ECE 484 FINAL PROJECT

Devul Nahar
Grainger College of Engineering
University of Illinois, Urbana Champaign
Illinois, United States
danahar2@illinois.edu

Aumkar Renavikar
Grainger College of Engineering
University of Illinois, Urbana Champaign
Illinois, United States
aar8@illinois.edu

Ritvik Goradia
Grainger College of Engineering
University of Illinois, Urbana Champaign
Illinois, United States
goradia3@illinois.edu

***Abstract*— The objective of our project is to develop a robust autonomous controller for the GRAIC Racing Competition that handles both planning and control of a car around a given track. For this project, specifically the F1 Shanghai Track was picked to determine the best controller that can traverse the whole track in the minimum amount of time without colliding into road barriers or obstacles such as other cars and pedestrians on the road. The entire perception model was already provided through CARLA's simulation environment, hence, the objective was purely to determine the best score attainable in two scenarios, where the first one has no obstacles, and the second involves multiple obstacles around the track.**

Keywords*— *GRAIC (Generalized Racing Intelligence Competition), CARLA (Car Learning to Act) Simulation Environment, Principles of Safe Autonomy (ECE484)

I. Introduction

For the GRAIC Racing Competition, our team will be developing a controller that uses the perception model provided through the CARLA simulation environment and aim to develop a robust controller that traverses the Shanghai F1 Track in the minimum amount of time without colliding with the track perimeters or other obstacles such as other cars and pedestrians in the simulation. The results will be quantitatively assessed with the pre-determined benchmark numbers provided for the two scenarios: No Obstacles – 130, With Obstacles - 155. The objective is for the car to complete the track while avoiding collisions and attain a final score below these two threshold values provided. Our group attempted to follow the concept of the racing apex that F1 cars use to attack corners.

II. Approach

A. General Approach for No Obstacles

We have access to up to 20 waypoints on the boundaries of the track ahead of the car. Using these, we calculate the midpoint of the waypoints on either side and use this as the path we want our car to follow. The next step is our controller which directs the car towards the center of the track by adding steer, throttle, and braking.

Using a simple tan function, we can determine the amount of steering required. However, since our coordinate frame is global (i.e. with respect to the track), subtracting this angle by the car's yaw gives us the required relative steering angle.

[Note: 'Yaw' refers to the rotational angle the car makes with respect to the global reference frame.

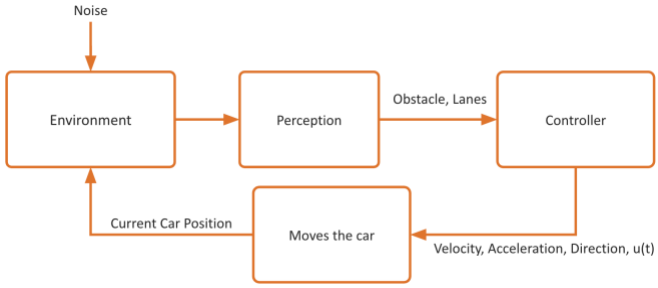

Fig. 1. Control Block Diagram illustrating the relationship between the perception module, the controller and the environment

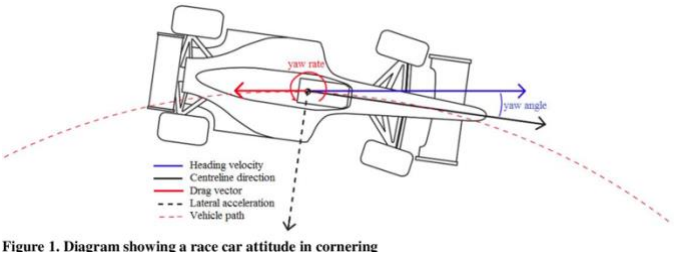

Fig. 2. Illustration of a yaw in an F1 car

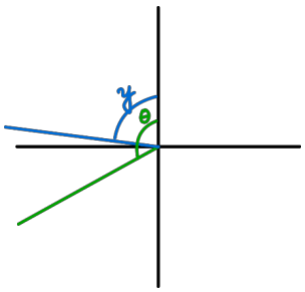

Fig. 3. Illustration of the angle and the yaw in our calculation of theta – yaw resulting in the direction the vehicle needs to move

This approach works well for smooth corners but in sharp corners, this results in oversteering and consequently hitting the barriers. Hence, we needed to alter our approach for turning angles greater than a certain threshold. Simply increasing speed is insufficient as this results in the car slipping, going wide and hitting the barrier on smoother corners. The approach we settled on is when the corner requires a turning angle greater than 1.5 radians, we choose to speed up the car. Although, the approach may seem counter-intuitive at first, this counteracts the oversteer and sets the car on the right track. Thus, by using oversteer, we can nullify any understeer. This adds to the novelty factor of our approach.

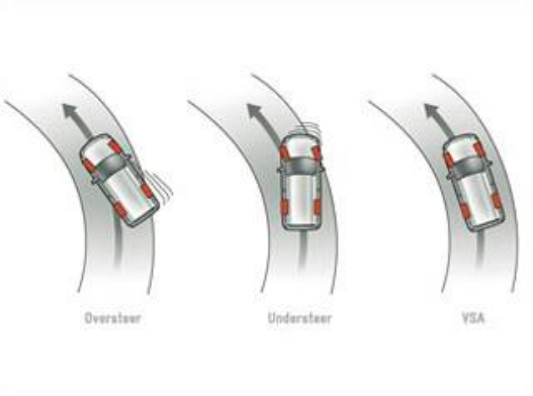

Fig. 4. Oversteering and Understeer

Now that the car can complete one lap, all that's left is tuning the controller to reduce the lap time. This becomes quite tricky because increasing the throttle penalizes at high speeds and increasing the braking penalizes at low speeds. We chose to make the throttle and brake values a cosine and sine function (respectively) of the turning angle. The original score we received on the Shanghai track was nearly 155, but after tuning our controller, we were able to bring that number down to approximately 106.

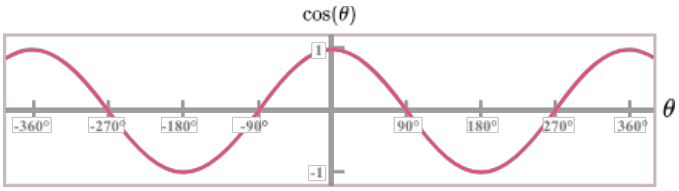

Fig. 5. Graph for Cosine Theta

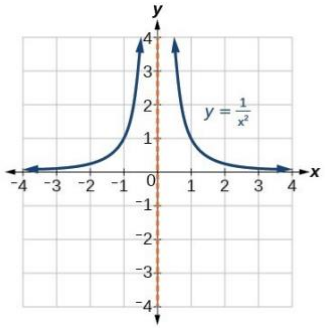

Fig. 6. Graph for 1/x^2

B. *Approach for Obstacle Detection*

To navigate around obstacles, we use the *filtered_obstacles* parameter. This is an array given to us by CARLA simulator, which tells us the obstacles that the car is able to detect through its virtual sensors. We then find the obstacle that is closest to the left boundary (obstacle a) and right boundary (obstacle b) using the 20 waypoints for both the boundaries. Now we move the car right if there is more space on the right side of obstacle a as compared to left side of obstacle b. Similarly, the car moves left if there is more space on the left side of obstacle b as compared to the right side of obstacle a. In doing so the car naturally moves through the space that has the most space without worrying about colliding with the railing or obstacles, as shown in the image below.

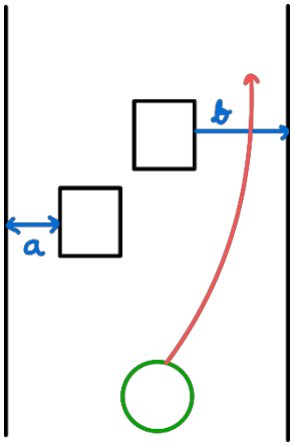

Fig. 7. Figure showing the motion of the car depending on the left or right distances

Once it determines whether it wants to move left or right, the next step is determining the angle, throttle, and the brake. To calculate this, we use the same approach we used when there were no obstacles. To elaborate, if the car is going to the left, we use the obstacle b's position as the left point and the nearest waypoint to obstacle b as the right point. We then find the midpoint between these two points, and use this midpoint to calculate the steer, throttle, and brake. On the other side, when the car wants to avoid obstacle a by turning right, it uses obstacle b's position as the right point and the closest left boundary waypoint as the left point. It then accordingly finds the angle, throttle, and brake the car needs to abide by in order for it successfully avoid the obstacle. If neither of these conditions, it implies that the road is completely blocked, in which case we halt the car, until the obstacles move and give the car enough space to navigate around it.

Additionally, if an object is detected on the road, we first calculate the distance between the object and our car. If the distance between the car and the obstacle is within a specific threshold, we drive the car as if there were no obstacles. The car only moves away from the obstacle if it quite close to it. This allows our car to relatively move at a fast pace and successfully avoid the obstacle just in time.

One edge case with this implementation is when there are multiple obstacles on the road, and our car moves right past one of them. However, if the object is still being detected by the car, it will try curving towards the car that it has just passed. To avoid this from happening, we declared a *passed_objects* array, which is simply an array of 0s and 1s marking whether the obstacles in *filtered_obstacles* have passed or not. If the obstacle has been passed, the car no longer steers according to that object.

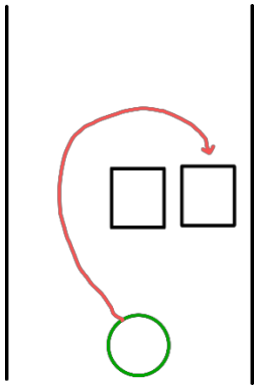

Fig. 8. Figure showing the situation where the object initally passes the closest object, and after passing it, crashes into the next closest object if passed_objects wasn't defined

This approach works really well throughout the track. However, one flaw to this approach is the processing time that the algorithm takes. There are multiple for loops running at each time step in order to do the necessary calculations that allows

the car to adequately navigate the obstacles. This introduces quite a lot of lag in our program, and as soon as there is lag sometimes the car ends up crashing in random spots, generally at points on the track where there are no obstacles. Due to the lag, the car continues the same inputs from the previous time step through CARLA, generally crashing into the road barriers around turns.

Additionally, we explored the effect of relative velocities between the vehicles, which allowed us to detect whether the cars ahead of us are moving such that there is space that is available to follow the car and successfully avoid the stationary obstacles and navigate into open space. We used the following code snippet using the Pythagorean Formula for the velocity: $np.sqrt(vel.x ** 2 + vel.y ** 2 + vel.z ** 2)$

III. Challenges

One of the main challenges we faced in this project was that we had designed our implementation for all 5 of our original team members. However, once we started working, we found out 2 out of our 5 members had dropped the course. We intended to implement a neural network for the perception which would have greatly increased our performance, but we had to scrap it because we were now working with a much smaller team.

Our controller that worked with no obstacles had to be vastly altered to be able to swerve around and through obstacles. Our original controller did not allow enough time for the car to react to moving obstacles, so it had to be significantly slowed down to avoid crashing into obstacles.

By far the biggest issue we faced which working on this project was the processor lag our code introduced. At certain points on the track, we observe significant processor lag and by the time our car receives instruction from our controller, it has moved further in the track and now requires new control outputs. This is evident in the demo video (in appendix) as when there is no lag, our car even passes through more complicated moving objects but when the lag is introduced, it crashes into the barrier on an empty straight. Additionally, we see that at certain points in our track, we see that the car wobbles, which is a result of controller lag as the car is unable to decide the correct action to be taken at that time step.

IV. Analysis & Throughness

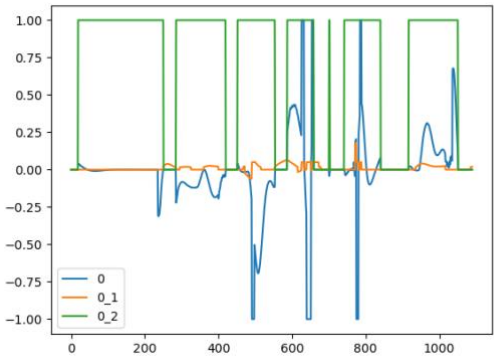

Fig. 9. This graph shows steer (blue), brake (orange), and the times at which an obstacle is detected (green) by the graph per time interval.

In this graph, we can clearly see that whenever we detect any obstacles, the steering and the brake change accordingly. This is evident as the car is only changing any relevant properties inside the green boxes. The green boxes are regions where the car is detecting an obstacle. Additionally, it is important to note that the brake and the steering in CARLA go from -1 to 1. If the steer is -1, then the car has encountered a very sharp turn and is thus taking a maximum turn to the left. On the other side, if the steer is 1, then it has encountered a very sharp turn and is accordingly navigating to the right. Similarly, a brake of 1 means an immediate halt. However, as one can see from the graph, the car never needs to immediately come to a halt. It slowly slows down its speed and navigates through the obstacles.

V. Conclusion

Our controller works well when there are no obstacles and achieves a score far better than the benchmark. When obstacles are introduced, although our algorithm is able to avoid all obstacles, we aren't able to receive a score as the car crashed before the end due to processing lag.

While working on this project one of the main things we learned was that there are several approaches to solve obstacle detection problem. Another approach we wanted to implement was to use vector fields to swerve around obstacles. This approach would return similar results and would not cause as much lag as the waypoint method since it likely doesn't involve as many loops as compared to our implementation of our controller.

VI. Appendix

Without Obstacles: https://youtu.be/FUXU0S6ML-8
With Obstacles: https://youtu.be/3hQFvj3W7iU

VII. Rerefences

GRAIC Challenge 2022:
https://popgri.github.io/Race/
CARLA Simulator:
https://carla.readthedocs.io/en/latest/start_quickstart/#carla-installation