Interceptor 4.0

Project Report

Team Delamain

Zhongheng Li, Nicolas Morant, Huayu Tsu,

DGMD E-17, Harvard University, Spring 2021

May 10th, 2021

I. DESCRIPTION & GOAL OF THE PROJECT

We wanted to build a level 4 autonomous law enforcement vehicle, Interceptor 4.0, that patrols neighborhoods and when needed chases vehicles. The challenges Interceptor 4.0 would face are as follows: lane detection and tracking, object detection and avoidance, GPS navigation, self-parking, specific vehicle tracking, and car chasing. In addition, it will need to follow the rules of law and during a pursuit should have the ability to ignore them while maintaining safety for its user and surroundings.

This autonomous vehicle would enable to increase the safety of law enforcement officers and people around them. It will be more reliable for chasing vehicles as no human will be able to run away from our AI. Officers will also be able to multitask in their Interceptor 4.0 while chasing a vehicle, enabling them to coordinate road blocks, spike strip, and more. In addition, the Interceptor 4.0 will facilitate law enforcement agencies to have more coverage without having the need for more personal. Furthermore, Interceptor 4.0 could be equipped with something like the Grappler Police Bumper (Figure 1) in order to safely stop the pursuing vehicle instead of using a dangerous pit maneuver that could affect both parties involved as well as their surroundings.

Figure 1. Grappler Police Bumper

Finally, the Interceptor 4.0 and the features developed could potentially be adapted for other first responders such as Fire Department's and EMT's vehicles.

II. TEAM ORGANIZATION

Our team was composed of three team members: Heng, Nicolas, and Jack. We each had our own list of features and tasks to work on (Figure 2). We decided to develop separately each feature to enable us to take advantage of our own schedule flexibility in order to optimize the speed of the development. In

addition, we scheduled weekly meetings in order to update each other on our advancements and make further decisions accordingly. In the last few weeks of our project, meetings increased at a rate of three times per week in order to accommodate for integration of the features.

Zhongheng (Heng) Li	Nicolas Morant	Huayu (Jack) Tsu
Object Detection	GPS Navigation	Lane Detection & Tracking
Object Avoidance	Self-Parking	Intersection Handling
Car Chasing	PowerPoints & Read.me	System Architecture

Figure 2. Team Members & Roles

III. SYSTEM ARCHITECTURE

We wanted to setup a baseline model that has the following capabilities under one file:

1. Manual control of Ego vehicle.
2. Toggling of Autopilot + our other features
3. Dynamic weather changing for testing
4. Data Collection
5. Debug View

This helped us integrate our own pipelines much easier later on. We cleaned up the codes from CARLA Python API examples to create a base model that fits our unique needs. We faced the difficulty of integrating different features together because each sensor would generate data at different time, we incorporated synchronous mode to unify server and client time so each tick generates one set of data for each sensor. We also wanted to have a real time debugging feedback so we can see what the autopilot sees. We used OpenCV to generate a Debug View (Figure 3) to help facilitate the live feedbacks. The Debug View helped speed up the development quite a bit.

Figure 3. Base_model with Debug View

- **Status: Completed**

The base model worked as intended. It brings all the development tools we needed under one roof and it has definitely helped us with our development.

- **Key Related Files:**
  - base_model3.py – Third iteration of our base model with full capabilities.
  - debug_cam.py – Manages debug view, variables from the Ego's perspective.

III. FEATURES

1. LANE TRACKING

We wanted to develop a lane track that is capable of accurately detecting and tracking lanes under all conditions. We started by using Assignment04's pipeline for lane detection. However, the result was not good because shadow and rain causes false positive. We ended up choosing to build on top of Thomas

Fermi's model because it was the most consistent. The baseline model is Google's EfficientNet-b0. We made several improvements on top of Fermi's model.

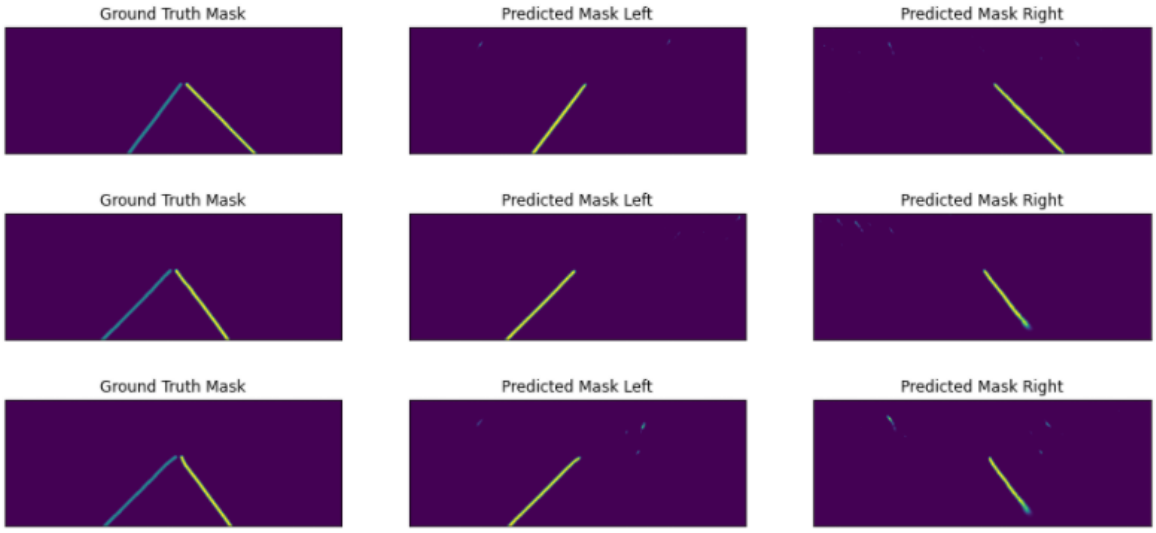

Figure 4. Model Ground Truth vs Predictions

On shadow/weather: by training the model using segmentation images. We were able to resolve the typical shadow instability with regular RGB camera.

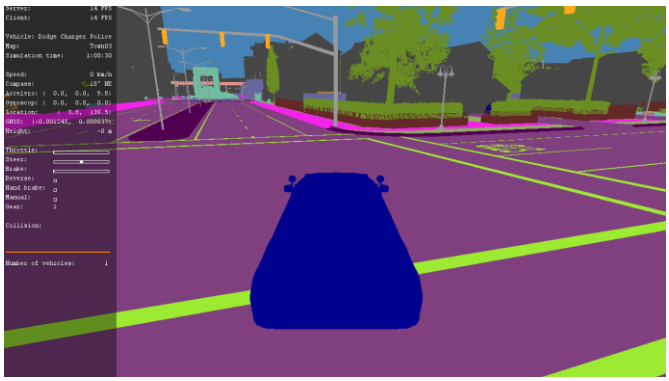

Figure 5. Segmentation Cam

On curvatures: roads that have sharp turns leads to model's instability because it cannot detect the lanes. We increased the field of view (FOV) to allow wider margin of safety. As we increase the FOV, the images become increasingly warped, with a heavy bias on closer pixels to the side. We found the happy balance for us is 120-degree FOV. This improvement alleviated the curvature problem, but doesn't solve it completely. We believe 4 cameras with 360-degree FOV is ideal.

Figure 6. Instability from Sharp Curvature

Regarding efficiency, the original model ran at 6 frames per seconds (FPS) (Figure 7), which was not really road worthy because its time gap was too great between frames at high speed. We were able to improve the efficiency of the algorithm by reducing the image resolution from 1024x512 to 768x288 pixels and size of input image. We reduced the input parameter from 500k pixels to 144k pixels. The improvement boosted the FPS performance to 20 FPS.

FPS	Time Gap(s)	20mph	30mph	65mph	80mph
60	0.02	0.15	0.22	0.48	0.60
30	0.03	0.30	0.45	0.97	1.19
20	0.05	0.45	0.67	1.45	1.79
6	0.17	1.49	2.24	4.84	5.96
1	1.00	8.94	13.41	29.06	35.76

Figure 7. Algorithm Performance Benchmark

The control mechanism was similar to the perspective transformation from "Assignment04". We took the midpoints from predicted lanes and project that on bird eye view as a trajectory. Using np.polyfit, we found the curvature of the trajectory and fed that into PID control. The algorithm for PID control was PurePursuit, and the code was included in Fermi's GitHub repository.

- **Status: Completed – Improvement possible**

The lane track achieved the objective of being deployable up to 45mph. It is able to handle shading related issues with high accuracy. However, the curvature problem is not 100% resolved. Further development will be necessary for improvement.

- **Key Related Files:**
  - lane_track.py - Lane detection + lane control pipeline.
  - best_model_multi_dice_loss.pth – Retrain model with reduced size segmentation images.

2. INTERSECTION

We wanted to develop a feature that can handle intersection navigation. To do that we'll also need detection and control. However, we were not able to train the intersection detection in ML model due to the limitation of Carla's environment. Carla's environment doesn't have the intersection labeling we need to generate sufficient quality dataset.

For detection, our solution is to look at lane detection result as a proxy indication to present of intersection. When the lane detection result becomes unstable with poly.fit errors, then we can assume we are at an intersection. It works well enough in the closed Carla environment, but we can foresee real world edge cases that will break this code.

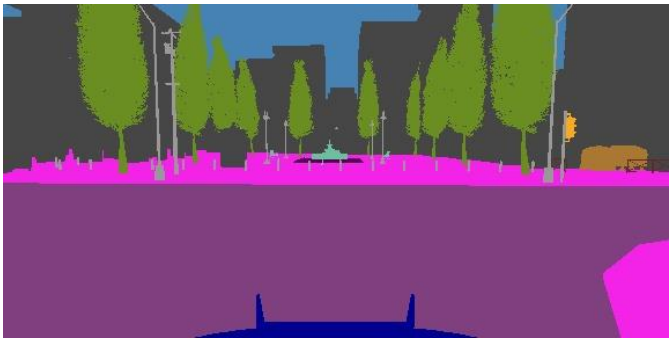

Figure 8. Instability from lack of lanes in intersection

Regarding control, the PID control we are using for intersection is GPS based as opposed to lane detection's visual based control. When the Ego senses it is at an intersection, it activates GPS control that uses PID to move from waypoints to waypoints. This simulates real world GPS waypoints.

- **Status: Completed**

The intersection handling worked as intended for the demo of this class. However, significant improvement is possible by redesigning the autopilot system. Some fundamental features need to developed first before we can improve the intersection to a deployable state.

- **Key Related Files:**
  - intersection.py – intersection detection + intersection control pipeline.

3. GPS NAVIGATION

For the GPS Navigation system, the idea was to find the shortest path possible between two points and return some waypoint locations in order to basically have directions to transmit to our vehicle. First, we had to add a GNSS sensor to the vehicle. Then we listened to the sensor, retrieving the data and storing it into a parquet file in order for us to use the geographic coordinate (latitude, longitude, altitude).

In order to achieve the rest of the navigation system, we had to acquire the topology of each CARLA maps. For the development of this feature, we focused mostly on town two and three. The topology function in CARLA gave us the ability to retrieve tuples of locations with x, y, and z coordinates where there is the start and end of a road, lane, junction, or a lane split and merge. It gave us a list of tuples of connection between each point. In order to understand better how their topology works, we created a script to visualize these points. The topology data (Figure 9) was great and primordial to the development of our own GPS navigation system. However, after analyzing it, we discovered that the majority of the CARLA maps had

Figure 9. Visualizing Town03 Topology Points

some inconstancy regarding their topology. There were some additional points that felt shouldn't be present. This resulted in a few issues in some maps as it seemed that some of the tuples were even sometimes incorrectly connected to each other or their sense of direction was incorrect. Nevertheless, we continued the development of this feature and work with the shortcomings that CARLA was throwing at us.

Next, we had to create a script to acquire and create the edge and node lists in order to use with the python package named "NetworkX". The script basically retrieved the topology tuples, transformed every CARLA location (x, y, z) to geolocations format (latitude, longitude, altitude). Then we created two data frames with the needed information and saved them into parquet files. The node list was composed of the

unique waypoint ID, latitude, longitude, and altitude. The edge list was constituted of the start and end waypoints ID and the distance between them. In other words, any node connected to each other was referenced in the edge list and thanks to having their geolocation we applied the great-circle distance formula to calculate in meters the distance between the two points.

Now that we had all the data needed, we started implementing the pipeline to read our data into NetworkX and be able to produce our first directed graph (Figure 10). The graph itself is a representation of the topology waypoints from CARLA. It ended up looking very similar to the 2D map of each town as we used the latitude and longitude to positioned each node on the graph like an x and y position, but with some differences like the curvature of the roads as these were not part of our data and not needed for the GPS navigation.

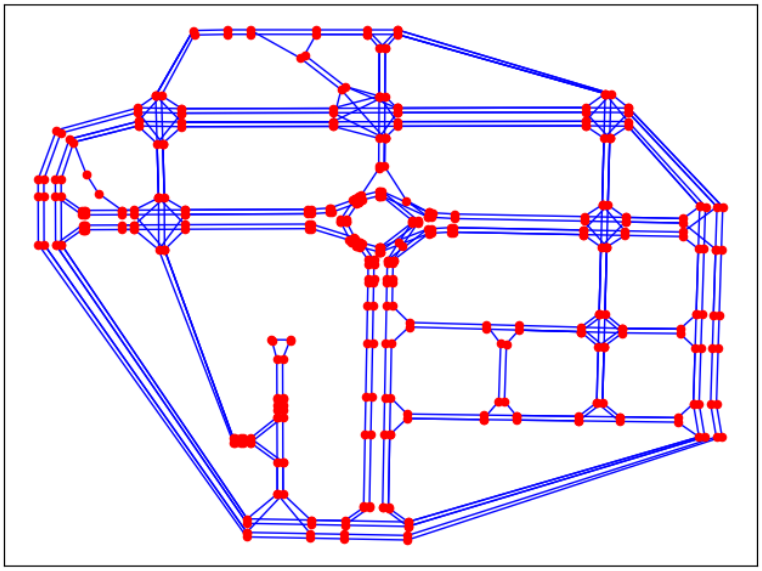

Figure 10. Town03 DiGraph

The next step was to implement the shortest path calculation. We used the Dijkstra's algorithm to search and find the shortest path from the starting location, in our case, the actual location of our vehicle acquired thanks to GNSS sensor and which was stored in a specific parquet file. The use of this algorithm mostly relies on the graph's weights which for us was the distance between each connected node, commonly referred as edges. By acquiring the actual vehicle location and entering a specific destination, we were able to fully create a data frame containing the list of waypoints our vehicle should follow in order to reach its destination. We added this shortest path to the directed graph in order to visualize it on a 2D plan (Figure 11).

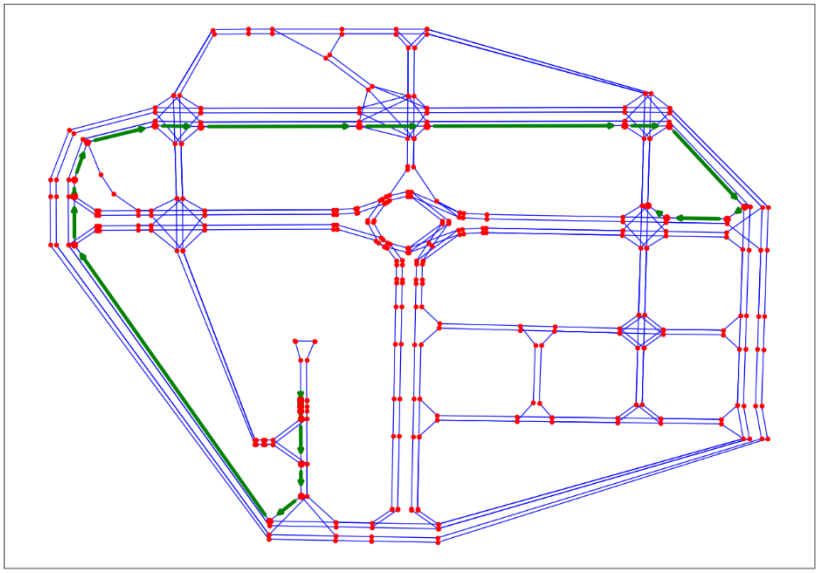

Figure 11. Town03 DiGraph showing the shortest path between two locations

The final step was to finalize our entire pipeline for our GPS navigation system in order to read the data frame containing each waypoints of the shortest path and transmit them one by one to the vehicle. One issue we ran into was that CARLA did not understand what geolocations meant in its environment. So, even thought the simulator had a function to convert a CARLA location to geographic coordinates, it would not understand the scale difference and so we actually had to create our own function to transpose geolocations back to CARLA locations in x, y, z format. Only after doing so, we were able to get the vehicle to move to the shortest path waypoints one by one and reaching the selected destination.

- **Status: Completed**

- **Key Related Files:**
  - topology_edge_and_node.py – Acquire topology and create edge and node lists
  - topology_waypoint_visualizer.py – To visualize on CARLA the topology waypoints
  - spectator_location.py – To get the geolocation of the spectator; useful for getting coordinates of a specific location like setting up a destination for example.
  - road_network_map.py – Saves the matplotlib graph of the road network.
  - shortest_path_digraph_and_df.py – Compute shortest path and directed graph.
  - shortest_path_visualizer.py – To visualize the shortest path in CARLA
  - nav_a2b.py – Full pipeline regrouping some of the functions above and to be use in base model.
  - base_model_nav.py – Base model for GPS navigation for basic demo using teleportation.

4. SELF-PARKING

The development of the self-parking feature was definitely the one that ran into the most roadblocks. We were able to fully implement the parallel and perpendicular parking maneuvers. We basically sent specific coefficients to the vehicle in term of throttle speed, steering angle and breaking power. These maneuvers worked perfectly in closed and predefined environments. Unfortunately, the rest of the feature was not possible to implement due to a few limitations from CARLA. The first one was the lack of ultrasonic sensor. This was a big downfall for us as we expected this simulator to have had the most basic and common sensor that is used in real-life. The use of other sensors like LIDAR and radar was out of the question as it would not have been realistic compared to what is used in real-life, and since we developed all our features with in mine the potential to apply it on a Jetbot, we decided that it was best to leave it like this until CARLA implements the ultrasonic sensor in their simulator. In addition, we discovered that the parking lots that we saw on the CARLA maps were not actually real. These are similar to buildings, meaning that we cannot spawn or drive a vehicle on it. This was a big disappointment for us as we assumed this was possible. Unfortunately, the only work around this constraint would be to fully modified their maps or create our own. However, due to time constraints these were not viable options.

- **Status: Completed maneuvers but reached roadblocks**

The parking maneuvers for perpendicular and parallel parking work flawlessly (see demos), but until CARLA implements an ultrasonic sensor and create parking lots that can actually be used, the development of such a feature reach a major roadblock. However, we did notice some potential additional things that could be used in this type of self-parking feature such as: the detection of empty parking spaces, and the calculation of distance between two parked vehicles to determine if there is enough space to park.

- **Key Related Files:**
  - maneuvres.py – Functions for parallel and perpendicular parking maneuvers.
  - base_model_park.py – Base model for self-parking maneuvers

5. CAR CHASING

If you have seen action movies depicting car chasing scenes, you may notice that car chasing is not just simply one car following another. The leading car will try to shake off the trailing car with sharp turns and drive between cars. And the chasing car will need to track the leading car by adjusting driving angle and speed relative to the leading car to keep up with it. And at the same time the chasing car should also be aware of the traffics on the road to ensure safety of the passengers and their surroundings. Our goal for the

autonomous car chasing is exactly that, to allow the passengers in the chasing car to focus on their mission without worrying about driving.

We defined the subgoals as follow: locking down leading car in sight; adjusting driving angle and speed to get as close as possible; predicting turning angles and optimal driving path in sharp corners; avoiding objects on the road to preserve safety.

For car chasing, we leveraged the proposed car chasing algorithm from a sim2real paper that published by Czech Technical University in Prague. This algorithm only required a single RGB camera to simultaneously predict both the 2D bounding box of the detected leading car and semantic segmentation of the drivable path to determine the relative angle and distance for the trailing car to pursue the leading car. As showed in figure 1, the algorithm subdivided the input image into coarse griddled semantic segmentation to determine the drivable path to approach to the leading car. The following are the pipeline of the car chasing algorithm.

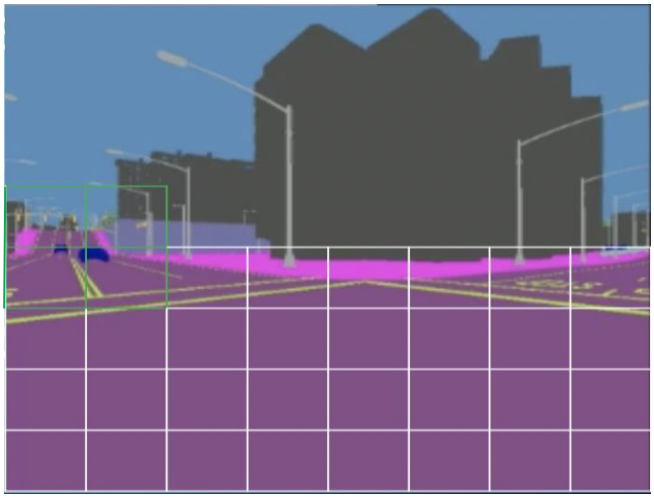

Figure 12. The car chasing algorithm use 8x8 grids to determine the location of the leading car and drivable path. The grids contain the leading car are highlighted in green. And the grids contain over 50% of drivable space, which are purple from the cityscape palette are highlighted in white. And the rest of the grids will be eliminated

The efficient single camera car chasing pipeline was as follow. For perception, we simultaneously used object detection and coarse semantic segmentation of the to determine the relative angle and distance of the leading car on the predicted drivable path. For localization, we estimated the distance and the angle of the chased car from a camera image by turning the estimation into a Perspective-n-Point (PnP) problem that utilized Carla 3D world coordinates of the detected object., the detected 2D bounding box and chasing cars' camera intrinsic camera calibration matrix to determine the rotation and translation between objects. For planning and control, we used short term pure pursuit algorithm to update the target purposing position at each frame to adapt to the change of directions and speed from the leading car. We also used coarse

griddled semantic segmentation-based planning to navigate on the optimal drivable path efficiently. Finally, we used the PID controller to control the throttle and speed based on the predicted angle and distance.

1. Detecting the 3D object bounding box

3. Extrapolation applied for moving averages

2. Frame includes occluded object location

4. Select optimal driving path for sharp corners

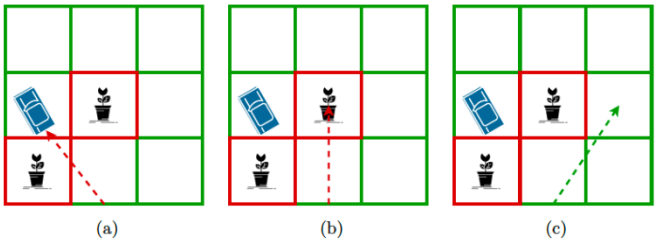

Figure 13. Illustrated the segmentation-based planning. First, the algorithm checks if it is possible to drive directly towards the car (a). This is not possible as the constructed line segment goes through a non-drivable grid cell. Then, the remaining cells on the same row as the detected car are checked (b,c). The algorithm finally proposes a direction represented by a line segment that goes only through drivable surfaces and that is closest to the direction of the chased car (c)

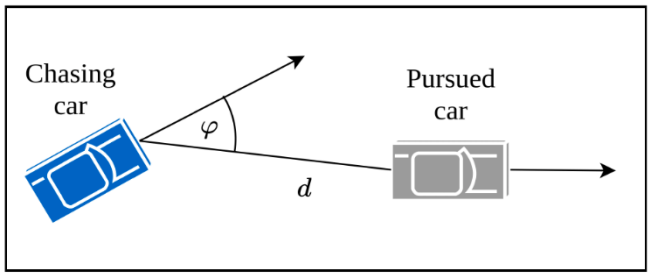

Figure 14. Angle and distance definition:
Distance and angle are estimated using a camera mounted on the chasing car.

- **Status: Completed**
- **Key Related Files:**
  - base_model3_car_chasing.py – Drive as leading car
  - car_chase_demo_v2.py – Run as chasing car

6. OBJECT AVOIDANCE

The original car chasing paper was only focusing on sim2real single car chasing scenario without any concern of other objects on the road. And both the leading car and the chasing car can crash into vehicles and pedestrians on road. This is very unsafe to let drive around with the car chasing algorithm.

Therefore, we looked into trajectory planning as our object avoidance algorithm. In figure 4, we created this simulated environment to test a Frenet trajectory planning framework that was actually used for reinforcement leaning. Vehicle will follow the predetermined waypoints that we generated from our GPS module and obstacles are placed on the path. The locations of the obstacles will need to be on both world coordinates and Frenet coordinates in order to let the Frenet planner's collision detector to recognize them. Obstacles can be randomly placed. But we have not activated auto pilot for them during our experiments.

Figure 15. We created this simulated scenario in CALAR to test our Frenet trajectory planning algorithm.

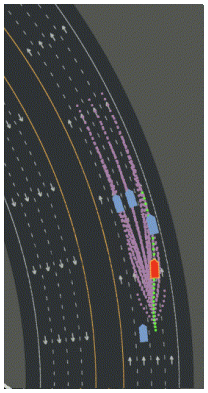

Figure 16. Original RL paper's Frenet trajectory planning demo.

We utilized an OpenAI gym like reinforcement environment and modified their Frenet motion planning module as our Frenet planner. We first turned the locations of the leading, trailing and obstacle vehicles from Cartesian Coordinates to Frenet Coordinates with Frenet parameters like the road's arc (s) and lateral offset on the road shoulder (d) as shown on Figure 16 below. We used our GPS navigation module to create the waypoints to predefine a driving path that Frenet planner required. In every other 2-time steps, the Frenet motion planner generate a set of desired trajectories. The collusion detector will filter

out the trajectories to find the optimal trajectory that will send to the PID controller. The PID controller will run for 2 seconds on the proposed path then the cycle continues.

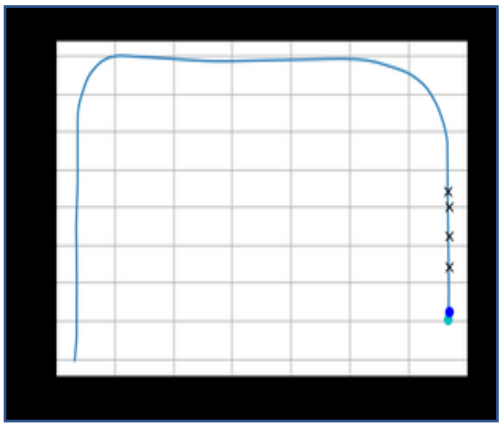

Figure 17. Frenet Trajectory Limitations that required predetermined lengthy driving path

The limitation of Frenet is very similar to regular pure pursuit algorithm that required a predefined route, which is not applicable to car chasing scenarios where the leading car's driving path is nondeterministic.

To resolve this issue, we fine-tuned the Frenet planner to be a dynamic short-term planner that predicted the optimal Frenet path based on the leading car's historical driving path. In a car chasing scenario, the dynamic short-term Frenet planner will be activated if the leading car is 20 meters away from the trailing car for cumulating enough way points for the trailing car's Frenet trajectory. The results can be seemed in Figure. And as you can see from the chasing car's camera the proposing driving path is very close to drive around the obstacles. This is very interesting to see.

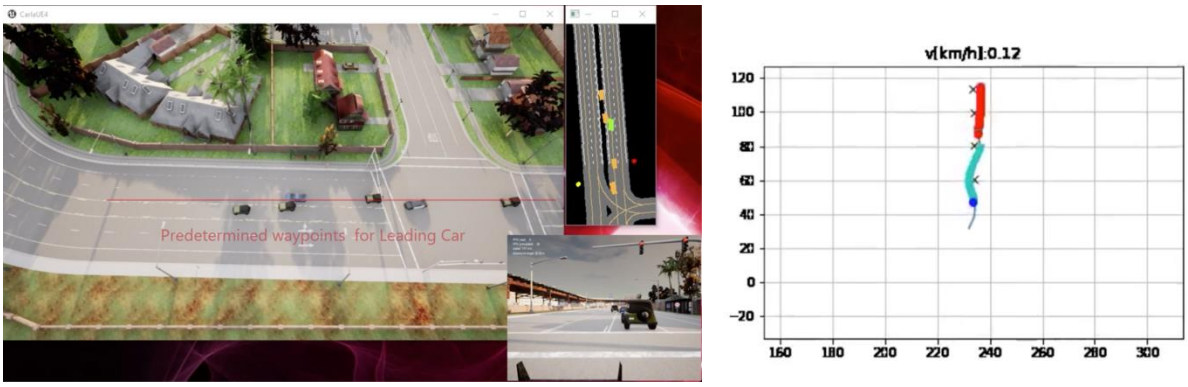

Figure 18. On the left we show both leading and chasing cars have their own Frenet trajectory planner running to avoid the Jeeps on their way. The leading car was following the predetermined waypoints. And the chasing car was following the driving path of the leading vehicle. The animation on the right plot the real-time trajectory for both leading and chasing vehicles.

Regarding the updated dynamic short-term Frenet planner, the pipeline was as follow. For perception, we detect objects on the road, and projecting chasing, leading vehicles and obstacle positions

on Frenet coordinates instead of Cartesian coordinates. We estimated the location of where we could be caching on the leading car to cumulate its driving path. For localization, if the distance between the leading car and trailing was more than 20 meters, we activated dynamic Frenet planning on trailing car to estimate the optimal trajectory based on the driving path of the leading vehicle. Then, proposed trajectories were eliminated during collision detection filtering, and optimal trajectory was be feed to the PID controller. For, planning and control, during short term planning, the optimal trajectory was only to be driven for two seconds via the PID controller.

- **Status: Completed but reached computational limitations**

We have been able to fully implement this feature. However, we were unable to finish its integration in our base model python file due to the dynamic short-term Frenet trajectory planning algorithm that brought computational limitations. We were only able to demo it in a fixed environment that ran on a very low framerate of around 2 to 5 FPS.

- **Key Related Files:**
  - car_chase_demo_v3.py – Run car chasing with dynamic Frenet short-term trajectory planning

There are limitations on our car chasing pipeline. The 3D coordinates of vehicles can be fully observable in CARLA. To realistically predict the 3D object locations, we can use joint monocular 3D vehicle detection and tracking alongside with our PnP approach to project objects on Frenet Space for real-time dynamic object avoidance.

Monocular 3D Vehicle Detection **Applying for perception in CARLA** **Project objects on Frenet Space**

Figure 19. Proposing future work to utilize Monocular 3D vehicle detection to project obstacle vehicle locations on Frenet space in real time while using a single RGB camera only.

Also, there were challenges from the CARLA simulator as well. The frame rate decreases significantly as we increase the complexity of sensor space. This can be due to the known limitations from CARLA, and our limited understanding of the framework. We may take our pipeline out from CARLA and

start following the sim-2-real car chasing code to deploy our pipeline on to a RC car with an NVIDIA Jetson AGX Xavier.

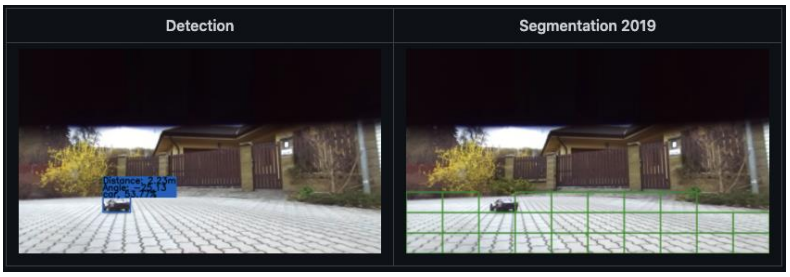

Figure 20. Car chasing demo on RC car from the Autonomous Car Chasing paper

IV. LESSONS LEARNED

Developing an autonomous vehicle with a level 4 of autonomy is much more complicated than we originally anticipated. Something as fundamental as lane tracking can be divided into subproblems, with different challenges to each subproblem.

For example, our lane tracking was divided into detection and control. Detection can be further subdivided into data collection and modeling, to master each subtask alone can take a whole semester of work.

For the GPS Navigation, we rely on the data itself of the road network and any error in it can potentially completely false our shortest path navigation. Also, the shortest path isn't necessarily the fastest one, so we could definitely develop this further by using for example the Google Maps API.

For car chasing, we learned that perception plays a huge role to supply the proposing angles and speed for the motion planner. Utilizing efficient algorithm with lower sensor space complexity during inference can improve performance in both simulation and real environment. Our project demonstrated that car chasing on a single leading car is a solvable problem. However, we still need to have robust algorithm for target identification and re-identification which required one to zero shot object detection capability. Vehicle reidentification is needed especially when the leading car runs off frames for longer than the cached predicted trajectory. If the leading car reentered to the scene with other obstacles cars, our algorithm shall re-identify it and chase it. We also need to improve our car chasing algorithm to handle object avoidance. We attempted to solve this problem with our proposing dynamic Frenet short-term trajectory planner. However, when this algorithm deploys to each agent, the simulation frame rate drops to single digits and

close to 0. Therefore, we will need to find ways to optimize our algorithm. However, able to get the hands-on experience on such trajectory planning algorithms extended our vision to explore more.

Finally, to get each feature to work in a closed environment is one thing, but to deploy it on all situation requires tremendous amount of fine tuning. For example, lane tracking works fine if the road has lanes, and the curvature isn't too sharp. But the moment the vehicle enters intersection, the lane track algorithm fails. Luckily by integrating our features together we were able to resolve this type of cases and made them work. There is definitely so much more we could implement. We are for sure looking forward to learning more about autonomous vehicles, maybe who knows, even work in real-life projects like Tesla, Waymo or others.

V. REFERENCES

- **Our project repository:** https://github.com/heng2j/delamain

- CARLA Documentation v.0.9.1.0: https://carla.readthedocs.io/en/0.9.10/
- Thomas Fermi's Automated Driving: https://github.com/thomasfermi/Algorithms-for-Automated-Driving/discussions/4
- Google Efficient Net: https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet
- NetworkX: https://networkx.org/
- Datacamp – NetworkX Tutorial: https://www.datacamp.com/community/tutorials/networkx-python-graph-tutorial
- Car chasing: https://arxiv.org/pdf/2011.13099.pdf
- Car chasing: https://github.com/MajidMoghadam2006/frenet-trajectory-planning-framework
- Frenet trajectory planning: https://arxiv.org/pdf/2011.13099.pdf
- Frenet trajectory planning: https://github.com/MajidMoghadam2006/frenet-trajectory-planning-framework
- Monocular 3D Vehicle Detection: https://eborboihuc.github.io/Mono-3DT/
- Monocular 3D Vehicle Detection: https://github.com/zhangyanyu0722/Carla_3D_Trackingse