(x)} + g_o * cos(\psi + \frac{pi}{2}), g_{pi}^{(y)} + g_o * sin(\psi + \frac{pi}{2}) : i\epsilon 0, ..n-1\}$$

where g_o represents the offset between the goal points and ψ represents the goal heading. Again the value g_o is tunable as it depends on the road width, lane width and the max car width which can be accommodated for all the scenarios. Once the new goal set is obtained, it needs to be verified in case the goal point is in drivable lane or not. This can be

done using the simulator API. Once the final goal set after validation is obtained, the local planner job is to generate paths to each of this goal in the new subset. The local planner thats implemented here is clothoid based path generator.

1) Clothoids: There are various choices for the curve generation techniques that can be adopted for a smooth trajectory, each with some pros and some cons. One such example is polynomial splines, where the interpolation conditions are given by a small linear system of equations. If the degree of polynomial is high enough to leave some free parameter, there is also oopputunity to tune the shape of resulting curve. But the disadvantage of this technique is the irrational form of curvature that may have singularities or more generally oscillations in the path. To solve this problem, Clothoids which is plananr curve with the property that the curvature is linear function of arclength is considered. The explicit equation of clothoid in parameteric form is given by

$$x(s) = x_0 + \int_0^s cos\left(\frac{\kappa^{'}\tau^2}{2} + \kappa_0\tau + \Theta_0\right)d\tau$$

$$y(s) = y_0 + \int_0^s sin\left(\frac{\kappa^{'}\tau^2}{2} + \kappa_0\tau + \Theta_0\right)d\tau$$

$$\Theta(s) = \frac{1}{2}\kappa_0^{'}s^2 + \kappa_0 s + \Theta_0$$

$$\kappa(s) = \kappa_0 + \kappa^{'}s$$

where (x_0, y_0) is the base point,Θ_0 is initial angle , κ_0 is initial curvature and $\kappa^{'}$ is sharpness or rate of change of curvature. These integrals are a form of Fresnel integrals and are charetectrized by above mentioned 6 parameters. As mentioned the curvature $\kappa(s)$ is linear function of arclength and the angle $\Theta(s)$ is computed using the above formulas[8].

When constructing a path, several functionals are to be minimized, that include minimum curvature, sharpness, jerk. Solving all these functionals at each point will yield a Hermite problem. When the solution is only for position and angle it is referred to as G^1 Hermite Problem. The spline which solves this problem will have G^1 continuity at each junction point. In some applications, lower complexity of G^1 spline is preferred with one less degree of continuity. But an efficient algorithm for the construction of an interpolating clothoid spline by setting up the nonlinear system of the G^2 continuity conditions is required[9]. The nonlinear system is then simplified, reducing the number of equations to be solved for each arc from 4 to 1, by making use of the results presented in Bertolazzi and Frego [10].

In this work, I have considered the path points to be generated initially in the local coordinate system, as it will be easier for the generation of arcs/splines from the origin. Later this path points are converted to global frame using transformation matrices

2) Collision Avoidance and Path selection: The trajectory generation and selection of the ego vehicle is said to be collision-free, jerk-free and a smooth transition from source to destination. Also there are certain environmental rules that needs to followed when the ego vehicle is travelling on the road. These include maintaining speed limit, avoid collisions, maintain lane center. To achieve all these characteristics for any autonomous motion, we can have a cost function which will compute the costs of individual paths that are obtained from previous step. This cost function takes into account distance of path from lane center, intersection of path with static and dynamic obstacles. To reduce the processing time of this process, it is better to compute the output of collision checking algorithm first and eliminate those paths that are more prone to collision. The result of that will be an array of paths that ensures collision free movement. Now cost function to compute the shortest and fastest path to reach the local goal is to be chosen[11].

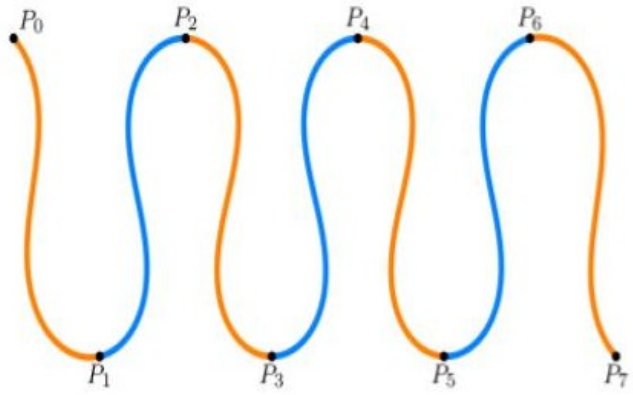

Fig. 6. Multiple clothoid segments that join with G^2 continuity

But in case of dynamic obstacle it is always wiser to predict the movement of obstacle for a certain time period. This prediction for say 3s will give us a fair idea of obstacle's movement which will be helpful for the cost function to ensure a even more safer path. In this work, the obstacle is considered to be a car. The constraints of car in a road is assumed that it will follow the lane center $R_{bound} < S_{obs} < L_{bound}$, it will not cross to other lane for overtaking and it will maintain its lane speed limit. Other random movements by the obstacle car is neglected for simplicity. Now in this work, the prediction of movement of car is obtained from the simulator API from which we can get the lane information of the obstacle car and also can predict the movement of any obstacle with the lane API's provided by the simulator itself. This is very simple for computation of our algorithm and can be obtained as set of points representing the trajectory of the obstacle vehicle. In the scenarios that are executed, I have assumed only one static obstacle and one dynamic obstacle. The path is computed only for dynamic obstacle as it accounts to create a change in maneuver of the car. [12]

Now lets assume the position of static obstacle S represented as $\{S_x, S_y\}$, the path of ego vehicle assumed as p_k and the path of dynamic vehicle as d_k

To check for collision of any other static obstacle for the path, we should obtain the size of obstacle and increase the obstacle with a circle radius of its size. The size of obstacle is obtained from the perception API's of simulator. Now the

static obstacle is inflated using the inflation radius which is max(length,width) of the obstacle. Once the circle is inflated with radius S_r and its center $\{S_x, S_y\}$, we check for collision of that path with the circle. Now we check if any of the path point is inside the obstacle using the crossing number algorithm or even-odd rule algorithm. Now in this algorithm , the circular object is approximated with a polygon of n vertices (n¿6). Now this polygon points are represented as $P_{sx}^{(i)}, P_{sy}^{(i)}$. Now the algorithm to find if the path point $p_k^{(i)}$ is inside the polygon is given below

Algorithm 1: path point inside a obstacle polygon

Input: Polygon points $P_{sx}^{(i)}, P_{sy}^{(i)}$
Path points $p_k^{(i)}$
Result: Collision of path with Obstacle (true/false)
initialization c = false; k = len(P_s) -1
for *i = 0 to len(p_k)* **do**
  for *j = 0 to len(P_s)* **do**
    if $p_{kx}(i)$ == $P_{sx}(j)$ *and* $p_{ky}(i)$ == $P_{sy}(j)$ **then**
      point is a corner of polygon
      return true
    end
    if *($P_{sy}(j) > p_{ky}(i)$) != ($P_{sy}(k) > p_{ky}(i)$)* **then**
      slope = ($p_{kx}(i)$ - $P_{sx}(j)$) *($P_{sy}(k)$- $P_{sy}(j)$) - ($p_{ky}(i)$ - $P_{sy}(j)$) *($P_{sx}(k)$- $P_{sx}(j)$)
      if *slope ==0* **then**
        point is on boundary
        return true
      end
      if *slope < 0 != ($P_{sy}(k) < P_{sy}(j)$)* **then**
        c = not c
      end
    end
    return c
  end
end

The algorithm is also valid for dynamic obstacles but with minor changes. The trajectory of dynamic obstacle is predicted for 3s. Now lets assume the trajectory of the dynamic obstacle d_k. The dynamic obstacle point is inflated for the car width for the whole length of the path . This creates a rectangle sized polygon with length being the prediction distance of path and width being the car width. Now the same Algorithm 1 is applied to check the path generated is colliding with the predicted path or not.

In this way each of the path is being validated as collision free and then all the paths are given to the cost computation function. Paths with "Collision free" tag is one of the important criteria for the cost computation. The cost computation algorithm uses the last path point and the local goal near the lane center g_{p0} from equation 2.

$$C_x^{(i)} = ||g_{p0} - p_x^{(i)} * ||; i\epsilon\{0, 1..n\}$$

given $p_x^{(i)}$ is collision free , * represents the last path point , n represents the number of paths. Now out of these paths and the cost computed one final path is chosen and sent to

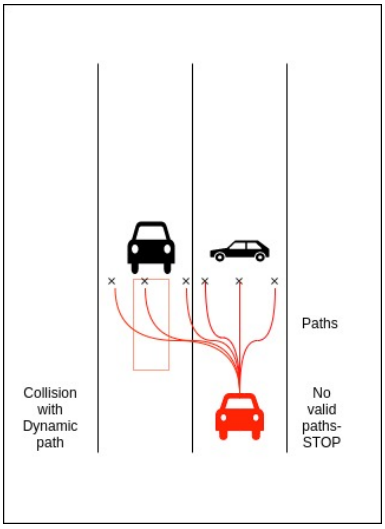

Fig. 7. Collision Avoidance with dynamic obstacle

velocity profiler. The path being chosen has the form with the minimum cost to destination and without collision.

$$P_x = p_x^{(i)} \tag{4}$$

$$subject\ to\ \min(C_x^{(i)}) \tag{5}$$

C. Velocity Profiling

The velocity profiling component is one major algorithm to smoothen the velocity of the vehicle. The path generated may have some curvatures where the vehicle might have to slow down. But in this work, a constant acceleration model is considered to accelerate and decelerate profile. The focus of this section is to use an efficient algorithm to append speed information subject to certain speed constraints. The reference path P_k consists of a sequence of spatial states, such as position, heading, curvature and arc-length. Several dynamic constraints will be imposed on the speed profile V, which is a sequence of scalar speed values. Algorithm 2 iteratively modifies the speed profile taking all constraints into account at every cycle until the speed profile no longer changes. To enforce each constraint, the speed of the current point V_f is adjusted to to meet the constraint values. is adjusted to meet the constraint values . The goal then is to adjust the current speed point v_f such that the corresponding interpolated jerk is bounded by its max limits.[13].

D. 2D Controller:

For path tracking, a lateral controller is used together with the longitudinal controller to control the vehicle. The longitudinal controller to control the acceleration pedal action which contributes to the longitudinal motion. The lateral controller to control the steering of the vehicle which helps the lateral motion of the ego car. . Indeed, in the vehicle motion several longitudinal and lateral couplings arise due to Kinematic and dynamic coupling of the longitudinal and lateral motions due to the yaw motion caused by the wheels steering. [14]

The algorithm chosen for Lateral controller is Pure pursuit . Pure Pursuit is a well-known algorithm for following a given

Algorithm 2: Velocity Profiling

Input: P_k path , Set Speed S, Ego state {x,y,ψ, v} , a_{max}

Result: Trajectory Points with smooth velocity

Function `Acceleration Profile()`**:**

- Fn_{start}
- $a_d = (S^2 - v^2)/2 * a_{max}$ // acceleration distance
- d=0
- ind =0
- **while** $d < a_d$ **do**
  - d += $|P_k^*$ - $P_k^* + 1|$
  - ind++
- **end**
- **for** *i = 0 to ind* **do**
  - di = $|P_k^*$ - $P_k^* + 1|$
  - $vf = v^2 + 2a_{max}di$
  - vf = saturate(S)
  - $P_k(v) = vf$
- **end**

Function `Deceleration Profile()`**:**

- Fn_{start}
- $b_d = (S^2 - v^2)/2 * -a_{max}$ // brake distance
- d=0
- ind =0
- **while** $d < b_d$ **do**
  - d += $|P_k^*$ - $P_k^* + 1|$
  - ind++
- **end**
- **for** *i = 0 to ind* **do**
  - di = $|P_k*$ - $P_k * +1|$
  - $vf = v^2 - 2a_{max}di$
  - vf = clamp(S)
  - $P_k(v) = vf$
- **end**

Function `Compute velocity Profile()`**:**

- Fn_{start}
- **if** *stop(c)* **then**
  - Deceleration_Profile()
- **else**
  - Acceleration_Profile()
- **end**
- return $V_k = P_k^*(v)$

path. This algorithm calculates the angular velocity to reach a target point using the current position and linear velocity of a robot and the curvature passing the target point. This algorithm has been used for many years at a robotics institute, for example Massachusetts Institute of Technology (MIT). PID controller is used for the longitudinal motion. It computes the error in velocity and based on that data it computes to accelerate and decelerate the vehicle[15]. PID is very straight forward action to control the longitudinal motion. But the simplest way to make the follow the trajectory is to use the lookahead point from the trajectory. A lookahead distance or time is configured accordingly to the vehicle speed and a point corresponding to obtained trajectory is chosen[16]

$$l_k = \{d_x, d_y, d_\psi, d_v\}$$

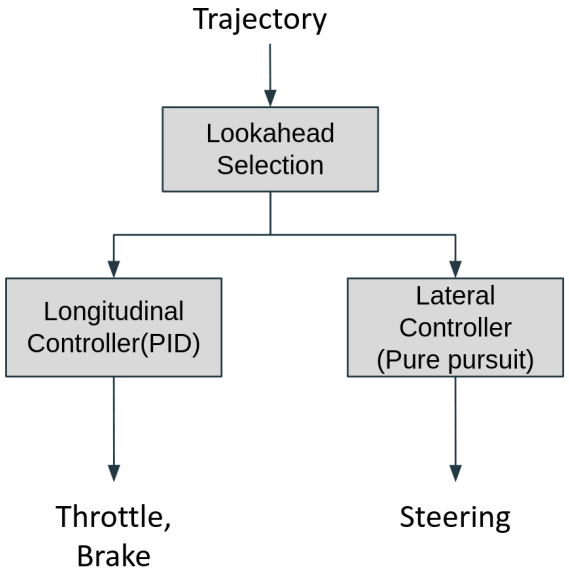

Fig. 8. 2D Controller block

with d_x, d_y, d_ψ, d_v being the desired position, yaw and velocity. Also the controller feedback is the ego state obtained from the simulator API which constitutes the ego state

$$x_0 = \{x, y, \psi, v\}$$

The PID controller algorithm is as folows,

$$v_e = d_v - v$$

$$o_{pid} = K_p v_e + K_i \int v_e dt + K_d \frac{dv_e}{dt}$$

The above term is then clipped with saturation limit of the accelaration and deceleration.To avoid the windup effects, a windup guard is added to the integral term. This saturated the integral term with the windup guard.

Pure pursuit action is tuned to be very smooth for lateral movements. This algorithm expects the lookahead point in vehicle coordinate frame for calculation. Now the same state in equations below represent in vehicle frame $x_{oD} = \{x_D, y_D, \psi_D, v_D\}$

$$\lambda = \arctan(y_D/x_D)$$

here λ represents the angle between vehicle longitudinal axis and line joining P_v and P_D. The length of segment $P_V P_D$ is equal to $\sqrt{x_D^2 + y_D^2}$ and also to $2Rsin(\lambda)$. So that circular arc radius is given by

$$R = \frac{\sqrt{x_D^2 + y_D^2}}{2Rsin(\lambda)}$$

The kinematic steering angle

$$\delta = \arctan(L/R)$$

Now that we have the steering δ and throttle o_{pid}, we can pass on the control commands to the simualtion using the carla API's

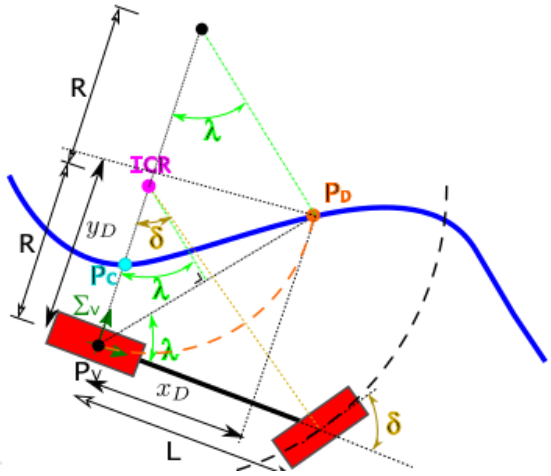

Fig. 9. Pure pursuit action

III. CARLA API INTEGRATION

CARLA is an open-source autonomous driving simulator. It was built from scratch to serve as a modular and flexible API to address a range of tasks involved in the problem of autonomous driving. One of the main goals of CARLA is to help democratize autonomous driving R&D, serving as a tool that can be easily accessed and customized by users. CARLA is grounded on Unreal Engine to run the simulation and uses the OpenDRIVE standard to define roads and urban settings. Control over the simulation is granted through an API handled in Python that is constantly growing as the project does. The CARLA simulator consists of a scalable client-server architecture. The server is responsible for everything related with the simulation itself: sensor rendering, computation of physics, updates on the world-state and its actors and much more. As it aims for realistic results, the best fit would be running the server with a dedicated GPU, especially when dealing with machine learning. The client side consists of a sum of client modules controlling the logic of actors on scene and setting world conditions. This is achieved by leveraging the CARLA API (in Python or C++), a layer that mediates between server and client that is constantly evolving to provide new functionalities. That summarizes the basic structure of the simulator. Understanding CARLA though is much more than that, as many different features and elements coexist within it. More information on carla can be found in their documentation[17].

A. API's used in this Project:

As mentioned above, the first stop to access the CARLA environment is to initiate the server. Carla with its build has given the Server intiating process with the

```
./CarlaUE4.sh
```

This will initiate the server. And we can use this environment by adding any number of actors, sensors and other parameters in the environment. This is the intial point to access any feature provided by CARLA. Once the server is initiated, we need to connect the client with Carla server and can get all the environment settings. These settings include the map that the current environment is operating, actors in that environment and attributes of those actors. To connect to the client, we need to use this line of code

```
client = carla.Client('localhost', 2000)
client.set_timeout(10.0)
```

Now to get the location of any actor, say our own ego vehicle, we can use the API's which can easily replace the localization module. This can also be used as perception information of other actors than ego vehicle. In carla actor is anything that plays a role in the simulation and can be moved around, examples of actors are vehicles, pedestrians, and sensors . To get the information of actor like position, orientation(together as transform), velocity, acceleration

```
Actor.get_transform()
Actor.get_velocity()
Actor.get_acceleration()
```

The bounding box of any actorcan be again obtained by API where we can get the width height and length of any actor. It is embedded in the actor class by default. It can be accessed by

```
Actor.bounding_box.extent.x // length
Actor.bounding_box.extent.y //width
Actor.bounding_box.extent.z //height
```

One of the important feature from Carla API is obtaining map related information. This map information include, location of infrastructure data like traffic lights, signals etc. Apart from that carla MAp is embedded with waypoints which are 3D directed point that stores information about the road definition that OpenDRIVE provides. To obtain the map object we need to have

```
world = client.get_world()
map = world.get_map()
```

This map object can be further utilized to obtain any waypoint that is embedded as map point in the simulator. Each of these waypoint represent part of a road/lane. So it has its position and orientation(transform), lane width at that location , lane marking information, flags representing the lane change availability based on traffic rules. To get a waypoint inforamtion on the map,

```
wp = map.get_waypoint(carla.Location)
carla.Waypoint =  // Represents the waypoint
carla.transform.position.x
carla.transform.position.y
carla.transform.position.z
carla.transform.rotation.x
carla.transform.rotation.y
carla.transform.rotation.z

wp.lane_width // represents the lane width
                where the waypoint is located
wp.lane_type // represents the type of lane
```

Now with this information , we can get all the lane related parameters that are required for the algorithms in previous section. Finally , to control any actor (say ego vehicle) with the velocity commands like throttle and steering which is obtained from 2D controller, we can use the VehicleControl class from the API list. This class provides interfaces steering, acceleration, brake, handbrake, and reverse gear. But I have used only brake, acceleration and sttering in this project.

```
control = carla.VehicleControl()
control.steer = steer_value
control.throttle = throttle_value
control.brake = brake_value
control.hand_brake = false
control.reverse = false
```

Now this vehicle control structure can be applied to any actor in the environment using

```
Actor.apply_control(control)
```

Also to put an actor in autopilot mode, carla provides its internal waypoint following API which helps the actor follow the road waypoints without any obstacle.The api for autopilot mode is

```
Actor.set_autopilot(true)
```

This API is specifically used in STOP_AND_GO scenario where there is a dynamic obstacle in the environment which passes by in the next lane

IV. Conclusion

A. Achievements

The aim of this project is to successfully execute autonomous motion planning for different scenarios in CARLA simulation environment. This motion planning objective is achieved with good accurate results with help of Carla's Perception API inputs and the algorithms used are described in detail in section II. The scenarios that are executed include LANE_FOLLOW, LANE_CHANGE and STOP_AND_GO scenarios. Clothoid based local planning algorithm generated smoother paths with continuity so as to minimize the jerks while in motion. The velocity planning algorithm which involved equations of motion is responsible for the generation of ramp like velocity profile of a path. The control action for these scenarios was quite smooth with the stable lateral and longitudinal controllers operating individually.

There was a separate GUI built to visualize the generation of paths and collision detection modules. This visualization helps in real time tracking and debugging of the motion planning algorithm. The GUI built was using a custom made python library which is optimized for high data operations. A sample of the live simulation execution and the live plotter is given in fig below

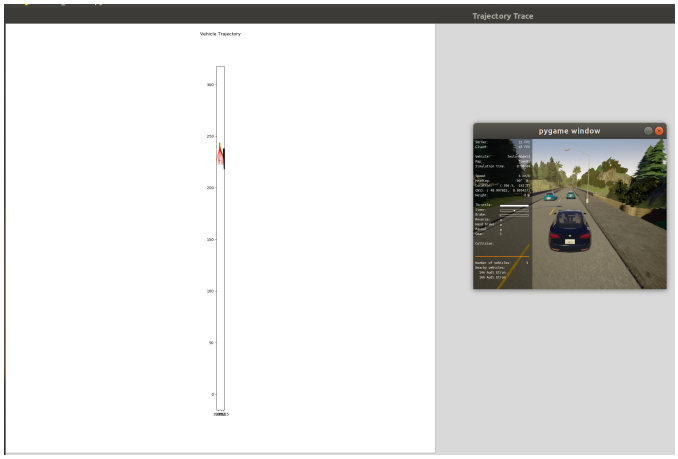

Fig. 10. Live plotter with Carla Simulation engine

B. Future Work

This project can be extended with adding further scenarios that are more complex with intersections, also integration of pedestrians and other complex actors in the environment. Now the project is using the carla API's for the planning algorithms, but in the future these API's can be replaced by the real sensor operations and can be developed as a full Autonomous Driving stack. The Behaviour planning algorithm right now used is a rule-based method which makes decisions based on the simple rules and algebraic formulas. But there are complex neural network algorithms that can be effectively used to get the decision that is more robust in all the scenarios. Neural network based algorithms can replace rule based algorithms for the fact that they can work in complex scenarios as devising rules for these scenarios will be difficult in future.

References

[1] N. H. T. S. Administration, "Automated vehicles for safety," https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[2] ResearchAndMarkets.com, "Autonomous vehicle market by automation level, by application, by component - global opportunity analysis and industry forecast, 2020-2030," https://www.researchandmarkets.com/reports/5206354.

[3] Statista.com, "06.02.2020 by 2030, one in 10 vehicles will be self-driving globally," https://www.statista.com/press.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[5] C. team, "Article- carla, open-source simulator for autonomous driving research." https://carla.org/.

[6] S. S. e. a. Marko Ilievski, "Article - design space of behaviour planning for autonomous driving," https://arxiv.org/pdf/1908.

[7] G.-T. H. Y.-Z. C. Tzu-Chen Liang, Jing-Sin Liu, "Practical and flexible path planning for car-like mobile robot using maximal-curvature cubic spiral," https://www.researchgate.net/publication/222552389.

[8] F. B. D. F. M. F. Enrico Bertolazzi, Paolo Bevilacqua and L. Palopoli, "Reactive planning for assistive robots," https://ieeexplore.ieee.org/document/8550215.

[9] L. P. m. F. Paolo Bevilacqua, Daniele Fontanelli, "Reactive planning for assistive robots," https://ieeexplore.ieee.org/document/8263556.
[10] F. M. Bertolazzi E, "G1 fitting with clothoids. math methods appl sci , 2015," https://onlinelibrary.wiley.com/doi/abs/.
[11] e. a. Laurene Claussmann, "A review of motion planning for highway autonomous driving," https://www.researchgate.net/publication/333124691.
[12] A. F. M. Zahraa Y. Ibrahi, Abdulmuttalib T. Rash, "Prediction-based path planning with obstacle avoidance in dynamic target environment," https://www.researchgate.net/publication/313679916.
[13] T. Gu, "Doctoral thesis- improved trajectory planning for on-road self-driving vehicles via combined graph search, optimization topology analysis," https://www.tianyugu.com/uploads/5/9/4/0/59403035/thesis.pdf.
[14] V. G. Carlos Massera Filho, Denis F. Wolf, "Longitudinal and lateral control for autonomous ground vehicles," https://ieeexplore.ieee.org/document/6856431.
[15] H. O. et al, "Pure pursuit revisited: Field testing of autonomous vehicles in urban areas," https://www.researchgate.net/publication/312272304.
[16] R. A. et al, "Longitudinal control for automated vehicle guidance," https://www.sciencedirect.com/science/article/pii/S.
[17] "Carla documentation," https://carla.readthedocs.io/.