Application of Machine Learning for Perception and Control Algorithms of High-Speed Autonomous Racing Vehicles

THESIS

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

by

Anshul Rajesh Raje
ID No - 2020AATS1024G

Under the supervision of
Prof. Paolo Burgio

BITS, PILANI - K K BIRLA GOA CAMPUS
2023

Acknowledgment

I would like to express my heartfelt gratitude to Prof. Paolo Burgio for providing me with this wonderful opportunity to write this thesis, conduct research, and learn many new things under his supervision. I would like to thank all of the F1/10 team members who have helped me during my time here with them. Last but not least, I would like to thank Dr. Sarang Dhongdi and the administration at BITS Goa for providing me with this once-in-a-lifetime opportunity in my undergraduate program. Without help from any of the people mentioned above, this work would not have been possible.

Anshul Rajesh Raje
December 2023

Certificate

This is to certify that the Thesis entitled "**Application of Machine Learning for Perception and Control Algorithms of High-Speed Autonomous Racing Vehicles**" is submitted by **ANSHUL RAJESH RAJE** ID No. **2020AATS1024G** in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Signature of Supervisor

Name - **Prof. Paolo Burgio**

Date - 09/12/23

Designation - Aggregate Professor

List of Symbols & Abbreviations Used

1. F1/10 - F1TENTH
2. AI - Artificial Intelligence
3. SAM - Segment Anything Model
4. p(x) - Probability of x
5. MPC - Model Predictive Control
6. CPU - Central Processing Unit
7. s_x, s_y - x and y coordinates of the car respectively
8. v - Velocity
9. δ - Steering angle
10. v_δ - Steering angle velocity
11. ψ - Heading
12. a_{long} - Longitudinal acceleration
13. L_{wb} - Wheelbase of the car
14. m - Mass
15. l - Length
16. w - Width
17. h - Height
18. l_f - Length from front axle
19. l_r - Length from rear axle
20. I - Inertia
21. CS_f - Front tire Cornering stiffness
22. CS_r - Rear tire Cornering stiffness
23. IMU - Inertial Measurement Unit

List of Figures

List of Tables

Thesis Abstract

Thesis Title: Application of Machine Learning for Perception and Control Algorithms of High-Speed Autonomous Racing Vehicles

Supervisor: Prof. Paolo Burgio

Semester: First
Year: 2023-24
Name of the Student: Anshul Rajesh Raje
ID No: 2020AATS1024G

Abstract

Autonomous vehicles (AV) are the talk of the town in this day and age. There are various topics to discuss and work on when trying to make the perfect AV (Figure i). This thesis will focus on two critical aspects of AVs: perception and control, specifically in the racing configuration. Perception, for racing, points to opponent car detection. This detection can be achieved by various methods, such as filtering LiDAR data or object detection using a custom-trained model on a dataset. This dataset, however, is quite tedious to create manually. Hence, this thesis introduces a novel approach to synthetic dataset generation for F1/10 cars. When it comes to control algorithms, there are many solutions out there, such as Stanley, Pure Pursuit, and Model Predictive Control (MPC). Various engineers prefer MPC as it uses a model to predict the future state of the vehicle and calculates the optimal solution based on those predictions. This, in turn, creates a huge computational load on the processing unit of the vehicle. Hence, this thesis also introduces a very simple kinematic model-based MPC that reduces computational load while maintaining the desired path tracking.

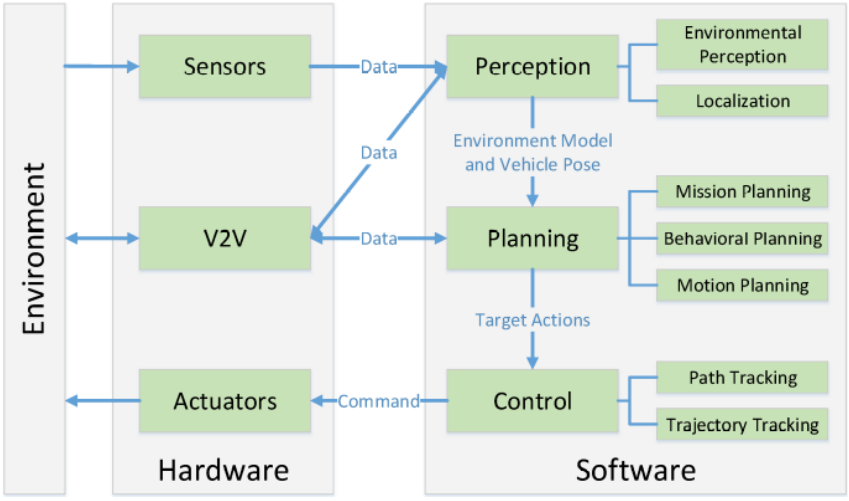


Figure i: Typical architecture of an autonomous vehicle

Table of Contents

Chapter A

F1/10

A.1 What is F1/10?

F1TENTH (F1/10) is an international community of researchers, engineers, and autonomous systems enthusiasts. This community contributes to much of the current research in the field of autonomous driving. F1TENTH is also an international competition that requires cars about 1/10th the scale of an F1 car to race autonomously around a track. The cars race in time attacks and head-to-head trials against rival cars.

A.2 Importance of F1/10

F1TENTH is the perfect entry point for individuals interested in learning about autonomous cars. With a widespread community across various countries, individuals can exchange ideas and collaborate on creating new and exciting solutions for autonomous driving. The F1/10 competition also serves as the perfect testing ground for developing ideas and solutions before deploying them on a full-scale car. Below is a picture of the F1/10 car used by the F1/10 team of HiPeRT Lab, UNIMORE.

Figure A.1: F1/10 racecar

Chapter B

Perception

B.1 Introduction

When it comes to perception in autonomous vehicles, data is very important. Vehicles must input and process a lot of data instantaneously to help them interpret the environment around them. Hence, AI and ML are key technologies when it comes to perception, as they not only help process a lot of information simultaneously but also allow us to make predictions with confidence.

Datasets for F1/10 opponent car detection already exist. However, these datasets are built around only one car model. The car models are not standard in the competition, making the existing datasets less effective. Hence, we want to create our custom dataset, built around various car models and track scenarios. This, in turn, presents several challenges.

Conventionally, to create a dataset, you need to click pictures and then manually annotate them. This would work for a dataset consisting of a few hundred images. But this method is no longer feasible as soon as you want to create a dataset in the size of thousands.

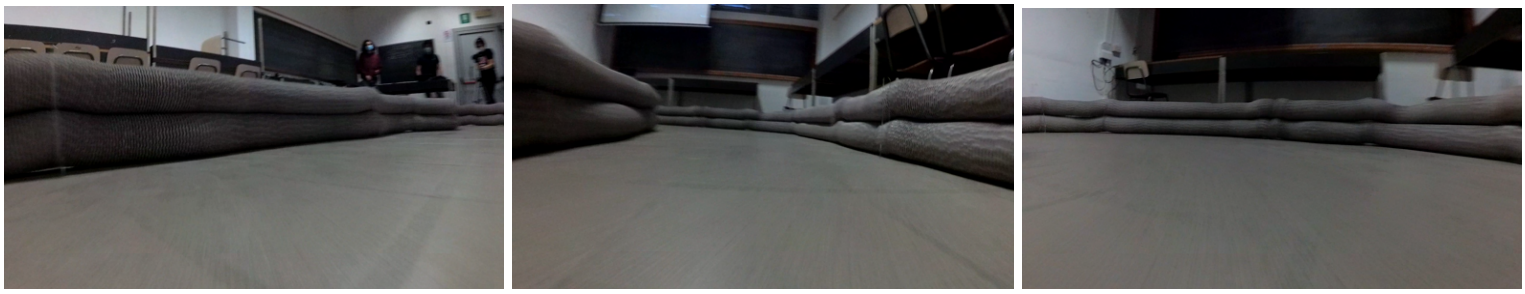
To overcome this, we can use “synthetic datasets.” These datasets are not real-life images but are, in turn, computer-generated and automatically annotated. Using this method, we can remove the need for human labor and efficiently generate thousands of images for the dataset. This method is not specific to F1/10. It can work for any scenario where we have a model of the object (in this case, the car) as well as the environment (in this case, the track).

B.2 Approach

The use of AI models to generate synthetic images is widespread. An example of such a model is DALL-E[1]. This model converts text into images. As helpful as this can be, generating the desired images purely using AI is challenging, as these can have a lot of unexpected noise. Hence, we will use a much simpler, hit-and-trial approach.

This approach was inspired by a blog post by Alex P.[2] on synthetic dataset generation. The approach simply involves pasting renders of the F1/10 car onto real track images with specific rules. We have not used generative AI at any point in the process. Here are the rules with explanations:

1) The background images (track images, terms used interchangeably) are separated into three categories: left, straight, and right, signifying a right turn, a straight track, and a left turn, respectively. It might be counter-intuitive, but this will be important later in the explanation.



Left *Straight* *Right*

Figure B.1: Sample background images for each category

Let's say we want to paste a car render on a "straight" image. We can easily do that by picking a random point on the image and pasting a rendered PNG of our model at that point. There is an obvious flaw in this plan. We only want the car pasted on any part of the track, not the image.

2) Every background image, no matter the category, must have a corresponding mask for the track. We use this mask to ensure the selected point lies on the track. This mask is generated using the Segment Anything Model by Meta AI[3], using the area as the parameter, as the track will usually have the largest area in the image.

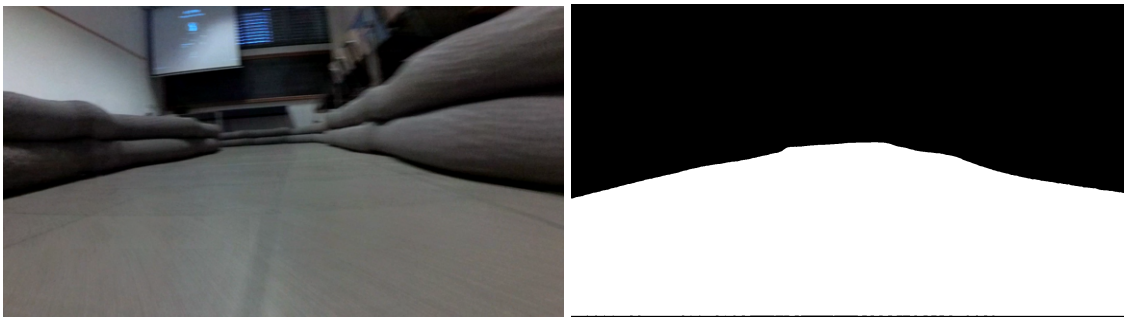


Image *mask*

Figure B.2: Original background image and its corresponding mask

Now, let's talk about the renders of the car. We are not using any form of live rendering. We are simply pasting pre-rendered images of the car. This pre-rendered image is chosen at random. This could work, but many of the images generated would be useless, as the car won't be appropriately oriented. To overcome this, we can split the background image into three columns: Left, middle (or straight, terms used interchangeably), and right. These 'columns' differ from the 'categories' in point 1. Now, we have a general idea of what the car should look like from our perspective. If we

choose the left column, we should be able to see the rear end and some parts of the right side of the car. We should see the car's rear end and left side if we choose the right column. Similarly, if we choose the middle, we should see the car's rear end, with minimal parts of the sides. Example below.

3) The renders are separated into three categories: left, middle, and right. These are separated based on the orientations as explained above. We can randomly choose a render from the corresponding category depending on our selected column. Each render must also have its corresponding mask, regardless of the category.

Left *Middle* *Right*

Figure B.3: Example renders of the car for each category

There are also scenarios where the columns are not always left, middle, and right. This only applied to straight tracks. The same logic cannot be applied to left or right turns. Let's take a left turn, for example. In this case, you are never likely to see the car in the scenario of the renders from the left turn category. If you see a car taking a left turn, you can see the rear end and left side, which are the renders in the right category. Hence, the left category in the background images signifies right turns, and the right category signifies left turns. This is done to match the correct renders to the correct category of background images, which makes life much easier, as we match left to left and right to right.

4) For a background image from the left category, we split the image into left, left, and middle/left columns. Simply put, columns 1 and 2 will always have a left render. Column 3 has a $p(0.5)$ of being from the middle renders and a $p(0.5)$ from the left renders. Similarly, a background image from the right category is split into middle/right, right, and right columns. An example image with column numbers is given below for your reference.

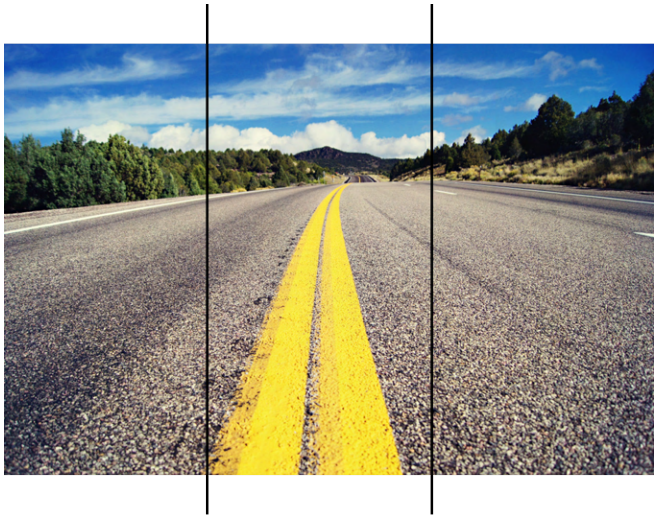


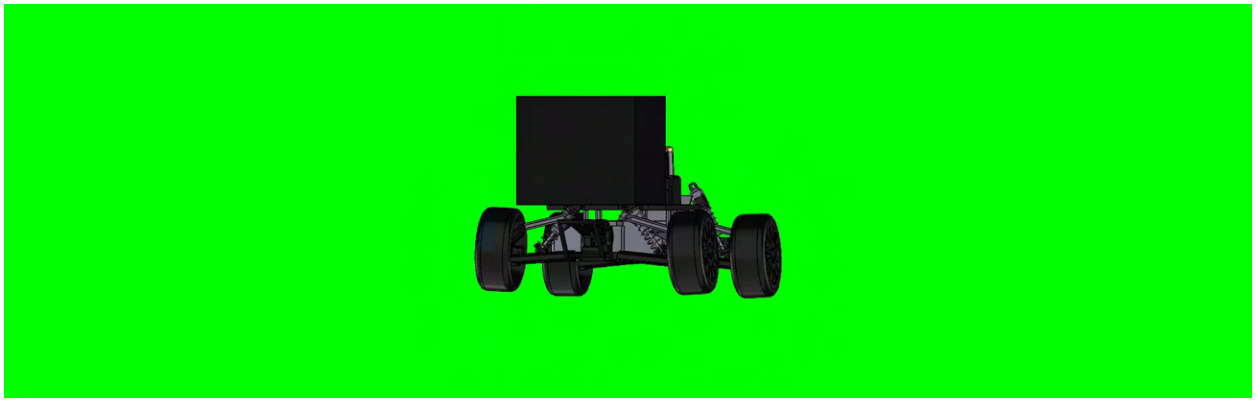
Figure B.4: How to split the image into columns

These four rules are applied to every image that is produced. Since we already know where we are pasting the render (as we chose the point), we can also create the labels for the bounding boxes in YOLO[4] format. These labels are automatically produced by our program in a txt file for every image. Using the images and labels produced, we can train a YOLOv4 model.

B.3 Addressing the flaw

As stated earlier, the advantage of a synthetic dataset is that we can generate very large datasets while skipping the steps of manually clicking and labeling images. However, from the steps mentioned above, there is one noticeable flaw. To create this dataset, we need a lot of existing data in the form of background (track) images and car renders. The background images are simple enough to obtain; just record one lap of the track and extract the individual frames. However, it is not so simple for the car renders. Using Solidworks, we need to reposition the car model and create the render and mask individually every single time. This defeats the purpose of the synthetic dataset, as it introduces a lot of human effort.

To overcome this problem, we can create an animated render of the car rotating on a green screen and individually extract the render and mask from each frame using some simple Python code. This drastically reduces human effort for individually creating every render and helps create renders for different car models much more quickly. An example is shown below:



(a) Frame

(b) Render

(c) Mask

Figure B.5: Steps for render generation - (a) Frame; (b) Render; (c) Mask

B.4 Results

Using the above-mentioned approach, we have a growing dataset of 1500+ images, all synthetically generated. There was no human intervention except for the initial data sorting, such as the background images and renders. Below are examples of the images generated.

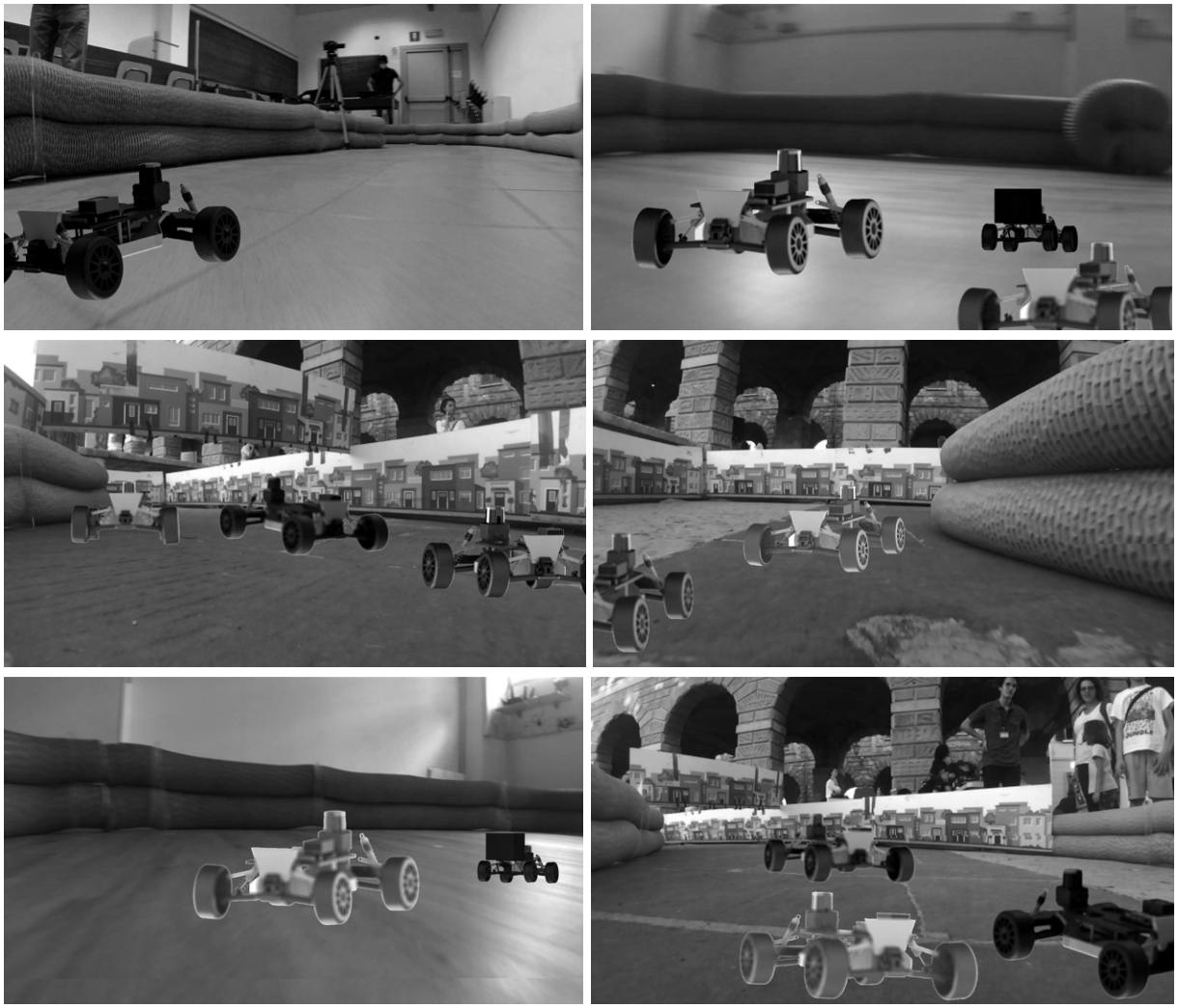


Figure B.6: Generated images

The resulting images were used to train a YOLOv4 model, which was tested on real videos and images captured during practice runs at the lab track and an outdoor track using a ZED[5] camera.

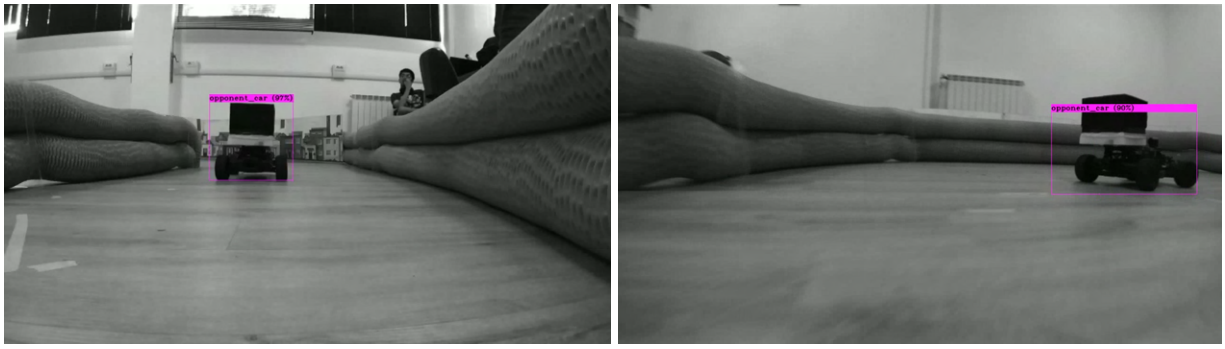


Figure B.7: Predictions

B.5 Future Work

The results shown in the previous section are decent. However, this approach is far from perfect. We have significantly reduced the human effort needed to create a dataset by automating the generation and labeling process. But, some effort is still needed to acquire the initial data, that is, the background images and the car renders. Along with acquiring the data, you must also sort them into specific categories. An improvement to the current solution would be to detect the general curvature of the track in the background image and automatically sort it based on that curvature. Another improvement would be to use the Solidworks API[6] to automatically generate the render of the car model based on the track's curvature and where you want to place the car on the track. These could be the next steps in further development and perfecting this method of synthetic dataset generation for F1/10 cars.

Chapter C

Control

C.1 Introduction

Control is another key aspect of autonomous driving. Without a robust control algorithm, the car won't be able to physically drive anywhere. There are various tested and proven control algorithms out there, including geometric controllers such as Pure Pursuit and Stanley Controller, as well as optimal controllers such as Model Predictive Control (MPC). Engineers usually prefer using optimal controllers, as these controllers use a model of the car to predict the future state of the vehicle and calculate the optimal solution based on those predictions.

However, this introduces a large computational load on the car's CPU. In the context of F1/10, this is a challenge. There is very limited computational power on the car, which has to be distributed to the multiple simultaneous tasks that the car must perform in real-time. This will lead to a delay in the MPC calculations and, hence, poor control.

Hence, in this thesis, we also discuss a very simple kinematic model that the F1/10 car can use for MPC. We use Splines for trajectory planning.

C.2 Vehicle Model

The vehicle model we use is a Kinematic Single-Track Model as described in [7]. The front and rear wheel pairs are considered to be one wheel since roll dynamics are not considered. Since this model also doesn't take tire slip into account, we can assume that the velocity of the car is always in line with the longitudinal direction of the car.

Variables used: s_x and s_y are the x and y coordinates of the car, respectively, velocity v, steering angle δ, steering angle velocity v_δ, the heading ψ, acceleration a_{long}, and the wheelbase l_{wb}. These can be visualized from the figure below from [7].

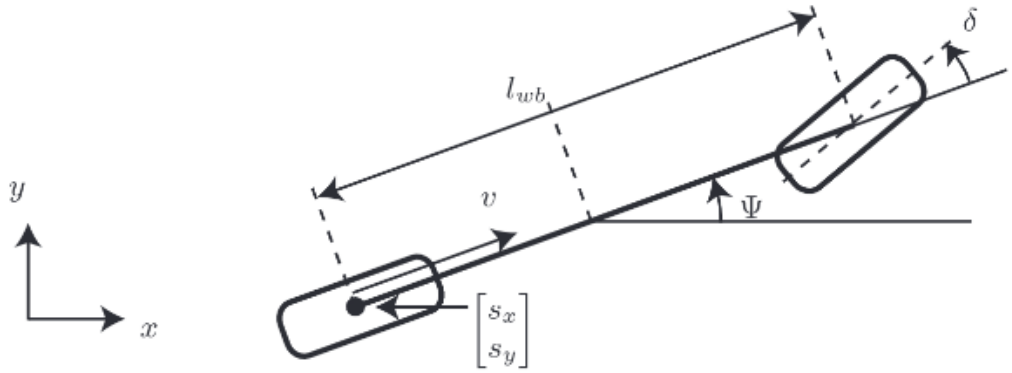


Figure C.1: Kinematic single-track model. [7]

The differential equations describing the model (state variables) along with their state space equations are given as:

$$\dot{s}_x = v\cos(\Psi) \quad \Rightarrow \quad \dot{x}_1 = x_4\cos(x_5),$$
$$\dot{s}_y = v\sin(\Psi) \quad \Rightarrow \quad \dot{x}_2 = x_4\sin(x_5),$$
$$\dot{\delta} = v_\delta \quad \Rightarrow \quad \dot{x}_3 = f_{lim}(x_3, u_1),$$
$$\dot{v} = a_{long} \quad \Rightarrow \quad \dot{x}_4 = f_{lim}(x_4, u_2),$$
$$\dot{\Psi} = \frac{v}{l_{wb}}\tan(\delta) \quad \Rightarrow \quad \dot{x}_5 = \frac{x_4}{l_{wb}}\tan(x_3),$$

where f_{lim} is a function that constraints the inputs within predefined bounds.

The outputs of the MPC model are the steering angle velocity and the acceleration. Therefore, their state space variables are:

$$u_1 = v_\delta$$
$$u_2 = a_{long}$$

C.3 Model Parameters

Name	Symbol	Unit	Value
Mass	m	kg	3.74
Length	l	m	0.58
Width	w	m	0.31
Height	h	m	0.074
Length from front axle	l_f	m	0.15875
Length from rear axle	l_r	m	0.17145
Inertia	I	kgm^{-2}	0.04712
Front tire Cornering stiffness	CS_f	$Nrad^{-1}$	4.718
Rear tire Cornering stiffness	CS_r	$Nrad^{-1}$	5.4562

Table 1: Model Parameters for F1/10 Gym

A lot of these parameters are not used currently, but will be very useful in the implementation of a model that takes tire slip into account.

C.4 Working with Splines

When it comes to path tracking, it is common to move from point to point. So, for example, in autonomous cars, we can move from GPS point to GPS point. This works perfectly fine as long as we have a lot of points available to help make a smooth trajectory. But what happens when we have widely spaced points? This can lead to jittery and uncomfortable path tracking. To help overcome this, we can use splines.

Splines, or more precisely, spline interpolation, is a method of generating curvilinear paths between points. Think of it as connecting dots, except with curves [8].

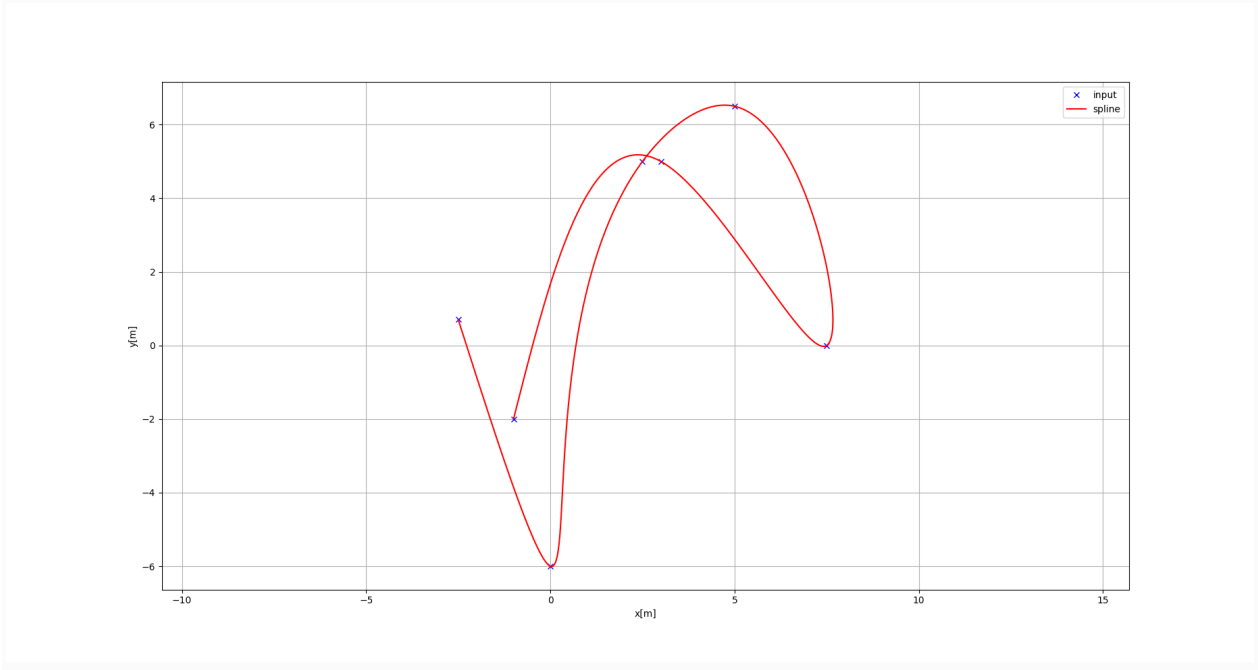


Figure C.2: Example of Spline generated using [9]

The three main variables when it comes to describing a spline path are s, kappa, and n (or d, depending on the publication). s is the distance traveled along the path from the starting point to the end. ‘kappa’ refers to the curvature corresponding to each ‘s’ point. n is the Euclidean distance of the vehicle from the closest s point. If the vehicle is to the left of the track, n is positive, otherwise, it is negative. Let’s say we are at s=0m. s=1m means the point ALONG the path, which is at a distance of 1m from the starting point. This is NOT the Euclidean distance. This s=1m point will have a corresponding kappa or curvature.

This is how we generate the smooth curvilinear path to be followed by the MPC and display the trajectory generated by the MPC prediction horizon.

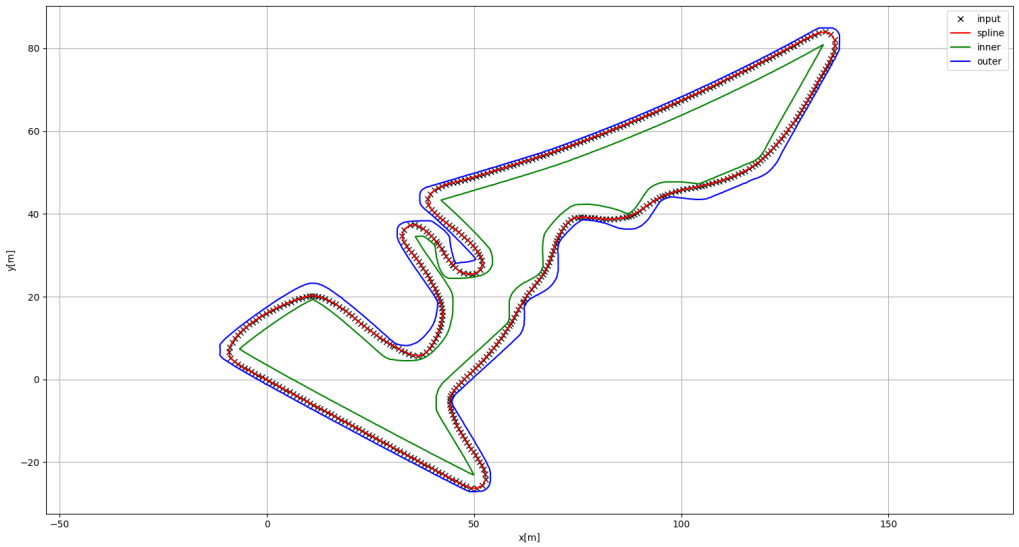
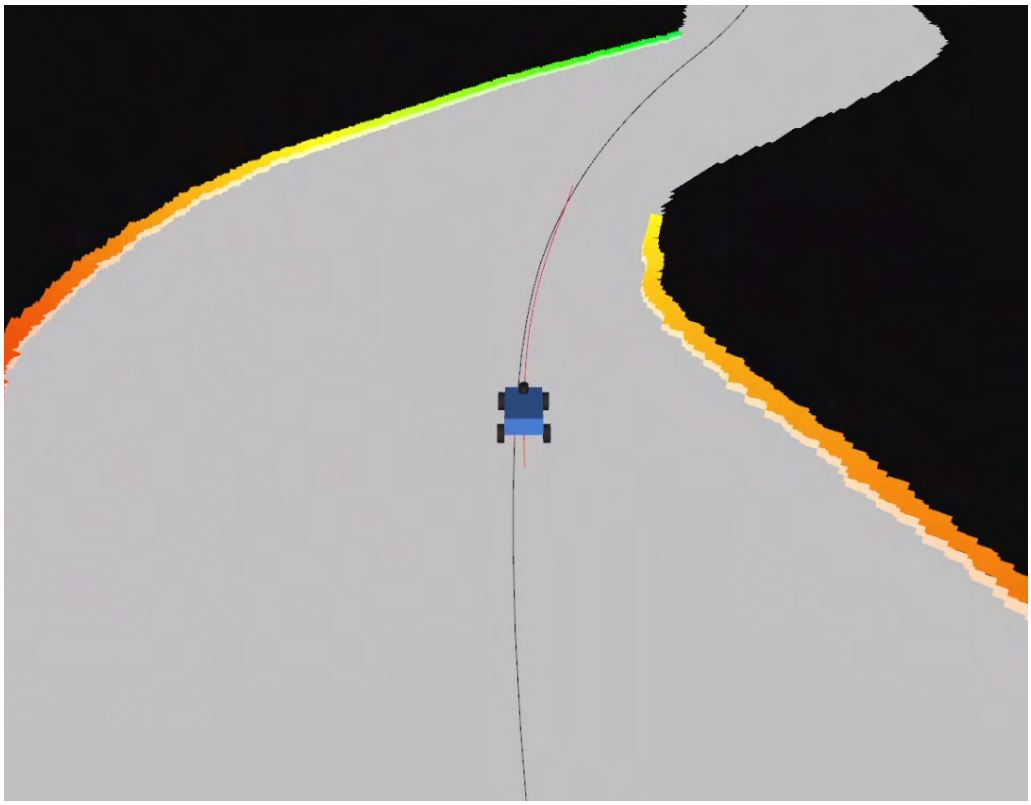


Figure C.3: Path Planning using Splines for Vegas track



Black - Global Path

Red - MPC Prediction Trajectory

Figure C.4: MPC Prediction Trajectory displayed using Splines in F1/10 Gym

C.5 Results

This approach was tested in the F1/10 Gym [10], on the Las Vegas track (Figure C.3) against a previous implementation of MPC by the F1/10 team of HiPeRT Lab. The previous implementation was in Python, whereas the current one is in C++. Let's use the programming languages as the identifiers. The metrics measured were the mean MPC computation time and the mean lateral error across one lap, as well as the lap time.

		MPC (Python)	MPC (C++)
Mean MPC Time	Lap 1	7.58ms	2.34ms
	Lap 2	7.54ms	2.02ms
	Lap 3	7.49ms	1.99ms
Lap Time	Lap 1	104.910s	107.184s
	Lap 2	104.239s	106.547s
	Lap 3	104.287s	106.523s
Mean Lateral Error	Lap 1	7.92cm	4.92cm
	Lap 2	7.89cm	4.57cm
	Lap 3	7.66cm	4.82cm

Table 2: Comparison between previous and current implementation

The reduction in computation time is to be expected, as C++ is much faster than Python. But it is also important to note that the C++ implementation has a much larger prediction horizon than the Python implementation. The former uses a prediction horizon of ~2.5s (63 samples, sampled every 0.04s), whereas the latter uses a prediction horizon of 1s (10 samples, sampled every 0.1s).

The Python implementation has a faster lap time than the C++ implementation. This is due to the fact that it uses a constant speed command of 4.5m/s. The only input to the vehicle from the Python MPC is the steering angle. Sure, this gives a faster lap time, but it is definitely not ideal. The C++ implementation gives two inputs to the vehicle from the MPC, the rate of steering as well as the acceleration. So, the car slows down in the turns, hence losing time, but also giving much smoother control. This is much more realistic, even in full-size vehicles. Hence, the C++ implementation can also handle higher speeds in a straight line, as it will tell the car to slow down in the corners.

The C++ implementation has a better mean lateral error. This is most certainly due to the slower speeds in the corners, allowing the car to take a much more controlled turn. Giving both acceleration and steering angle as input to the car from the MPC was

attempted with the Python implementation with no success. This could be due to the fact that the prediction horizon of the Python implementation was not exactly accurate. I did say that it was 1s. However, it was not sampled by time but space. At every sampling point, the prediction horizon didn't actually predict the state of the car in the future but instead assumed that it would be a certain "lookahead" distance from its current location. For your reference, the lookahead distance was set at 1.5m. So adding this every 0.1s over a duration of 1s results in a constant prediction horizon of 15m, regardless of the actual state of the car. This trick was used by the student who worked on the Python implementation as it could not handle the sampling by time (the reason was not stated). Hence, the C++ implementation is more accurate and closer to traditional MPC, with no tricks involved.

C.6 Future Work

The approach proposed in this thesis worked in the simulation. However, cars race and drive around in the real world. The next step is to definitely test this codestack on the physical F1/10 car and tune it to its limit. After achieving this, we can look at understanding the forces acting on the car and the wheels and try to develop a dynamic model of the car by filtering Inertial Measurement Unit (IMU) data. This dynamic model will certainly outperform the kinematic model, as taking tire slip into account is a major improvement to the model, giving a much more accurate prediction.

Conclusion

In this thesis, we talked about two key aspects of AVs: perception and control. We introduced some basic stepping stones in both of these topics that can help propel work in these fields for not only F1/10 but also AVs in general. For perception, we introduced a novel approach to synthetic dataset generation for opponent detection in F1/10 cars. This has the potential to be applied to everyday self-driving cars as well. For control, we talked about a simple kinematic model for MPC, integrating splines for trajectory generation, that outperforms the previously implemented Python MPC in various metrics. Future improvements that can be worked on in both the perception and control approach have been talked about in their own respective “future work” sub-sections.

References

[1] DALL-E, OpenAI, accessed 6 July 2023, <https://openai.com/dall-e-2>

[2] Alex P. (2022, January 10). How to Create Synthetic Dataset for Computer Vision (Object Detection), Medium, accessed 24 June 2023, <https://medium.com/@alexppppp/how-to-create-synthetic-dataset-for-computer-vision-_object-detection-fd8ab2fa5249>

[3] Segment Anything Model, Meta AI, accessed 11 September 2023, <https://segment-anything.com/>

[4] Kundu, R. (2023, January 17). YOLO: Algorithm for Object Detection Explained [+Examples], V7 Labs, accessed 6 July 2023, <https://www.v7labs.com/blog/yolo-object-detection>

[5] ZED 2, StereoLabs, <https://www.stereolabs.com/products/zed-2>

[6] SolidWorks API, Dassault Systemes, <https://help.solidworks.com/2021/english/SolidWorks/sldworks/c_solidworks_api.htm>

[7] M. Althoff, M. Koschi, and S. Manzinger, "CommonRoad: Composable Benchmarks for Motion Planning on Roads," in Proc. of the IEEE Intelligent Vehicles Symposium, 2017, pp. 719-726.

[8] Trajectory Generation, WPILib, accessed 20 October 2023, <https://docs.wpilib.org/en/stable/docs/software/advanced-controls/trajectories/trajectory-generation.html>

[9] Sakai, A. (2023). PythonRobotics. GitHub. <github.com/AtsushiSakai/PythonRobotics/tree/master/PathPlanning/CubicSpline>

[10] F1/10 Gym (2023). F1TENTH. GitHub. <https://github.com/f1tenth/f1tenth_gym>

[11] Orukpe, P. E. (2012). MODEL PREDICTIVE CONTROL FUNDAMENTALS. *Nigerian Journal of Technology*, *31*(2), 139–148. <https://www.nijotech.com/index.php/nijotech/article/download/6/242>

[12] A. Raji et al., "Motion Planning and Control for Multi Vehicle Autonomous Racing at High Speeds," 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC), Macau, China, 2022, pp. 2775-2782, doi: 10.1109/ITSC55140.2022.9922239.