Model Predictive Control for F1/10 Cars

Kacper Floriański

A thesis submitted for the degree of *Bachelor of Science*

School of Computing

Newcastle University

Newcastle upon Tyne, UK

May 2021

12500 words

Abstract

The popularity of autonomous systems has grown massively in recent years. More and more aspects of our lives are automated by machines, often to the point that we can not imagine living without the improvements they bring into our lives. However, since plenty of these systems can endanger human lives (e.g. self-driving cars), it is crucial to analyse their safety and performance.

In particular, this thesis aims to reflect on the problem of autonomous racing, in which a vehicle must not only traverse the racing track as fast as possible, but also avoid crashing at the same time. More specifically, in this dissertation, three variations of Follow The Gap motion planning algorithm are joint together with Model Predictive Control, to drive a simulation of the F1TENTH racing car through a variety of racing tracks.

Acknowledgments

Firstly, I would like to thank my supervisor, Sergiy Bogomolov, for creating an opportunity to work on such an exciting project. The professional advice and guidance on this thesis, as well as the general academic knowledge shared by Dr. Bogomolov have been invaluable.

Then, I would also like to thank Felix Fiedler, who has kindly helped me understand the technicalities of the *do_mpc* Python tool.

In addition, whole heartedly, I would like to express my sincere gratitude to Paulius Stankaitis, for answering every small and big question I would raise throughout the year. I am extremely appreciative of your patience and the friendly chats we had, and will treat you as a role model if I ever get to supervise others in my work or further studies.

Finally, I believe I owe a large thank you to my parents, who have supported me throughout all of my academia years. Without you, I would have not managed to be where I am.

Declaration

I declare that this dissertation represents my own work except where otherwise stated.

Contents

Chapter 1

Introduction

This chapter describes the motivation behind the dissertation and provides a collection of objectives. Additionally, an outline of the document is provided to describe the structure of the thesis in more details.

1.1 Research motivation

Autonomous systems play an increasingly important role in the modern society [1]. They help to automate several aspects of our every day's lives (for example through self-driving cars), as well as allow performing tasks that are too dangerous or too difficult for humans. Since the malfunctions of the vehicles could endanger human lives, the systems are safety-critical [2], and therefore optimising them and analysing their overall safety is of crucial importance. Consequently, it becomes more and more important to carefully design control algorithms for such machines.

The subtopic of autonomous racing has been chosen because of the high degree of clarity in terms of the achievable goals (to minimise the lap time and to ensure safety), as well as the applicability of racing to important aspects of autonomous driving, such as collision detection and avoidance. Furthermore, while this dissertation does not cover the problem of multi-agent racing, the subject offers a further degree of difficulty and research potential when considering interactions with other

autonomous or manually-driven vehicles (such as in [3]).

To avoid high expenditures of development and prototyping, modelling and simulation environments which can approximate the physics of a vehicle have been chosen. In addition, since different environmental conditions and many unnecessary details could make it difficult to correctly analyse and evaluate the control methods, the simulations are preferred as they provide the desired level of abstraction.

Finally, addressing the reasoning for investigating Model Predictive Control (MPC) and Follow The Gap (FTG), the former is a well-known method modernly used in a variety of industrial and academic setups ([4, 5]). It is therefore well-established in terms of practical applicability and research relevance, making it a good core algorithm. The latter one is a fast-to-implement reactive control method that has previously outperformed other algorithms implemented for the F1TENTH racing vehicle [6]. Contrary to several existing research papers utilising MPC with some sort of precomputed reference trajectory (such as [7]), FTG reacts to the environment in real-time and can quickly navigate unknown areas.

1.2 Objectives

This thesis aims to reflect on the problem of autonomous car racing, where high-performance requirements are intertwined with safety assurance. More intuitively, to win a race, a vehicle must drive as fast as possible to achieve the best time, but also it must not drive too fast or it risks crashing and never finishing the competition. Therefore, the general objective is to investigate control algorithms that navigate a racing track while minimising the traversal time.

More specifically, this research aims to investigate FTG planning algorithms utilising MPC. As a brief introduction, MPC is a well-known method for deriving control inputs based on the control system's estimations for some finite time horizon [8], and FTG is a selected reactive control method capable of finding safe driving directions within some navigable area [9].

To address the aim, the following objectives are considered:

- To develop a collection of algorithms utilising FTG and MPC.
- To evaluate the safety of the algorithms and understand the reasoning behind selected vehicle actions in various racing scenarios.
- To measure the performance and find the factors which result in better race times.
- To visualise the algorithms, for better intuition and understanding of the control methods, especially when describing them to potential readers.
- To create a maintainable, well-documented, and open-source software package that can be used by future researchers (or other individuals and organisations).

1.3 Outline

This thesis is structured in 6 chapters:

Chapter 1 - Introduction

An introduction to the topic, featuring research motivation and a collection of objectives. In this chapter, a rationale behind analysing autonomous control is provided, and the subtopic of car racing is addressed. Then, after explaining the reasoning for choosing FTG and MPC, the aim of this thesis is included, alongside a list of goals.

Chapter 2 - Background

An overview of preliminary knowledge providing a brief introduction to path planning algorithms and MPC. Provides descriptions of theoretical concepts used in this thesis, including mathematical models of the most important components.

Chapter 3 - Algorithms

A collection of thoroughly explained algorithms and ideas used within the dissertation. More specifically, this chapter consists of the vehicle model, the control methods, and the lap time measurement strategy. At first, the Single Track vehicle model is presented. Then, a description

of how MPC is used in this thesis is provided, explaining how following some target reference point is achieved using a cost function based on Euclidean distance between two points. Next, three variants of FTG algorithms are included, featuring both visual and textual descriptions. Finally, an explanation of the technicalities of how lap times are measured is presented.

Chapter 4 - Technical Stack

A detailed description of the technical tools and implementation details, as well as the framework developed as part of this thesis. Provides an overview of Robotic Operating System (ROS) and the F1TENTH simulator based on it, explains the structure of the source code hosted on GitHub, and delivers a short presentation showcasing basing concepts of the Python *do_mpc* library (which implements MPC).

Chapter 5 - Results and Evaluation

A section providing experimentation results and the associated evaluation. At first, the tuning parameters are described, highlighting how changing them modifies the behaviour of the algorithms. Next, an initial comparison of the control methods using arbitrary tuning is provided, discussing a major issue with the Farthest Point FTG algorithm. Subsequently, experiments concerning avoiding obstacles are undergone, alongside an attempt to describe safety regions with systems of inequalities. Finally, an in-depth tuning of the algorithms is performed, evaluating the results and summing up the entire chapter.

Chapter 6 - Conclusions

A brief summary of what this thesis has covered, including potential future work.

Chapter 2

Background

This chapter provides optional, preliminary knowledge helpful in understanding the technical chapters of this thesis. In particular, brief overview of path planning and MPC is provided.

2.1 Path planning

Motion planning is a computational approach to guide an object from the source to its destination [10]. The calculated trajectories must only go through the safe regions of the path and avoid any obstructions. In addition, the path planning problems often require optimisation to minimise some navigational effort, such as the traversal time or distance [11].

In general, the notion of configuration space C is used to describe the possible positions of a robot. In this dissertation, the problem of 2D path planning in the cartesian plane is considered. Therefore, on the most basic level, $C = R^2$ (where R denotes the set of real numbers), and each configuration can be expressed as a pair of coordinates (x, y) (more complex configuration spaces could include for example the vehicle's heading angle). Let $S = (s_x, s_y)$ denote the starting point and $G = (g_x, g_y)$ the destination (goal). Then, let $C_{free} \subset C$ denote the configuration space containing all safe points within the path. Note that in such a case $\overline{C_{free}}$ will describe the obstacles.

The aim is to find a sequence of control commands such that the resulting sequence of movements M starts with the point S, ends with G, and all the visited points in between are within the safe regions of the path.

$$M = \left((s_x, s_y), (x_1, y_1), (x_2, y_2), \ldots, (g_x, g_y) : \, (x_k, y_k) \in C_{free} \, \forall \, k \in \mathbb{N}_+ \right)$$

Finally, it is important to note that the complexity of motion planning problems can grow, as the system may be restricted by some external constraints, be subject to the uncertainty of the associated environment, or depend on its current trajectory (in which case it is called nonholonomic).

Path planning algorithms could be classified into three sections [12]:

- Reactive control - reacting to the environment when discovering it
- Representational world modelling - following some pre-defined grids and maps
- Combinations of both

2.2 Model Predictive Control

A model predictive controller is a mathematical tool used to derive a collection of control inputs based on the predicted states of some, often highly nonlinear, control systems [8]. To begin with, let the model's state be denoted by a vector $\boldsymbol{x}$ and the control inputs by a vector $\boldsymbol{u}$. Then, the evolution of the system f between two subsequent time steps $n \to n+1$ can be written as:

$$\boldsymbol{x}\left(n+1\right) = f\left(\boldsymbol{x}\left(n\right), \boldsymbol{u}\left(n\right)\right)$$

which can be further simplified notation-wise as:

$$\boldsymbol{x}_{n+1} = f\left(\boldsymbol{x}_n, \boldsymbol{u}_n\right)$$

Repeatedly applying this formula within some finite time window (called the time horizon) results in predicting how will the system behave in the near future (Figure 2.1).

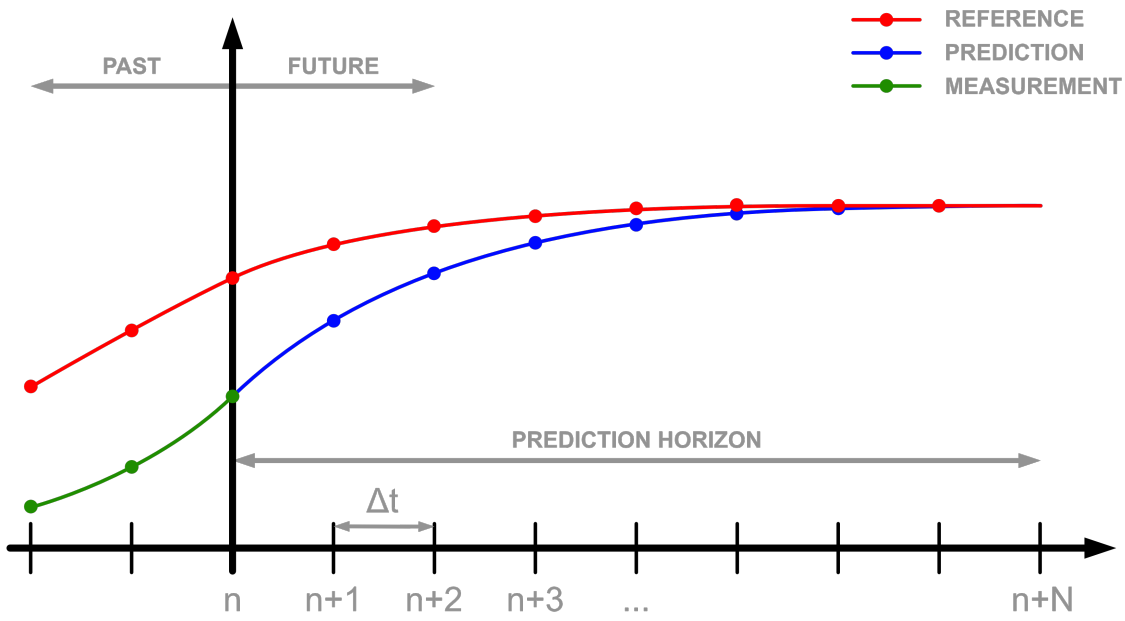


Figure 2.1: Model Predictive Control

In order to modify the control inputs, or, more specifically, to formulate what should they be for the next iteration, MPC attempts to adjust them to minimise some predefined cost function f_{cost}:

$$\min_{\boldsymbol{u}} f_{cost}\left(\boldsymbol{x}, \boldsymbol{u}\right)$$

The cost functions can be split into the **stage** and **terminal** costs, which are sometimes called the running and endpoint costs, or Lagrangian and Mayer terms. The former one is added at each time step within the horizon, while the latter one is only added once, at the end. Let l denote the stage cost and m the terminal cost. Then the cost function f_{cost} becomes:

$$f_{cost}\left(\boldsymbol{x}, \boldsymbol{u}\right) = \sum_{n=0}^{N} l\left(\boldsymbol{x}_n, \boldsymbol{u}\right) + m\left(\boldsymbol{x}_N, \boldsymbol{u}\right)$$

where N is the length of the horizon and the next state can be computed using the previous state and the input vector ($\boldsymbol{x}_{n+1} = f\left(\boldsymbol{x}_n, \boldsymbol{u}\right)$), as described earlier. Sometimes, the cost function can also include an additional, arbitrary control penalty cost, so that the issue of planning problem infeasibility is avoided.

Together with the control constraints, the optimisation problem for the current horizon becomes:

$$\begin{aligned} \min_{\boldsymbol{u}} \quad & \sum_{n=0}^{N} l\left(\boldsymbol{x}_n, \boldsymbol{u}\right) + m\left(\boldsymbol{x}_N, \boldsymbol{u}\right) \\ \text{subj. to} \quad & \boldsymbol{x}_{n+1} = f\left(\boldsymbol{x}_n, \boldsymbol{u}\right) \; \forall \, n \in \mathbb{N}_+ \\ & \boldsymbol{x}_0 = \boldsymbol{x}\left(0\right) \\ & \boldsymbol{u} \in \boldsymbol{U} \end{aligned}$$

where $\boldsymbol{U}$ denotes all possible inputs, taking the constraints into consideration. It is important to mention that most constraints will be in form of the $c\left(\cdot\right) \leq 0$ inequalities, where $c\left(\cdot\right)$ is some function utilising the model's formulation.

Chapter 3

Algorithms

This chapter describes the control strategy used in this dissertation and the associated algorithms that address the research objective. Firstly, after introducing the overall control loop, it explains how the Single Track model with slip factor works, providing a system of equations defining it. Then, it features a detailed explanation of how following a target reference point is achieved with gls and provides an example utilising it to follow a pre-defined trajectory. Afterwards, three FTG variants (namely, *Farthest Point FTG*, *Middle Point FTG*, and *Halves FTG*) are included, defining how exactly do the control methods considered in this thesis work. Finally, an approach to select the racing track's starting line, as well as lap timing methodology, are provided.

3.1 Control strategy

In general, the vehicle is driven by finding some target reference point at first, and then solving the optimisation problem and deriving the control inputs. Every iteration FTG is run to cleverly avoid the obstacles and generate some point to drive towards, and then MPC is used to derive the control inputs (Figure 3.1).

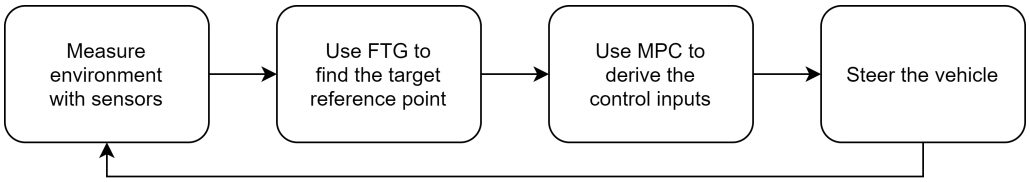


Figure 3.1: Control loop

The communication between both algorithms formulates the control strategy of this thesis.

3.2 Vehicle Model

This thesis models the F1/10 racing vehicle with the Single-Track model with slip factor (Figure 3.2), also known as the bicycle model [13]. While it is more accurate than simpler models such as the Point-Mass model, it is also more computationally expensive. Overall, it provides a good trade-off between accuracy and performance.

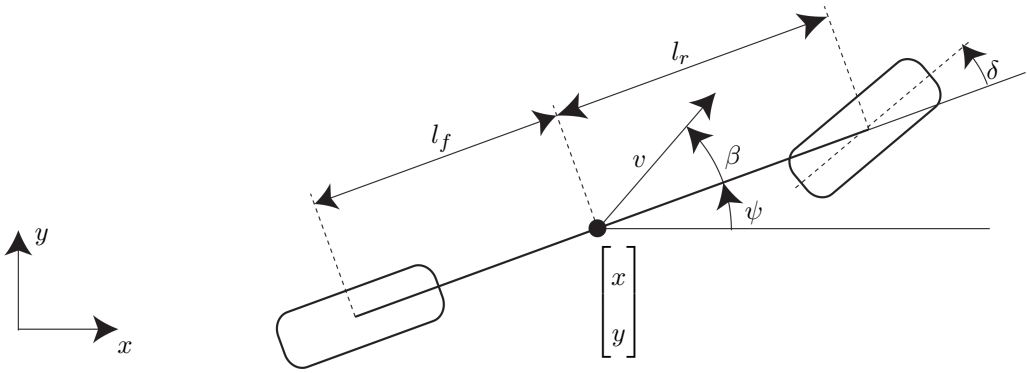


Figure 3.2: Single Track Model

More precisely, let the vehicle's position be denoted with a pair of cartesian coordinates (x, y), its current heading angle with ψ, velocity with v, and the steering angle with δ.

Then, the slip angle will be denoted with β, and the distances from the car's centre of gravity to its rear and front axels with l_r and l_f. The model equations

$$\dot{x} = v\cos(\psi + \beta)$$
$$\dot{y} = v\sin(\psi + \beta)$$
$$\dot{\psi} = \frac{v\cos(\beta)}{l_f + l_r}\tan(\delta)$$
$$\beta = \arctan\left(\frac{l_r}{l_f + l_r}\tan(\delta)\right)$$

will be used to derive the next states of the MPC formulation in section 3.3.

3.3 Control methods

This section introduces the MPC concepts used to follow a specific reference point and the FTG algorithms utilising them. More specifically, a collection of three control methods combining MPC and FTG is provided.

3.3.1 Distance-based point following

The vehicle is represented with three state variables and two input variables:

$$\boldsymbol{x} = [x, y, \psi]$$
$$\boldsymbol{u} = [v, \delta]$$

Each iteration the control inputs will be derived using the model equations defined in the previous section, by minimising the distance between the vehicle and some preselected target reference point. More precisely, the terminal cost is defined using squared Euclidean distance:

$$f_{d-cost} = (t_x - p_x)^2 + (t_y - p_y)^2$$

where the target point is at coordinate (t_x, t_y), and the vehicle at its final prediction state is at position (p_x, p_y). While this rather simple formula is more than enough to result in the vehicle following the target reference point, it can be enhanced for example by also considering the correctness of the heading angle ψ [14].

It is also important to mention that since the target reference point (potentially) changes every code iteration, it must be defined as a time-varying parameter in the MPC formulation. Therefore, rather than only considering the state and input vectors, next states should be derived using an additional collection of time-varying parameters $\boldsymbol{z}$:

$$\boldsymbol{x}_{n+1} = f\left(\boldsymbol{x}_n, \boldsymbol{u}_n, \boldsymbol{z}_n\right)$$

3.3.2 Waypoints following

Following a pre-defined trajectory consisting of a collection of cartesian coordinates is an example use-case scenario demonstrating the distance-based point following (e.g. Figure 3.3). In order to utilise MPC, a custom way to continuously choose consecutive points in the reference trajectory, as the vehicle progresses, has been designed and implemented.

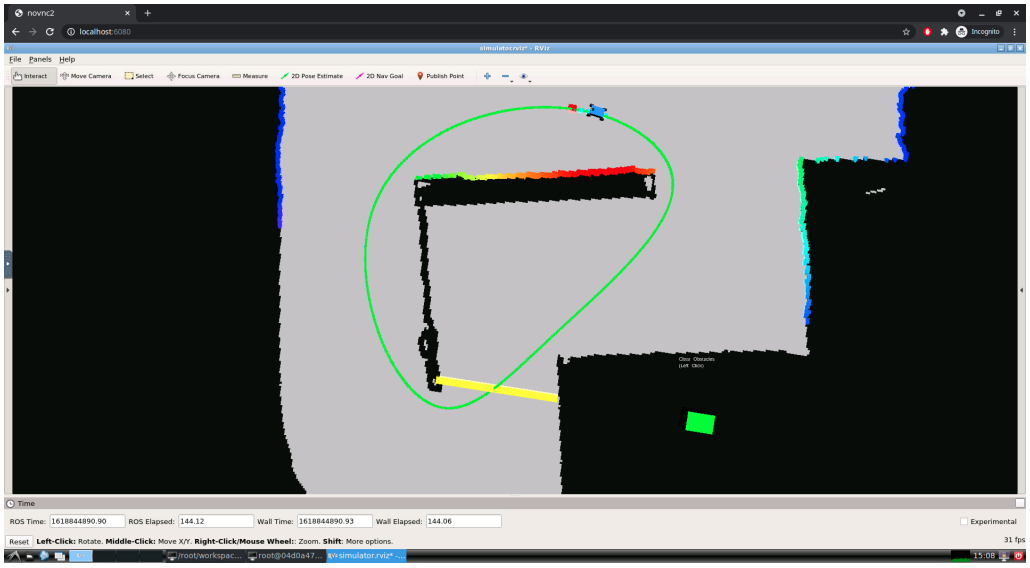


Figure 3.3: Waypoints Follower screenshot

Preferring simplistic solutions, the Euclidean distance between the vehicle and the current target reference point has been used. Upon reaching some threshold N, the next waypoint in the reference trajectory is chosen (the list of waypoints is sorted).

The resulting pseudocode can be seen in Listing 3.1:

```
waypoints := read_from_file(waypoints.csv)
target_waypoint := waypoints.first()
while True:
    distance := compute_distance_to_waypoint(target_waypoint)
    if (distance < THRESHOLD) then
        target_waypoint := get_next_waypoint()
    endif
endwhile
```

Listing 3.1: Waypoints Following pseudocode

Naturally, this solution depends entirely on the value of the threshold, as it can be parametrised differently for different race tracks. A more verbose solution should for example consider distances to intermediate points within the trajectory.

3.3.3 Follow The Gap

FTG is a path planning method used to safely avoid obstacles by driving towards gaps within the track [9]. For clarity, since there exist multiple variants of FTG, each method in this thesis is thoroughly described, and an associated pseudo code is provided.

Farthest Point

Farthest Point FTG works by selecting the farthest point within the largest gap in the current LiDAR scan area. Additionally, a safety bubble is defined around the point closest to the vehicle, to avoid driving near the obstacles. More precisely, the closest LiDAR point within the scan range is first selected, and the scan ranges of all points within some radius r away from it (including the point itself) are marked with some ignore distance value (typically 0). Then, a longest sequence of non-ignored scan ranges is found, and the farthest (largest scan range value) point within such sequence will be the trajectory reference point (Figure 3.4).

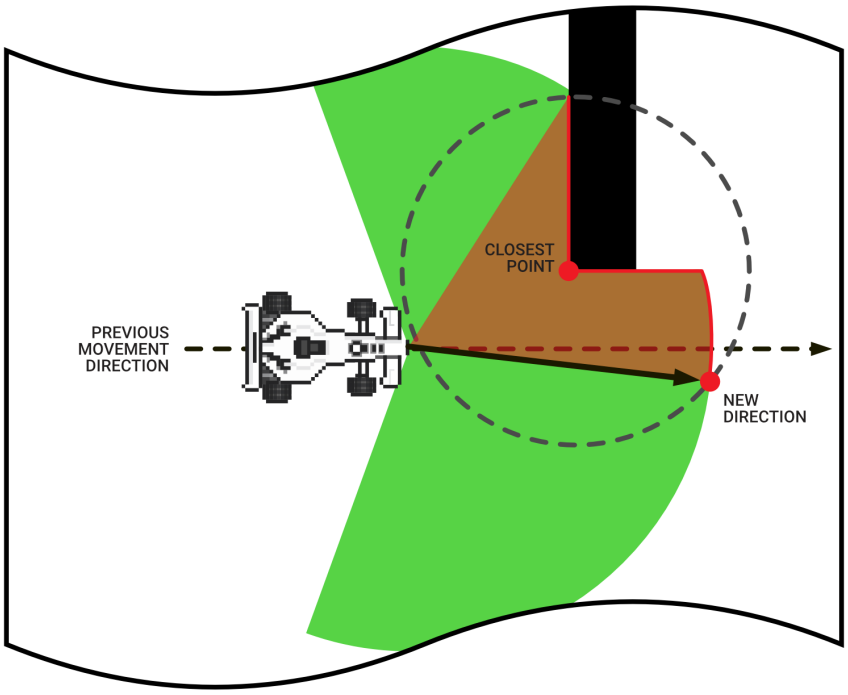


Figure 3.4: Farthest Point FTG

Naturally, there exist more advanced versions of this algorithm, which for example further break down the sequences of non-ignored scan ranges by finding all ”jumps” in consecutive LiDAR readings. A jump is defined as a difference between two consecutive ranges, such that it is higher than some threshold N.

The pseudocode for this algorithm is present in Listing 3.2:

```
while True:
    points := process_lidar(lidar_scan)
    closest_point := min(points, key=lambda point: point.range)
    mark_safety_radius(closest_point, points)
    sequence := find_longest_sequence_of_safe_points(points)
    target_point := max(sequence, key=lambda point: point.range)
endwhile
```

Listing 3.2: Farthest Point FTG pseudocode

Middle Point

Middle Point FTG works by selecting the middle point within the largest gap in the current LiDAR scan area. Similarly to the Farthest Point FTG, a safety bubble is defined to avoid driving into the obstacles. However, in this case, the bubble is created around the vehicle, rather than around the point closest to it. Just like in the previous algorithm, all points within the safety radius will be marked with 0. Then, upon finding the longest sequence of non-ignored scan ranges, the middle point is selected (Figure 3.5).

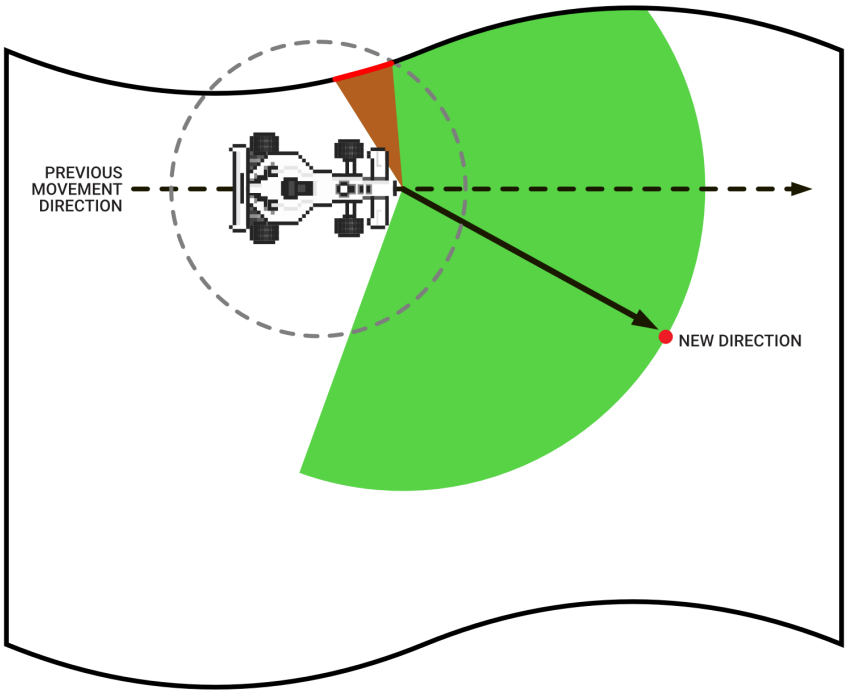


Figure 3.5: Middle Point FTG

The pseudocode for this algorithm is present in Listing 3.3:

```
while True:
    points := process_lidar(lidar_scan)
    points := filter_by_safety_radius(points)
    sequence := find_longest_sequence_of_safe_points(points)
    target_point := sequence[len(sequence)//2]
endwhile
```

Listing 3.3: Middle Point FTG pseudocode

Halves

Halves FTG is a custom, proposed method for selecting the target reference point (3.6).

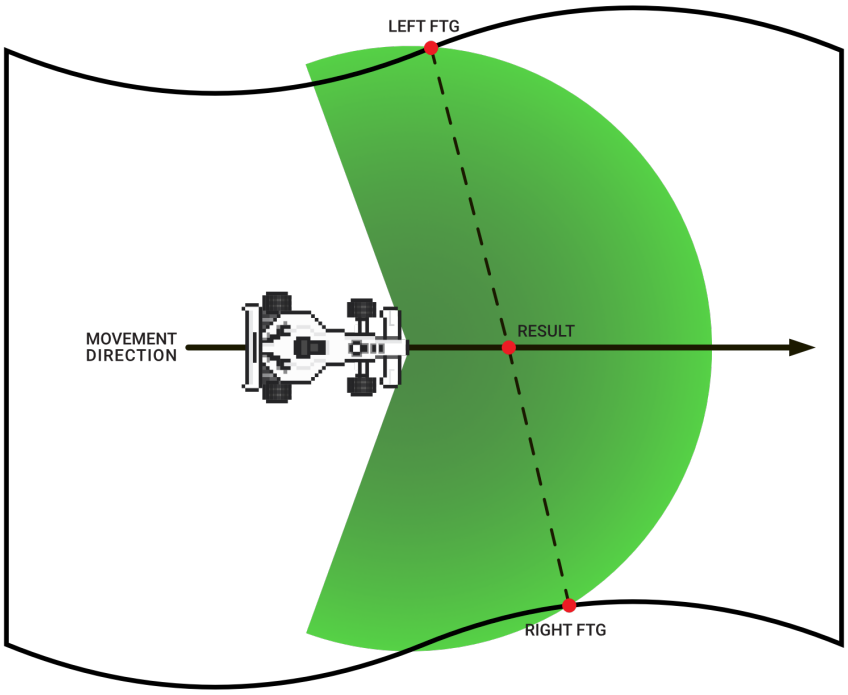


Figure 3.6: Halves FTG

Let L denote LiDAR scan ranges to the left-hand side of the vehicle, and R denote LiDAR scan ranges to the right-hand side of the vehicle. First, to refine the final results, all scan ranges are passed through a filter, such that any values above some threshold N will be ignored. This is to ensure the results are not (nearly) the same as if the farthest point FTG algorithm was run. Then, the Farthest Point FTG is run for L and R to find a pair of target reference points. Finally, these target points are combined (by averaging the coordinates) to form a single target reference point, which is then selected as the final target reference point, used in point-following MPC. In a case when either of the halves fails to produce a target point (for example because all points are too far away from the vehicle), a default position for the target point is selected based on the tuning parameters.

The pseudocode for this algorithm is present in Listing 3.4:

```
while True:
    points := filter_by_max_distance(process_lidar(lidar_scan))
    left_points, right_points := split_into_halves(points)
    left_target_point := farthest_point_ftg(left_points)
    right_target_point := farthest_point_ftg(right_points)
    sum_of_x_coordinates := left_target_point.x + right_target_point.x
    sum_of_y_coordinates := left_target_point.y + right_target_point.y
    target_point := (sum_of_x_coordinates / 2, sum_of_y_coordinates / 2)
endwhile
```

Listing 3.4: Halves FTG pseudocode

3.4 Measuring Lap Time

To accurately measure how long does it take for a vehicle to traverse the entire race track, a lap time measurement strategy has been designed and developed.

3.4.1 Defining the starting line

To begin with, a starting line must be created. Since the racing tracks vary and there is no unified starting position for the vehicle, a dynamic method marking such a line must have been created. Intuitively, a starting line is just some line segment defined by two points A and B, such that they are on the opposite edges of some area within the racing track.

It would make sense for the starting line to be perpendicular to the middle of the track. However, the orientation of the vehicle is unknown at first, so the car may not necessarily be perpendicular to the road. Therefore, to accommodate for the initial rotation, the starting line simply must be perpendicular to the initial rotation of the car, where the rotation axis is just a line passing through the back and the front of the car. More precisely, measurements of the LiDAR scan at the right-hand side of the vehicle and the left-hand side of the vehicle are taken and converted to cartesian coordinates. These will be the A and B points defining the line segment AB (Figure 3.7).

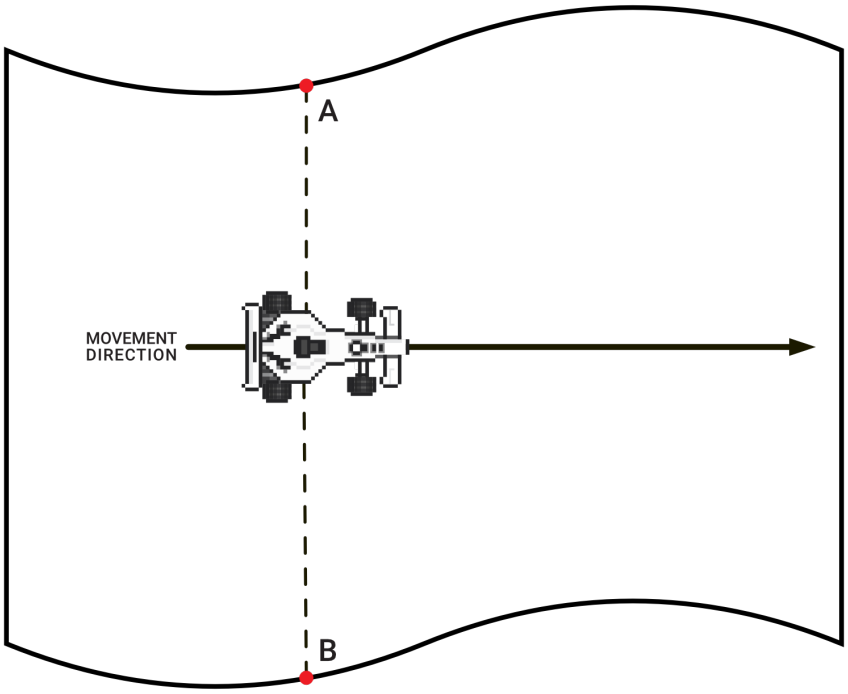


Figure 3.7: Creating the starting line (graphic)

The implementation within the simulator can be seen in Figure 3.8.

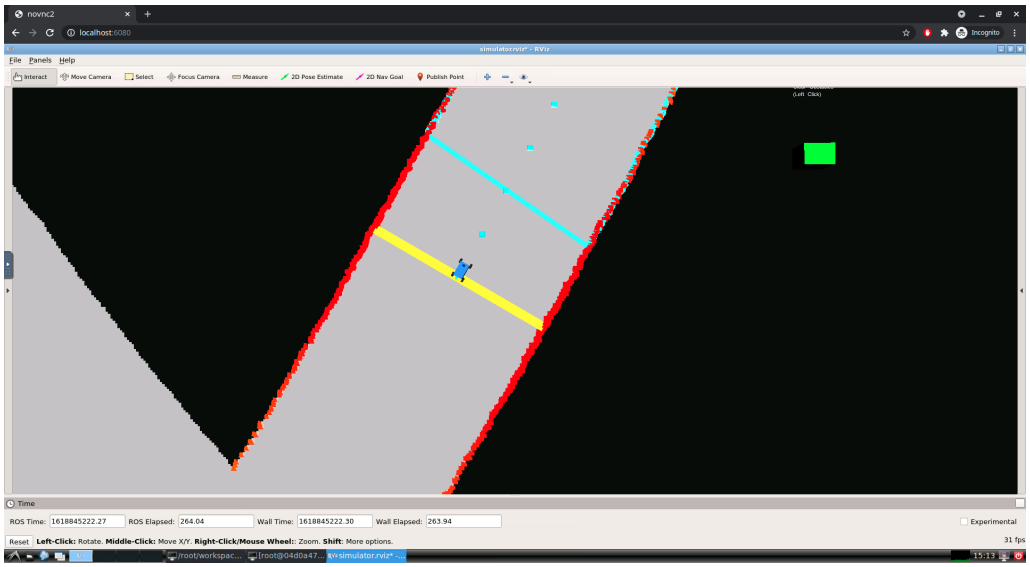


Figure 3.8: Creating the starting line (screenshot)

3.4.2 Finishing a lap

To find out the time required for the vehicle to traverse the entire race track, it must be known when does a car pass through the starting line. This requires computing the distance between the vehicle's position and the line segment AB. Let $A = (a_x, a_y)$ and $B = (b_x, b_y)$. Then, for a vehicle at position $P = (p_x, p_y)$:

$$l = (a_x - b_x)^2 + (a_y - b_y)^2$$

$$c = max\left(0, min\left(1, \frac{\begin{bmatrix} p_x - a_x \\ p_y - a_y \end{bmatrix} \cdot \begin{bmatrix} b_x - a_x \\ b_y - a_y \end{bmatrix}}{l}\right)\right)$$

$$p = (a_x + c\,(b_x - a_x)\,, a_y + c\,(b_y - a_y))$$

$$d = (P - p)^2$$

where l can be interpreted as squared length of the segment, c as the clamped value, and p as the projected value. Then, d is the final (squared) distance between the line segment AB and the vehicle at position P.

However, determining if the vehicle has finished a lap requires slightly more than just checking the distance. This is because the vehicle can be within some distance N from the starting line for multiple iterations, as the time delta between them could be extremely small. Therefore, a boolean flag has been introduced to mark finishing a lap.

Most intuitively, if the vehicle was previously far away from the starting line, and now is close (within some threshold E), the vehicle must have been getting closer to the starting line over time, and therefore the current lap is being finished. In that case, we can raise the flag and temporarily stop checking whether the car is in the process of finishing the lap. Then, on the contrary, if the vehicle was close to the starting line, and is now far away (farther than some threshold S), it must be driving away from the line and therefore starting a new lap. In that case, the flag can be lowered and the next code iterations will start checking for finishing the lap again.

The pseudocode is present in Listing 3.5:

```
flag_raised := False
while True:
    distance_to_start := compute_distance_to_start()
    if (flag_raised) and (distance_to_start <= E_THRESHOLD) then
        mark_new_lap()
        flag_raised := False
    else if (!flag_raised) and (distance_to_start >= S_THRESHOLD) then
        flag_raised := True
    endif
endwhile
```

Listing 3.5: Detecting new laps pseudocode

Chapter 4

Technical Stack

This chapter describes the practical aspects of the project, expressing the technicalities of the simulation and the source code structure. To address the research aim, the experimentation setup present in this chapter has been developed, allowing rapid prototyping, debugging, and visualising the FTG algorithms utilising MPC. In addition, the framework is available on any platform via the docker containerisation feature and can be used within any machine quite easily. Finally, an introduction to the Python library providing an implementation of MPC is included at the end of this chapter. All resources and the framework's source code can be accessed at the online repository [15].

4.1 Robotic Operating System

ROS is a modelling and simulation framework used to create robotic software [16]. It features an ability to model complex machines as well as to create the surrounding environments they typically operate in. Furthermore, it allows interactions between the robots and the environments through a collection of simulated sensors. ROS has been chosen as the F1/10 simulator is based on it and as such features similar functionalities.

4.1.1 Sensors

The most important environmental data provided by ROS are LiDAR scans and odometry information. While the simulation itself provides artificial values, it is important to understand what would the values measured in a real-world scenario be, and how to obtain them. Therefore, this section briefly introduces the ideas behind LiDAR scanners and vehicle position computations.

LiDAR is a sensing technology capable of measuring distance to objects with lasers. It emits near-infrared light beams which are reflected by the environment and analysed to estimate the distances to the objects around the sensor [17] (notice the LiDAR scanner on top of the F1TENTH racing car in Figure 4.1).

Figure 4.1: F1TENTH racing car with LiDAR on top [18]

While it is often used in 3D imaging scenarios [19, 20], in the case of the F1/10 simulator only a flat array of ranges (distances) is provided at consistent angle intervals. Given that the scanner's field of view is always between some minimum and maximum angles α, β, and knowing the angle interval θ between each of the laser beams (Figure 4.2), it is possible to compute coordinates of the detected obstacles relative to the car's position.

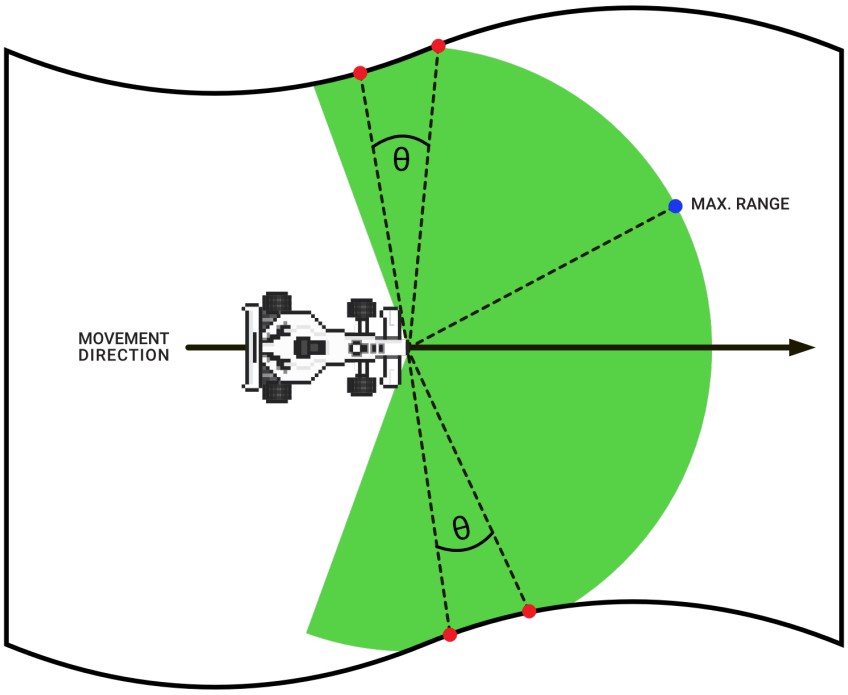


Figure 4.2: Graphical explanation of how LiDAR works

However, to compute the scan's global coordinates (rather than relative to the car), odometry calculations are additionally required. They can be computed by comparing the change of the vehicle's state between two measurements [21]. Together with some global coordinates system denoting the reference position, this difference allows understanding how the vehicle progresses through the map over time. Note that in real-world scenarios, tracking vehicle position using odometry may be error-prone, for example, due to the wheel slippage or floor roughness when localising the car with wheel encoders [22].

4.1.2 Simulator

The F1/10 simulator is a 2D, lightweight framework based on ROS, written in C++ programming language [23]. It is recognised by several universities across the globe and there exist several research documents utilising it [24–26]. Moreover, the F1/10 cars modelled in the simulator are used in the industrial setup for rapid development and prototyping [27].

The tool exposes a vast collection of Application Programming Interface (API)-s and a complex visualisation environment (Figure 4.3). To make things easy, most common interactions and command arguments' combinations are summarised in the shell scripts, in the GitHub repository (4.2).

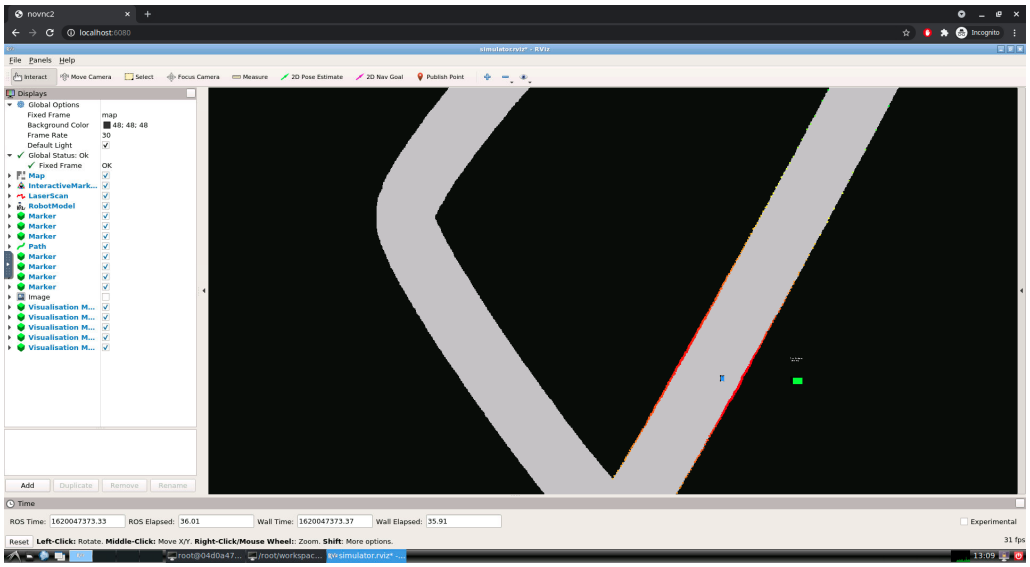


Figure 4.3: Simulator's GUI after launch

To give a brief introduction on how does the simulator interact with the code, it is crucial to introduce the concept of the publisher-subscriber pattern, which is used to exchange messages between distinct software components [28]. In this system, the publishers send data categorised with custom labels while the subscribers express interest in receiving such data, but only the information categorised with some pre-selected labels (4.4). Notably, the publishers don't know about the subscribers, and vice-versa [28].

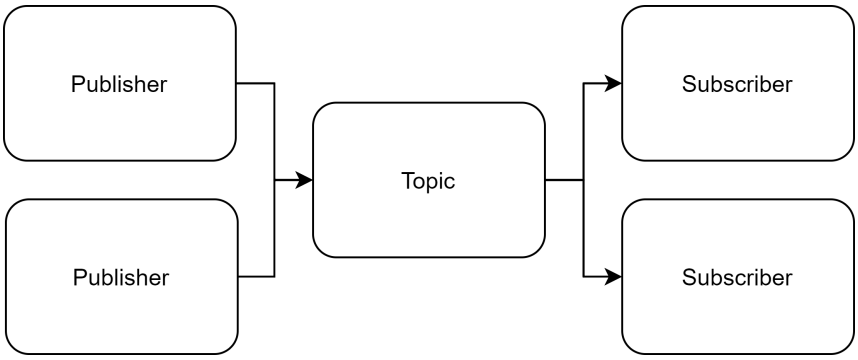


Figure 4.4: Publisher-subscriber diagram

While the simulator exposes several API topics (4.5), the most important considered by this thesis are the scan, the odometry, and the drive ones. The scan and odometry topics collect and publish information on the modelled environment. They track the position and orientation of the vehicle, as well as record data on segments of the racing track around the car. On the other hand, the drive topic listens to incoming data, as this is where the control inputs can be sent. It accepts *AckermannDrive* messages consisting of velocity, steering angle, and metadata. Any output data created by MPC will be published to this topic, resulting in ROS driving the car.

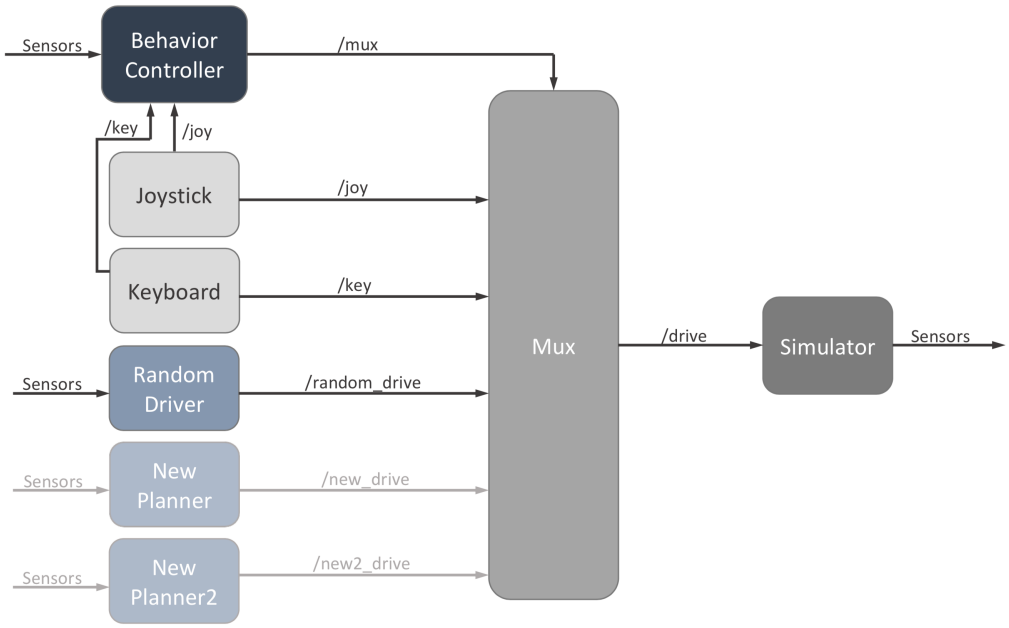


Figure 4.5: Simplified graph of ROS nodes [29]

In addition, ROS exposes an ability to visually inspect the results of the simulation by providing highly parameterisable visualisations. In the case of the simulator, it displays the racing car, as well as allows to load custom race tracks and add ROS markers. The tracks are loaded by providing a pair of files and a correct simulator launch argument. The first file is just a Portable Network Graphics (PNG) representation of the track (e.g. Figure 4.6), whereas the second file is a YAML description of the track's parameters. Together, they describe which pixels are considered an obstacle, and which are considered free space. Additionally, the YAML file holds the initial position and rotation of the racing car.

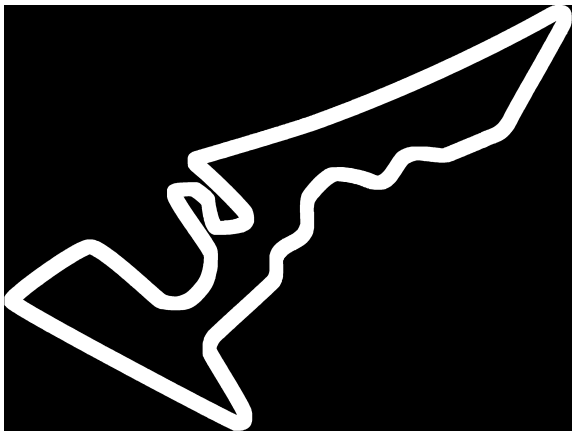


Figure 4.6: Vegas racing track PNG

The markers can be added by publishing data to marker topics, each exposed in the associated RVIZ file. There exist several marker types which can be sent in each marker message, but this thesis only considers *points*, *line lists*, and *line strips*. The *point* is a single dot at some coordinates (x, y), the *line lists* are a collection of lines where every two vertices are considered as the edges of a line, and the *line strips* are a collection of lines where every two consecutive vertices are considered as the edges of the next line (essentially ensuring the lines are connected).

4.1.3 Dockerisation

Since the F1/10 simulator is based on *Melodic* version of ROS, its target platform is the *Ubuntu 18.04 (Bionic)* release. This causes a lot of issues with running the framework on a more modern operating system.

Therefore, the proposed solution is to enclose it in a docker container [30], as it allows running a virtual version of the target operating system on a different machine. In addition, configuration instructions can be included in the *Dockerfile*, which is a docker specific file used to define how the virtual system should be built, making the entire process easy to execute. Finally, because every docker container can be trivially rebuilt, the framework adds a layer of robustness, guarding against unexpected errors and system failures.

To create a *Dockerfile* for this thesis, a few configuration choices must be made at first. To begin with, the container must provide virtualisation of its desktop view, to allow visually inspecting the output of ROS commands (including visualisation of the simulator). The selected method to achieve this goal is Virtual Network Computing (VNC), which is an easy-to-use software designed to remotely access other machines [31]. The respective Dockerfile commands used to run VNC-compatible Ubuntu are present in listing 4.1.

```
FROM dorowu/ubuntu-desktop-lxde-vnc:bionic
RUN apt-get update
```

Listing 4.1: Creating a VNC-based Dockerfile

Once the root docker image is available (and after updating the system), it is necessary to define where will all relevant files get stored. Therefore, a workspace folder is created (Listing 4.2).

```
ARG WORKSPACE=/root/workspace
RUN mkdir -p $WORKSPACE
```

Listing 4.2: Creating a workspace folder in the Dockerfile

Afterwards, a new version of Python is fetched to replace the default installation (Listing 4.3).

```
RUN cd $WORKSPACE && curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
RUN apt-get install python3.8 python3.8-dev python3.8-distutils build-essential -y \
    && cd $WORKSPACE \
    && python3.8 get-pip.py \
    && rm get-pip.py \
    && update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.8 1
```

Listing 4.3: Installing Python in the Dockerfile

At this stage ROS can be installed. Getting it is slightly non-trivial as it requires including ROS release in the system's sources list, as well as registering the associated key (Listing 4.4).

```
RUN apt−get install dirmngr −y
RUN sh −c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release −sc) main" > /etc/apt/sources.list.d/ros−latest.list' \
    && apt−key adv −−keyserver 'hkp://keyserver.ubuntu.com:80' −−recv−key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654 \
    && apt−get update \
    && apt−get install ros−melodic−desktop−full −y
```

Listing 4.4: Installing ROS in the Dockerfile

Then, the installation of the F1/10 dependencies happens. Firstly, ROS packages are installed. Subsequently, the simulator is cloned from GitHub and built using *catkin*, which is a ROS build system package used to compile some of the framework's components (Listing 4.5).

```
RUN apt−get install git −y \
    && apt−get install ros−melodic−tf2−geometry−msgs ros−melodic−ackermann−msgs ros−melodic−joy ros−melodic−map−server −y \
    && python3.8 −m pip install catkin_pkg \
    && mkdir −p $WORKSPACE/simulator/src \
    && cd $WORKSPACE/simulator/src \
    && git clone https://github.com/f1tenth/f1tenth_simulator.git \
    && bash −c "source /opt/ros/melodic/setup.bash && cd $WORKSPACE/simulator && catkin_make"
```

Listing 4.5: Installing the F1/10 simulator in the Dockerfile

Afterwards, the software package provided with this dissertation is downloaded and installed (Listing 4.6).

```
RUN mkdir $WORKSPACE/code \
    && cd $WORKSPACE/code \
    && git clone https://github.com/TheCodeSummoner/f1tenth−racing−algorithms.git \
    && cd $WORKSPACE/code/f1tenth−racing−algorithms \
    && python3.8 −m pip install .
```

Listing 4.6: Installing f1tenth-racing-algorithms in the Dockerfile

Later, *yq*, which is a tool used to modify YAML files [32], is retrieved (Listing 4.7). It will allow adding a collection of ROS marker topics for more verbose visualisations.

```
RUN curl -LJ0 https://github.com/mikefarah/yq/releases/download/v4.6.0/yq_linux_amd64 --output /usr/local/bin/yq \
    && chmod +x /usr/local/bin/yq \
    && $WORKSPACE/code/f1tenth-racing-algorithms/assets/scripts/prepare-visualisation-topics.sh
```

Listing 4.7: Installing yq in the Dockerfile

Finally, a setup shell script is made to contain all necessary environment variables and ROS commands (Listing 4.8). Interacting with the simulator would be impossible without them.

```
RUN touch $WORKSPACE/setup-workspace.sh \
    && echo "source /opt/ros/melodic/setup.bash" >> $WORKSPACE/setup-workspace.sh \
    && echo "source $WORKSPACE/simulator/devel/setup.bash" >> $WORKSPACE/setup-workspace.sh \
    && echo "export PYTHONPATH=/opt/ros/melodic/lib/python2.7/dist-packages" >> $WORKSPACE/setup-workspace.sh
```

Listing 4.8: Creating a setup-workspace script in the Dockerfile

4.2 GitHub

GitHub is a well-known version control service providing a user interface for the *git* version control tool, as well as a collection of software components enhancing the development [33]. Furthermore, GitHub provides extensive means of communication between the contributors and the consumers, for example through *issues*, a *kanban* board, or recently added *discussions*. It is used in this project to host the associated source code, and while it is not going to use all of the provided tools at the time of the release of this thesis, GitHub is an excellent choice for future maintenance and enhancements.

The code is written in Python programming language (since it is supported by the simulator) and structured in a hierarchy of packages (4.7). The *f1tenth* package holds a collection of sub-packages will all required python modules, whereas the *assets* folder contains support components, such as screenshots or shell scripts. Notably, the associated *Dockerfile* will be stored outside of both folders.

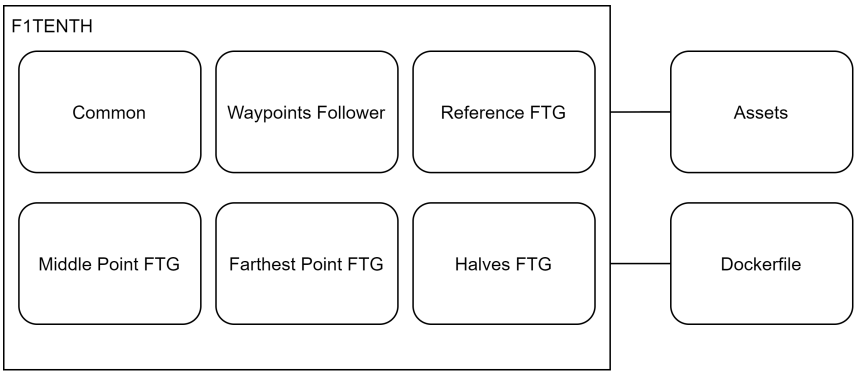


Figure 4.7: Source code structure

In more details, the *assets* folder contains the definitions of racing tracks (called maps), a collection of utility shell scripts, several videos showcasing the algorithms, and a number of other non-code files. On the other hand, the *f1tenth* directory contains the implementation of the algorithms, with the *common* package being the most important component, as it stores base classes and shared utility methods used in the other sub-packages. It defines the Point Following MPC (3.3.1) and a base implementation of FTG utilising it, as well as a class capable of interacting with the simulator and a variety of visualisation constructs.

Notably, the interaction with ROS happens by subscribing to and publishing to several exposed topics (described in more details in section 4.1.2). The summary of the communication pipeline can be seen in Figure 4.8:

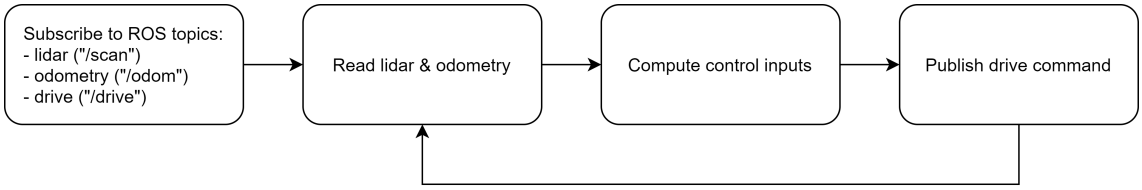


Figure 4.8: Communication pipeline

Finally, all sub-packages contain algorithms utilising both MPC and FTG, apart from the *Reference FTG*, which does not utilise MPC and acts only as a reference algorithm used during the development process. Refer to Figure 4.9 for the overall inheritance hierarchy:

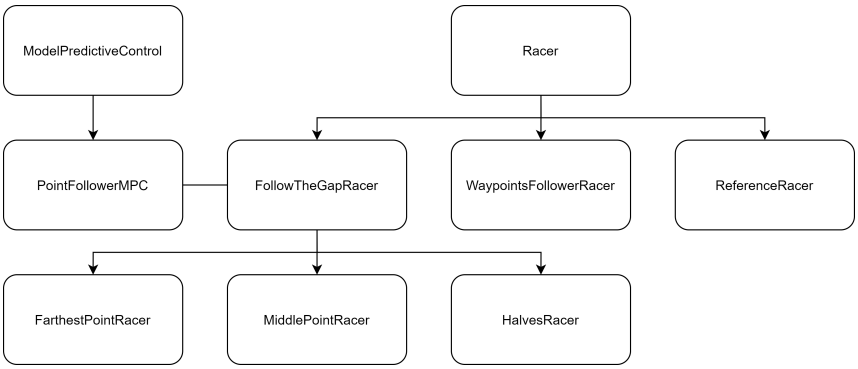


Figure 4.9: Inheritance hierarchy

4.3 Python MPC library

do_mpc is a Python library written to abstract away the difficulties of implementing MPC solvers [34]. Among other features, it offers a robust multi-stage model predictive control toolbox made available through a collection of extensively documented classes. In addition, the underlying infrastructure uses *casadi*, which is a popular open-source tool for nonlinear optimisation and algorithmic differentiation [35]. As with most other Python libraries, it can be installed with the *pip* package manager.

do_mpc consists of seven main modules: *model*, *simulator*, *optimizer*, *controller*, *estimator*, *data*, and *graphics*. In this thesis, *model* and *controller* are most used. The former one provides full description of the model, and the latter one includes all control-related functionality.

4.3.1 Model

On the most basic level, MPC models can be either discrete or continuous. Further configuration includes defining the state of the model, the control inputs, and the system dynamics. In Listing 4.9, following the declaration of the model (here, continuous rather than discrete), the state and input variables are defined. Then, the RHS (Right-Hand-Side) function defining how the state variable should be derived is included. Finally, the model is initialised with the *setup* function.

```
from do_mpc.model import Model
model = Model("continuous")
some_var = model.set_variable(
    var_type="_x",
    var_name="some_var",
    shape=(1, 1)
)
some_input = model.set_variable(
    var_type="_u",
    var_name="some_input",
    shape=(1, 1)
)
model.set_rhs("some_var", some_var + some_input) # definition on how some_var can be derived
model.setup()
```

Listing 4.9: Defining an MPC model

4.3.2 Controller

The MPC controller is responsible for solving the optimisation problem. It requires a collection of solver parameters, the objective functions, and the constraints. Once it's connected to the model, the initial state and guess can be marked to prepare the controller for future state predictions and control inputs derivation. In Listing 4.10, at first, the controller is created from the previously defined model and configured. Afterwards, its cost function and control constraints are set. Finally, the controller is initialised and ready to predict the modelled object's behaviour and derive the control inputs.

```
from do_mpc.controller import MPC
controller = MPC(model)
controller.set_param(n_horizon=some_horizon_length, t_step=some_timestep)
controller.set_objective(mterm=some_objective_function)
controller.bounds["lower", "_u", "some_input"] = some_lower_bound
controller.setup()
controller.x0 = initial_state
controller.set_initial_guess()
```

Listing 4.10: Defining an MPC controller

4.3.3 Plotting

Graphical functionalities in *do_mpc* are provided with wrappers around the *matplotlib* library [36]. While it is possible to interact with the data by extracting related *matplotlib* objects directly, calling the wrapping functions might prove to be more intuitive in the context of MPC. In Listing 4.11, a simple plot of how the state and control variables changed over time is included.

```
from do_mpc.graphics import Graphics
import matplotlib.pyplot as plt
graphics = Graphics(controller.data)
fig, axis = plt.subplots(2, sharex="all", figsize(16, 9))
graphics.add_line(var_type="_x", var_name="some_var", axis=axis[0], color="green")
graphics.add_line(var_type="_u", var_name="some_input", axis=axis[1], color="red")
```

Listing 4.11: Plotting MPC data

4.3.4 Example

To provide better understanding of the tool, let's consider a simple example problem. Let P denote the vehicle's position at coordinates (x, y), v its velocity, and a its acceleration. Then, let's denote the position after time dt as $P' = P + v' \times dt$, where $v' = v + a \times dt$. The vehicle will be controlled by adjusting its longitudinal and latitudinal accelerations. Additionally, to avoid explaining this simple example using a complex simulator, the *Python Arcade* Python library [37] will be used for visualisations. Note that if the below methods are unchanged in two consecutive listings, they don't require modifications at that time.

First, let's create a simple skeleton class, in which each method will be filled step by step (Listing 4.12).

```
import arcade
import numpy as np
import do_mpc
import casadi
import matplotlib.pyplot as plt



class Application(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)

    def setup_model(self):
        pass

    def setup_controller(self):
        pass

    def setup_plotting(self):
        pass

    def on_draw(self):
        pass

    def on_update(self, dt: float):
        pass

    def plot(self):
        pass

app = Application(800, 600, "do_mpc example")
arcade.run()
app.plot()
```

Listing 4.12: Example structure

Before any MPC code can be written, the visualisation and system dynamics must be prepared. For the purpose of having some goal for the optimiser to solve, let's consider the vehicle trying to reach some target point $T = (200, 300)$. Let's add *Python Arcade* visualisations (Listing 4.13) and inspect the results (Figure 4.10).

```
def __init__(self, width, height, title):
    super().__init__(width, height, title)
    arcade.set_background_color(arcade.color.WHITE)
    self.target = np.array([200, 300])
    self.position = np.array([0, 0])
    self.velocity = np.array([0, 0])
    self.acceleration = np.array([1, 1]) # 1, 1 to verify the point is moving
    self.setup_model()
    self.setup_controller()
    self.setup_plotting()

def on_draw(self):
    arcade.start_render()
    arcade.draw_point(
        x=self.target[0],
        y=self.target[1],
        color=arcade.color.RED,
        size=10
    )
    arcade.draw_point(
        x=self.position[0],
        y=self.position[1],
        color=arcade.color.LIGHT_BLUE,
        size=10
    )

def on_update(self, dt: float):
    self.velocity = self.velocity + self.acceleration * dt
    self.position = self.position + self.velocity * dt
```

Listing 4.13: Initial example canvas

Figure 4.10: Initial state of the Python Arcade canvas

The canvas is painted correctly and the vehicle is moving as expected, so the code is ready for the first MPC-related addition. First, let's create the model's state and input variables. While it would be enough to create the position variable only and derive it using acceleration, it will be easier to describe the dynamics in two steps with the help of velocity (Listing 4.14).

```
def setup_model(self):
    self.model = do_mpc.model.Model("continuous")
    self.position_var = self.model.set_variable(
        var_type="_x",
        var_name="position",
        shape=2
    )
    self.velocity_var = self.model.set_variable(
        var_type="_x",
        var_name="velocity",
        shape=2
    )
    self.acceleration_var = self.model.set_variable(
        var_type="_u",
        var_name="acceleration",
        shape=2
    )
    # derivative of position is velocity and derivative of velocity is acceleration
    self.model.set_rhs("position", self.velocity_var)
    self.model.set_rhs("velocity", self.acceleration_var)
    self.model.setup()
```

Listing 4.14: Adding the model

Next, the MPC controller ought to be created, to allow predicting vehicle's behaviour over time using the dynamics that have just been implemented (Listing 4.15). The objective function will consist of the Mayer term only, causing the vehicle to drive towards the target point by minimising the Euclidean distance between them:

$$C = (P - T)^2$$

Additionally, the control constraints will restrict the acceleration vector not to go over some threshold N:

$$a_x^2 + a_y^2 \leq N^2$$

```
def setup_controller(self):
    self.controller = do_mpc.controller.MPC(self.model)
    self.controller.set_param(
        n_horizon=20, t_step=0.2,
        nlpsol_opts={"ipopt.print_level": 0, "ipopt.sb": "yes", "print_time": 0}, # suppress console print
        store_full_solution=True # need this for graphics later on
    )
    lagrange_term = casadi.DM.zeros() # no stage cost
    mayer_term = (self.target[0] - self.position_var[0]) ** 2 + (self.target[1] - self.position_var[1]) ** 2
    self.controller.set_objective(lterm=lagrange_term, mterm=mayer_term)
    self.controller.set_nl_cons("cons", self.acceleration_var[0] ** 2 + self.acceleration_var[1] ** 2, 100)
    self.controller.setup()
    self.controller.x0 = np.array([self.position[0], self.position[1], self.velocity[0], self.velocity[1]])
    self.controller.set_initial_guess()
```

Listing 4.15: Adding the controller

Having the model and the controller, the vehicle can now utilise MPC to derive the control inputs (Listing 4.16).

```
def on_update(self, dt: float):
    self.velocity = self.velocity + self.acceleration * dt
    self.position = self.position + self.velocity * dt
    self.acceleration = self.controller.make_step(
        np.array([self.position[0], self.position[1], self.velocity[0], self.velocity[1]])
    ).flatten()
```

Listing 4.16: Deriving control inputs

The vehicle is now capable of moving on its own. However, it is unclear whether MPC is actually being used. Therefore, further visualisation of the predicted coordinates will be added (Listing 4.17). Let's also inspect the results (Figure 4.11).

```
def on_draw(self):
    arcade.start_render()
    arcade.draw_point(
        x=self.target[0],
        y=self.target[1],
        color=arcade.color.RED,
        size=10
    )
    arcade.draw_point(
        x=self.position[0],
        y=self.position[1],
        color=arcade.color.LIGHT_BLUE,
        size=10
    )

    for point in self.predicted:
        arcade.draw_point(
            x=point[0],
            y=point[1],
            color=arcade.color.LIGHT_BLUE,
            size=4
        )

def on_update(self, dt: float):
    self.velocity = self.velocity + self.acceleration * dt
    self.position = self.position + self.velocity * dt
    self.acceleration = self.controller.make_step(
        np.array([self.position[0], self.position[1], self.velocity[0], self.velocity[1]])
    ).flatten()
    x_predicted = self.controller.data.prediction(("_x", "position", 0)).flatten()
    y_predicted = self.controller.data.prediction(("_x", "position", 1)).flatten()
    self.predicted = np.dstack((x_predicted, y_predicted))[0]
```

Listing 4.17: Visualising trajectories

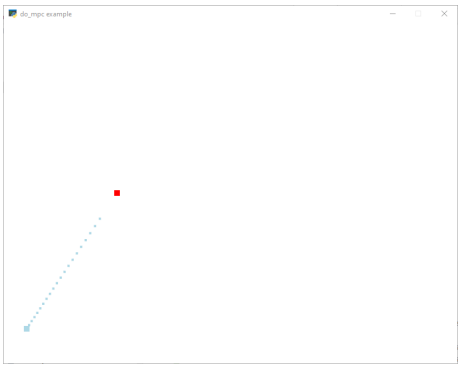


Figure 4.11: MPC with trajectory visualisation

Finally, let's add matplotlib plotting of the state and input variables (Listing 4.18) and inspect the results (Figure 4.12).

```
def setup_plotting(self):
    self.graphics = do_mpc.graphics.Graphics(self.controller.data)
    fig, ax = plt.subplots(2, sharex="all", figsize=(16, 9))
    fig.align_ylabels()
    self.graphics.add_line(var_type="_x", var_name="position", axis=ax[0], color="r")
    self.graphics.add_line(var_type="_x", var_name="velocity", axis=ax[1], color="g")
    self.graphics.add_line(var_type="_u", var_name="acceleration", axis=ax[1], color="b")
    ax[0].set_ylabel("Position")
    ax[1].set_ylabel("velocity (green), and acceleration (blue)")
    ax[1].set_xlabel("Time")

def plot(self):
    self.graphics.plot_results()
    self.graphics.reset_axes()
    plt.show()
```

Listing 4.18: Plotting model's state and control inputs

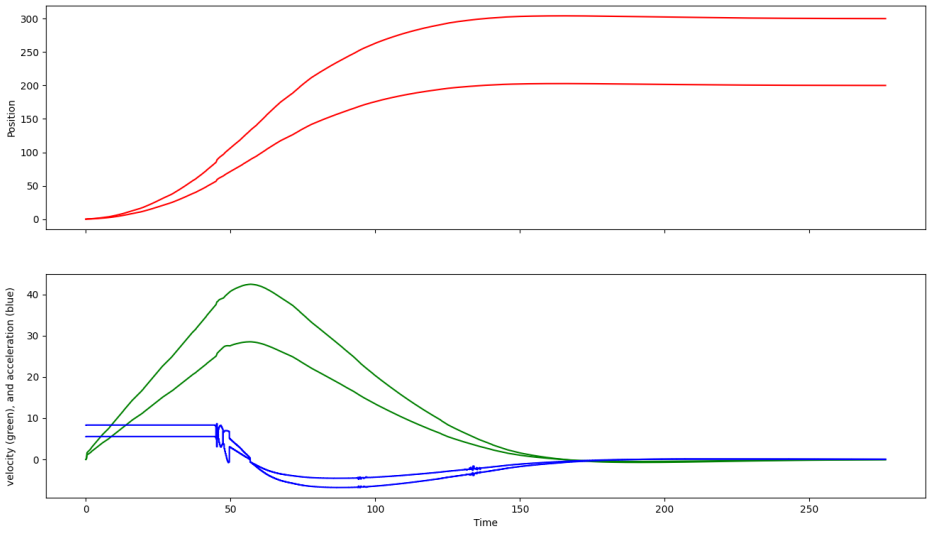


Figure 4.12: Plotting model's state and control inputs

This concludes creating a simple system utilising MPC to drive a vehicle towards the target reference point.

Chapter 5

Results and Evaluation

In this chapter, the algorithms get tested by performing several experiments and gathering their results. The control methods are then compared against each other and the associated issues are highlighted. In addition, the tuning parameters are thoroughly discussed and explained. Before evaluating the results, a hypothesis that *the racing track's width and turn curvatures affect safety assurance greatly* is stated, which this chapter attempts to also answer.

5.1 Tuning parameters

It is crucial to explain what the tuning parameters for each algorithm are, and how changing their values adjusts the behaviour of the control methods.

Firstly, three parameters have shared meaning between all of the algorithms. More specifically, *Horizon Length* and *Time step* are related to the configuration of MPC, corresponding to the length of the prediction horizon and Δt property shown in Figure 2.1. Section 2.2 can be accessed for more information. Then, the *LiDAR scan index* is responsible for adjusting the LiDAR scanner's field of view. As it can be seen in Figure 4.2, LiDAR works by sending a number of laser beams. In the case of the F1TENTH simulator, there are 1080 such beams at regular intervals, with $\theta \approx 0.0058$ (in radians).

Note that multiplying θ by the number of the laser beams gives 2π, which is equivalent to a full rotation around the origin (the vehicle), and is the default field of view of the scanner. Each increment in the *LiDAR scan index* parameter will result in the field of view decreasing by $2 \times \theta$ (decrease on each side of the vehicle). This can be seen in Figure 5.1.

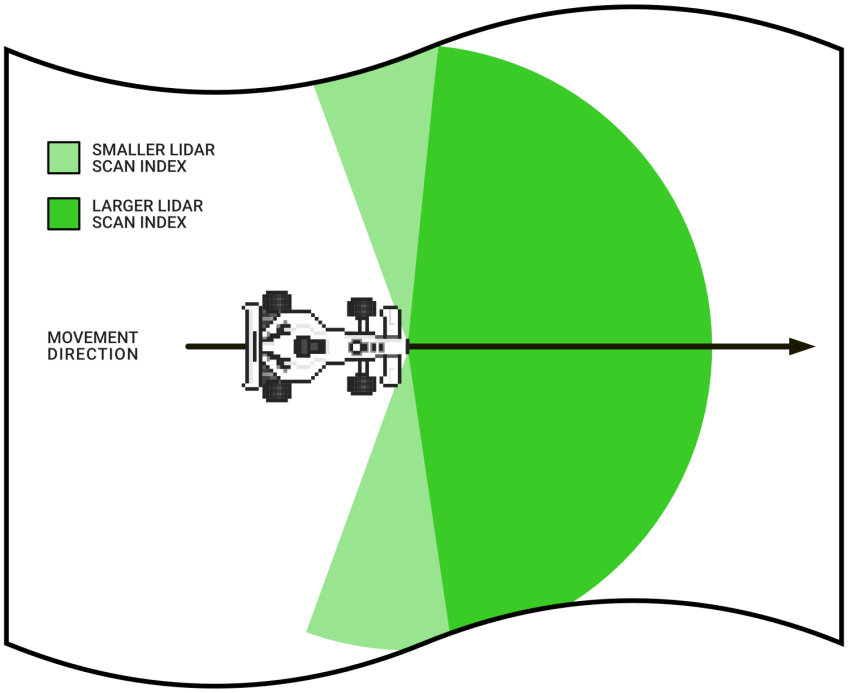


Figure 5.1: Larger values of the LiDAR scan index result in a smaller field of view

Another tuning parameter present in all algorithms is the *Safety Radius*. It is responsible for defining an area around the origin, in which all LiDAR points are considered unsafe. However, in the Middle Point FTG the safety area is around the vehicle, whereas in the other control methods it is around selected points, different from the position of the car. Larger radii mean larger areas defined around the origins, resulting in a smaller set of points considered in the algorithms.

Finally, in addition to the already presented tuning parameters, Halves FTG contains several additional values. *Distance limit* is responsible for filtering out LiDAR scan ranges, such that no parsed ranges are larger than some threshold N. The larger the limit, the smaller the set of points considered valid in the algorithm.

Then, the *Default LiDAR scan index* and the *Default LiDAR scan range* provide an ability to create an artificial reference point, in case the Farthest Point FTG fails to produce the farthest point for either of the halves. This can be seen in Figure 5.2.

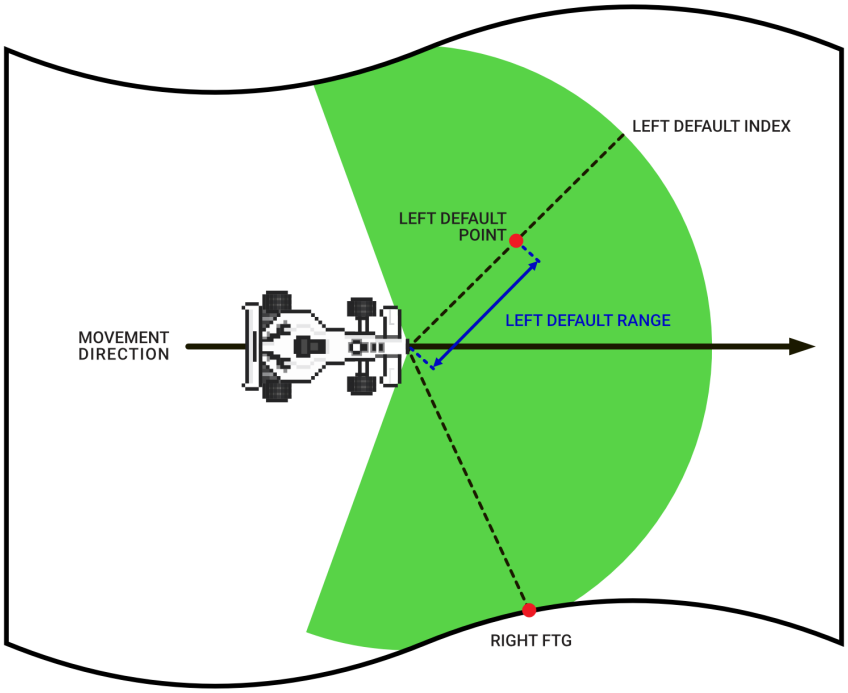


Figure 5.2: A default point is used when no valid FTG point has been detected on the left

5.2 Initial comparison

At first, the algorithms are tuned to run on the *Vegas* racing track and timed. The parameters (Table 5.1) were selected such that the vehicle can traverse the map without crashing (where possible), but without explicit optimisation to minimise the lap times.

Tuning parameters for the Farthest Point FTG are not present, as the algorithm has proven to be unsuitable for racing. The vehicle crashed every experiment regardless of the tuning. This is because the connection between the algorithm's reference point selection and the associated Point Follower MPC is fundamentally wrong. For example, as it can be seen in screenshot 5.3, when the vehicle gets close to the wall with the farthest point from the car being alongside that wall, the vehicle is not capable of turning away from it.

This is because the car only attempts to follow whichever point is farthest from it, without considering any ”drive-away” safety radius.

Parameter	Value
Horizon length	6
Time step	0.027
Safety radius	5
LiDAR scan index	350

Parameter	Value
Horizon length	6
Time step	0.027
Safety radius	4
LiDAR scan index	350
Distance limit	9
Default LiDAR range	1
Default LiDAR scan index	135

Table 5.1: Initial parameters for Middle Point FTG (left) and Halves FTG (right)

In other words, it’s not attempting to get away from obstacles when it’s already close to them.

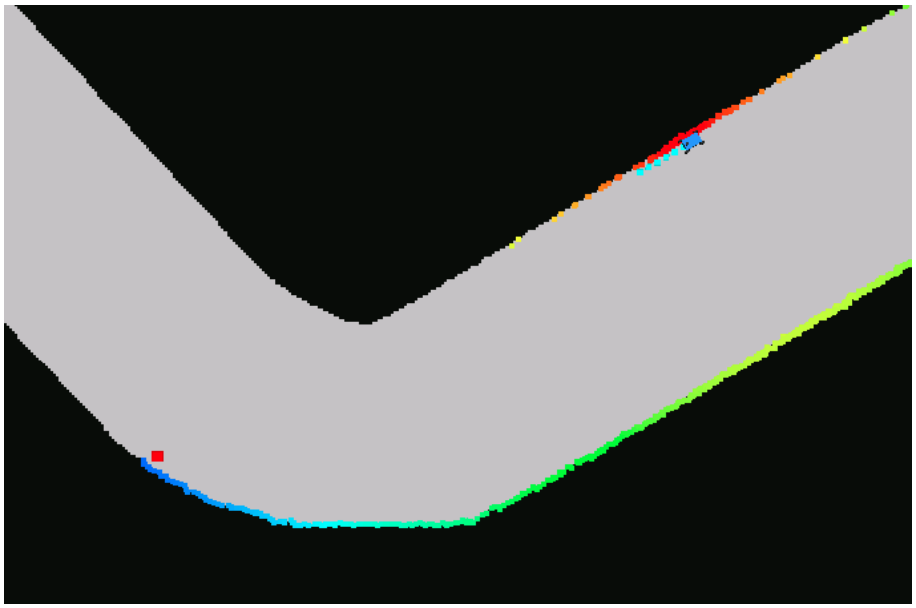


Figure 5.3: Farthest Point FTG issue

This behaviour has been confirmed on multiple racing tracks, always resulting in crashes, which renders the algorithm unusable in its current form. For instance, the combinations 5.2 of tuning parameters have been tried out on the *Vegas* racing track. More specifically, while keeping the default values, the tuning parameters get replaced one at a time (the values go back to default after each experiment), resulting in 16 experiments in this case.

Parameter	Value
Horizon length	5
Time step	0.03
Safety radius	2
Lidar scan index	350

Horizon length	Time step	Safety radius	Lidar scan index
3	0.03	$\sqrt{2}$	100
5	0.1	2	200
10	0.5	4	350
20	2	8	500

Table 5.2: Default parameters (left) and attempted tuning (right)

In order to mitigate the issue of the target reference point being too close to the walls, the Halves FTG algorithm has been designed and implemented. It battles the problem by running Farthest Point FTG for each side and always combining the results, so that the final point is closer to the middle of the track, and therefore farther away from the walls. As mentioned in the earlier 3.3.3 section, in case of failing to detect reasonable points for either of the halves, a default point relative to the vehicle is chosen, to keep the approach of combining the halves into a single target reference point always valid.

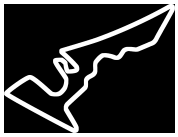
Going back to the initial comparison, the results for the Middle Point FTG and Halves FTG are provided in the table 5.3. While it is clear that the Halves FTG outperforms the Middle Point FTG, the difference is only 0.74 seconds on average, which could be subject to the tuning parameters. Therefore, further results obtained by running the algorithms with different tuning parameters and on different racing tracks are required, before providing a judgment on the control methods.

Middle Point FTG	Halves FTG
67.17	66.56
66.42	65.67
66.43	65.63
66.42	65.65
66.44	65.74
66.43	65.63

Table 5.3: Lap times

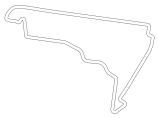
5.3 Further comparison

This section attempts to compare the results of each algorithm on a selection of racing maps. Note that even though the same tuning parameters may be used in two distinct experiments, the results are likely to vary, for example due to how burdened the machine was at a time, and how quickly the calculations could be performed. More specifically, *Vegas* and three racing tracks selected arbitrarily are used to evaluate the performance and safety (Figure 5.4).



(a) Vegas

(b) Catalunya



(c) Mexico city

(d) Shanghai

Figure 5.4: Selected racing tracks

5.3.1 Middle Point FTG

To begin with, Middle Point FTG is evaluated on *Vegas* and *Catalunya* racing maps. Similarly to trying out Farthest Point FTG parameters in the earlier section, each experiment will replace a selected value one step at a time, keeping all other parameters at their default. This will provide a reasonable discussion point for understanding how changing the parameters modifies the car's behaviour (in terms of the performance and safety).

Parameter	value
Horizon length	5
Time step	0.03
Safety radius	5
Lidar scan index	350

Parameter	value
Horizon length	5
Time step	0.03
Safety radius	3
Lidar scan index	350

Table 5.4: Middle Point FTG - default parameters; Vegas (left) and Catalunya (right)

Parameter value	Vegas lap time	Catalunya lap time
3	67.24	60.99
5	67.21	60.96
10	67.26	61.00
20	67.11	61.36

Table 5.5: Middle Point FTG - horizon length

Parameter value	Vegas lap time	Catalunya lap time
0.03	67.21	61.03
0.075	67.95	62.46
0.1	68.44	63.33
0.2	Crash	Crash

Table 5.6: Middle Point FTG - time step

Parameter value	Vegas lap time	Catalunya lap time
2	Crash	Crash
3	72.00	60.96
4	68.68	60.01
6	65.98	Crash

Table 5.7: Middle Point FTG - safety radius

Parameter value	Vegas lap time	Catalunya lap time
100	Crash	Crash
200	67.18	60.96
350	67.22	60.94
500	Crash	Crash

Table 5.8: Middle Point FTG - LiDAR scan index

The above results suggest that adjusting *Horizon length* only affects the performance, and to a small degree (worst and best times on both maps differ by less than a second). Then, *Time step* has high crash rates when it's too large, and adjusting it heavily modifies the performance of the car.

Continuing, *Safety radius* seems to have similar significance performance-wise but also results in crashes when the value is too low. Finally, *LiDAR scan index* has a safe boundary within which it does not affect the performance much, but outside of the boundary, the vehicle is likely to crash. To summarise, *Time step* and *Safety radius* affect the performance of the race by a much higher degree then the other parameters, and therefore most focus should be spent on them when fine-tuning the behaviour. The results for the other racing tracks (*Mexico city* and *Shanghai*) are not included due to the vehicle crashing regardless of tuning, and are further discussed in the 5.3.3 section.

5.3.2 Halves FTG

Halves FTG had similar issues as the other algorithm with the low-width racing tracks. In particular, no tuning could be found such that the algorithm could safely traverse any of them. Therefore, only the results for the *Vegas* racing track are provided.

Parameter	value
Horizon length	5
Time step	0.03
Safety radius	5
Lidar scan index	350
Distance limit	9
Default scan index	135
Default scan range	1

Table 5.9: Halves FTG - default parameters

Parameter value	Vegas lap time
3	66.49
5	66.45
10	66.51
20	66.65

Table 5.10: Halves FTG - horizon length

Parameter value	Vegas lap time
3	66.45
5	70.78
10	Crash
20	Crash

Table 5.11: Halves FTG - time step

Parameter value	Vegas lap time
$\sqrt{2}$	66.09
2	66.07
3	66.43
4	66.51

Table 5.12: Halves FTG - safety radius

Parameter value	Vegas lap time
100	Crash
200	66.07
350	66.46
500	Crash

Table 5.13: Halves FTG - LiDAR scan index

Parameter value	Vegas lap time
8	67.88
9	66.37
10	66.15
11	66.21

Table 5.14: Halves FTG - distance limit

Parameter value	Vegas lap time
0.5	66.73
1	66.49
2	Crash
3	Crash

Table 5.15: Halves FTG - default LiDAR scan range

Parameter value	Vegas lap time
100	66.62
200	66.19
300	66.11
400	66.30

Table 5.16: Halves FTG - default LiDAR scan index

Conclusions similar to the previous control method, regarding shared tuning parameters, can be deduced. In addition, differences in the *Distance limit* tuning affect the performance significantly, whereas *Default LiDAR scan index* less, and *Default LiDAR scan range* almost does not affect it at all. It is important to highlight that the lastly mentioned parameter has important safety implications, resulting in crashes when the values gets higher than 1.

5.3.3 Issues

Most generally, running the algorithms within all maps different to *Vegas* has been much more troublesome, as the track's width is much smaller. There is less space for the car to turn, and the track's curvature often resembles the obstacles present in the earlier section (refer to 5.4 for more details). In particular, there are two specific cases in which the algorithm's safety assurance is unsatisfactory, and crashes occur regardless of tuning. The first one is when there are multiple, consecutive turns at a near-90-degrees angles (Figure 5.5). In such cases the track curvature resembles side obstacles, as each corner heavily "goes into" the middle of the overall direction (see 5.4), where the direction is starting at the beginning of the track segment on the screenshot (bottom edge) and ending at the end of that segment (right edge).

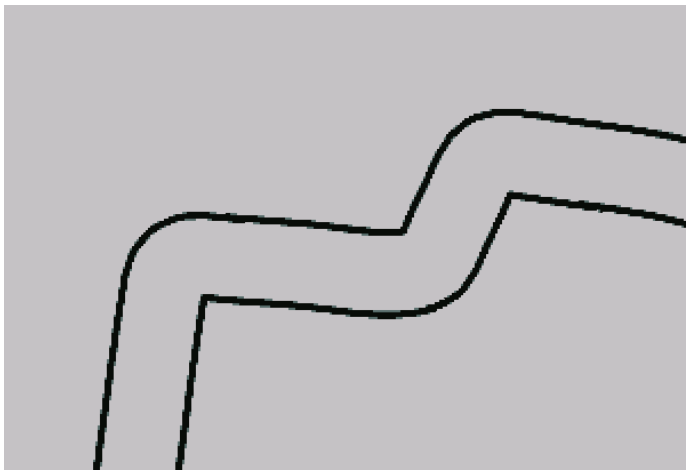


Figure 5.5: Consecutive turns at around 90 degrees angles

The other one is when there exists an extremely sharp U-turn, in which case the vehicle fails to detect where should it navigate towards (regardless of the *LiDAR scan index* parameter's tuning). In particular, the sharper U-turn on *Shanghai* map (Figure 5.6) was too much of an obstacle for the algorithms.

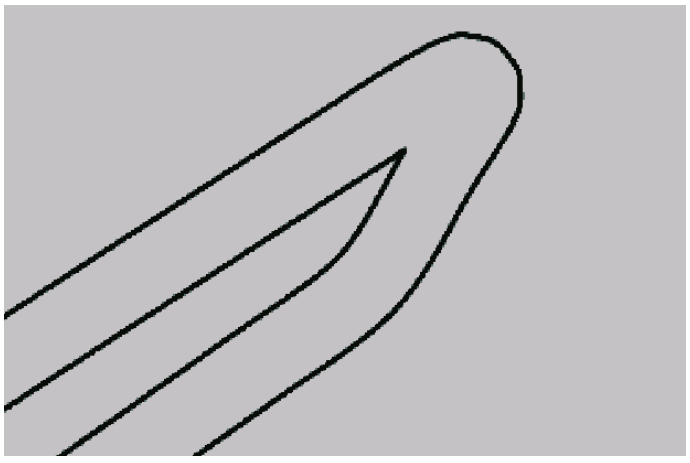


Figure 5.6: Sharp U-turn

Interestingly, on the contrary, the vehicle managed to navigate the rounder courner on that map (using Middle Point FTG), which suggests that in this case turn curvature, rather than the width of the track in general, is the culprit.

Finally, in addition to both of the previously mentioned issues, the simplicity of the cost function proved to be problematic in the low-width racing tracks. Since the MPC optimiser would always attempt to minimise the distance to the target, sometimes it would try to adjust the control inputs such that the vehicle would do a hard turn and follow a circular trajectory (to end up exactly in the target point), rather than overshoot the reference point a little. More verbose cost functions could potentially mitigate this issue resulting in the algorithm navigating the tracks more successfully.

5.3.4 Final comparison

The parameters resulting in the best lap times are combined into the following tunings:

Parameter	Value
Horizon length	5
Time step	0.03
Safety radius	6
LiDAR scan index	200

Parameter	Value
Horizon length	5
Time step	0.03
Safety radius	2
LiDAR scan index	200
Distance limit	10
Default LiDAR range	1
Default LiDAR scan index	300

Table 5.17: Tuned parameters for Middle Point FTG (left) and Halves FTG (right)

The lap times are shorter by 1.23 on average for the Middle Point FTG algorithm, and 0.40 for the Halves FTG, when compared to the initial results (Table 5.3). While in both cases the best measured times were better than any of the times achieved when experimenting with the tuning parameters, the improvement for the Middle Point FTG algorithm is three times greater than for the Halves FTG. In this case, Middle Point FTG has outperformed Halves FTG, which is different to how it was in the initial comparison.

Middle Point FTG	Halves FTG
65.91	66.02
65.17	65.23
65.13	65.37
65.19	65.25
65.13	65.26
65.11	65.36

Table 5.18: Lap times

5.3.5 Summary

It is safe to conclude that tuning of the parameters highly affects the results. Middle Point FTG is more versatile, as it avoids crashing on maps of low-width better, which might make it a more appealing approach. However, Halves FTG seems to be less affected by the tuning parameters, making it more stable in its behaviour. In addition, it has more tuning parameters, which results in more fine-tuning options, which in turn could potentially result in even better race times, given enough time to find the right combination of values. Moreover, while the simplicity of the MPC cost function is beneficial for the purpose of reducing the computational time, using a more complex method could result in an overall safety improvement.

To answer the hypothesis, it is clear that both the track's width and turn curvature affect safety. However, reducing the track's width drastically affects the algorithms' ability to navigate the track without crashing, whereas more curvy paths result in issues only in the case of extremely sharp U-turns. In the other cases, as long as the width of the track is of reasonable size, the vehicle seems to behave correctly and avoid crashing.

5.4 Obstacle avoidance

A brief trial against avoiding obstacles has been conducted. More specifically, it was important to understand the vehicle's behaviour in avoiding the obstructions in the middle of the track and close to the racing track's edges.

5.4.1 Experimentation

Farthest Point FTG has failed to avoid the obstacles in both cases. In particular, the vehicle would start driving towards one side of the racing track upon detecting it as the largest sequence of non-ignore LiDAR scan ranges, and then switch when that side was too close (due to the safety radius around the point closest to the car). It is especially visible when avoiding the side obstacles (Figure 5.7). See Figure 5.8 to see the result of avoiding the middle obstacle.

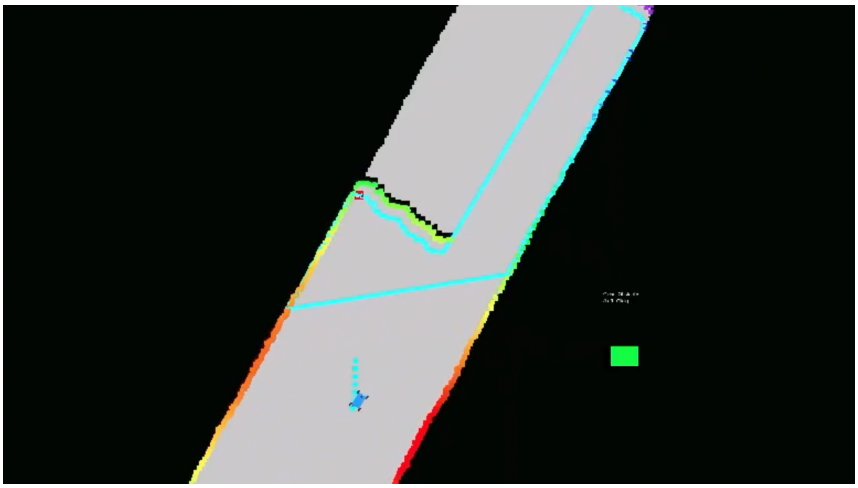


Figure 5.7: Farthest Point FTG avoiding side obstacles, notice how the farthest point is on the left hand side rather on the right hand side (due to an overall bigger gap on the left)

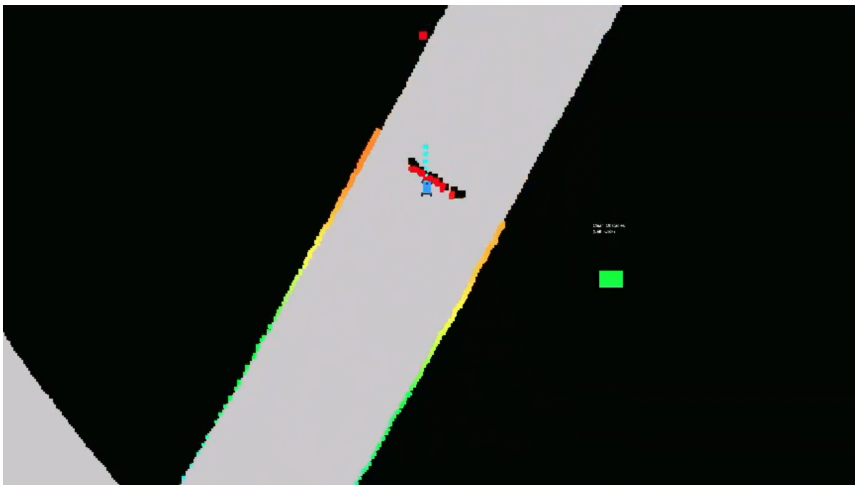


Figure 5.8: Farthest Point FTG avoiding a middle obstacle

Halves FTG had a moderate success when avoiding side obstacles. However, when the obstructions where outgoing from the walls too far, the car would still fail to avoid them (Figure 5.9). Furthermore, the algorithm is completely incapable of steering away from the obstacles in the middle (Figure 5.10), since, upon detecting reference points to the left and to the right of it, the final target point is a combination of both, often being right in the middle.

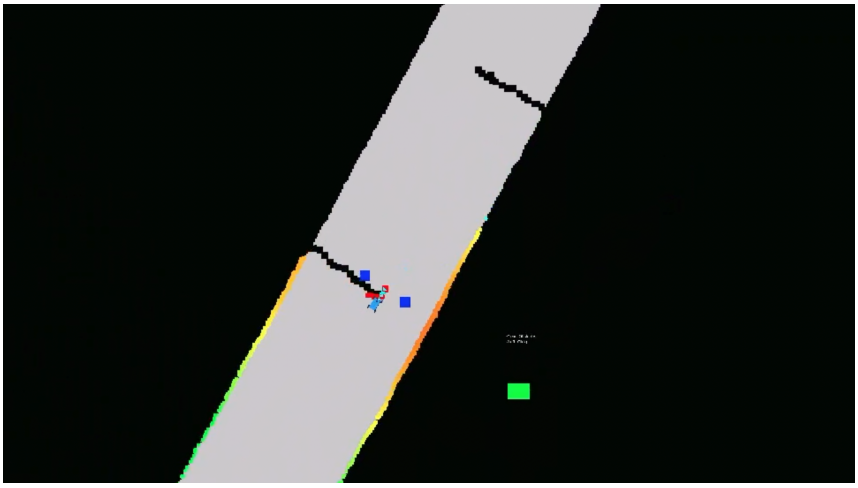


Figure 5.9: Halves FTG avoiding side obstacles

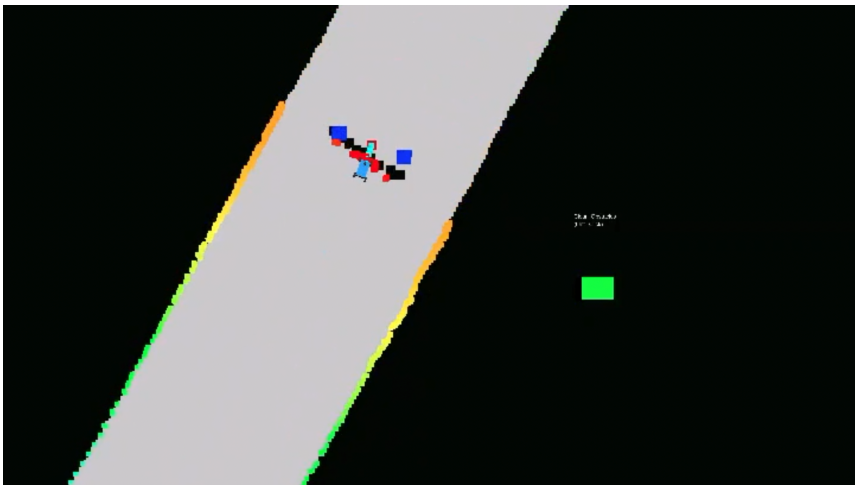


Figure 5.10: Halves FTG avoiding a middle obstacle

Middle Point FTG proved to be the most versatile when it comes to avoiding obstacles. It successfully avoided the side obstacles (Figure 5.11) and nearly passed the middle obstruction (Figure 5.12). The more space between each of the side obstacles and the walls there was, the more seamlessly the entire process occurred.

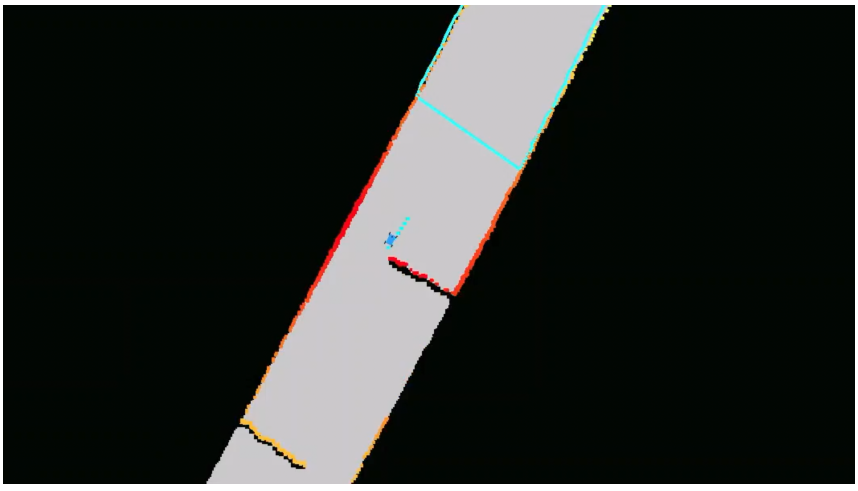


Figure 5.11: Middle Point FTG avoiding side obstacles

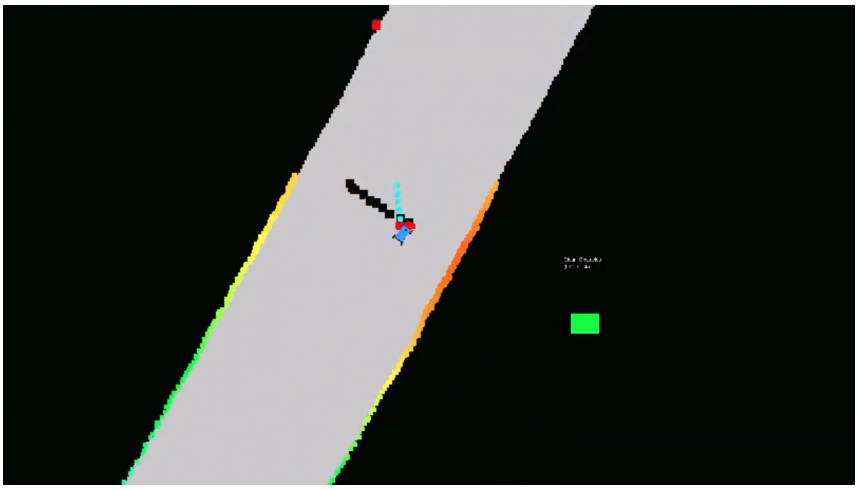


Figure 5.12: Middle Point FTG avoiding a middle obstacle

5.4.2 Attempted collision avoidance using MPC constraints

Based on the results so far, it is clear that the algorithms are not performing well when avoiding obstacles present in the racing track. Therefore, an alternative approach utilising MPC constraints was attempted, by expressing the LiDAR scan area as a polygon. More specifically, each wall point detected by the scan was considered a vertex and, since there are 1080 points available, the shape was initially simplified to offload the computational burden.

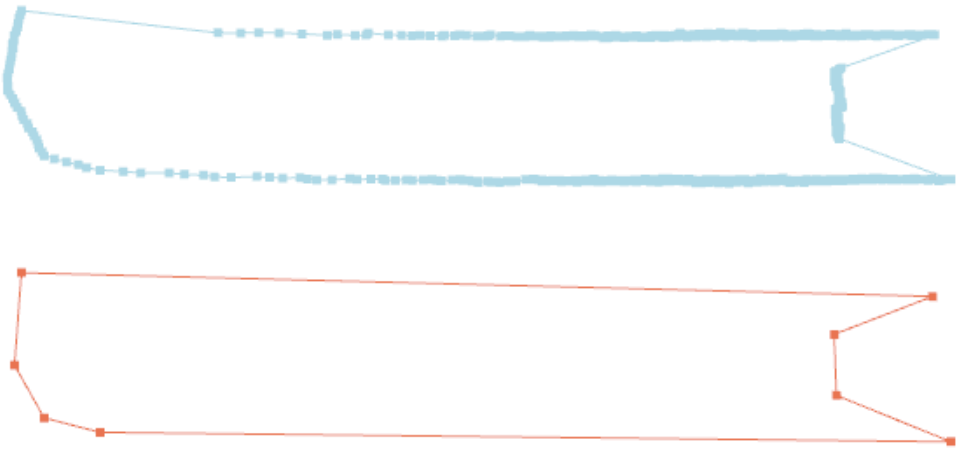


Figure 5.13: Initial and simplified LiDAR scans

In the event of expressing such a shape as a collection of non-linear constraints, MPC would be capable of keeping the vehicle within the specified region. Describing convex shapes (where a shape is convex when all of its interior angles are less than or equal to 180 degrees) in this format is possible and generally considered as a rather simple task, and it is known that for every convex polygon there exists a system of inequalities describing it. For example, consider a triangle with vertices at coordinates $(0, 0)$, $(1, 3)$, and $(2, 1)$. Then the associated inequalities describing it are:

$$-3x + y \leq 0$$
$$\frac{1}{2}x - y \leq 0$$
$$-2x - y + 5 \leq 0$$

Unfortunately, there is currently no method to represent any non-convex shape this way. In particular, in Figure 5.13, the ”dent” on the right-hand side causes the shape to be non-convex. LiDAR scans have a tendency to be of a non-convex shape, especially when detecting obstacles in the middle of the racing track. Therefore, it is infeasible to implement collision avoidance this way.

Chapter 6

Conclusions

This chapter concludes the thesis by providing a very brief overview of what was covered and answering the objectives stated in the introduction. In addition, a discussion on further work has been included.

6.1 Summing up

This thesis has addressed the issue of autonomous racing, by investigating safety assurance and performance of three control algorithms, with the help of the F1TENTH simulator. In this dissertation, MPC has been used to drive the vehicle towards target reference points created by the variations of the FTG algorithm and the results have been presented and discussed. Most generally, Farthest Point FTG proved to be unsuitable in its basic form, whereas Middle Point FTG outperformed Halves FTG slightly, especially in terms of safety assurance.

Addressing the aim of this thesis, motion planning with FTG and MPC has been investigated. Two out of three algorithms were capable of navigating *Vegas* racing track without issues but struggled with traversing the low-width maps. Then, since Farthest Point FTG did not work in its basic form, it had to modified to work with MPC. More specifically, Halves FTG was created to battle the associated issue.

Continuing, lower width of the race track had greatly affected safety assurance, to the point that no tuning combinations could be found to avoid crashing. On the other hand, track curvature caused significant issues only in the extreme case of a sharp U-turn. Finally, while every tuning parameter affected the algorithms' behaviour to a larger or smaller degree, the most influential were the *Time step* and *Safety radius*.

Addressing the objectives of this dissertation:

- A collection of three algorithms utilising FTG and MPC has been implemented.
- Safety of the algorithms has been evaluated, and the reasoning for vehicle crashes has been explained.
- Several parameters were evaluated in terms of performance, improving the initial tuning of the algorithms.
- Graphical visualisations of the algorithms and explanations of the problems have been provided, in addition to extensive usage of drawing components within the F1TENTH simulator.
- An open-source software package has been shared on GitHub.

6.2 Future work

There are three potential research areas not covered by this thesis, which, if investigated, could result in the overall improvement of autonomous racing algorithms.

Firstly, collision avoidance attempted in this thesis has failed due to the lack of algorithms capable of describing geometrical shapes with systems of inequalities. More specifically, if there was a way to represent non-convex shapes with such systems, the MPC formulation could include their representations in its constraints definitions, as safety regions. This should result in a guarantee of safety when controlling the vehicle. Alternatively, a possible investigation could determine some other methods to describe LiDAR scans with systems of inequalities.

Then, while the distance-based MPC cost function used in this project was satisfactory in terms of the trade-off between optimality and complexity, a more verbose computation could result in better safety assurance. While performance is indeed necessary, the race will be over as soon as the vehicle crashes, in which case it is better to ensure safety first.

Finally, while the FTG algorithms were quick to implement and easy to understand, they only provided a relatively small amount of information. Rather than having a single target reference point, it would be more beneficial to derive a collection of such points for the vehicle to follow. By generating locally-optimal trajectories based on LiDAR scans only, plenty of existing MPC formulations could be applied to further minimise the resulting lap times, and to ensure the safety of the reference paths.

Bibliography

[1] D. P. Watson and D. H. Scheidt, "Autonomous systems," *Johns Hopkins APL technical digest*, vol. 26, no. 4, pp. 368–376, 2005.

[2] J. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th International Conference on software engineering*, ICSE '02, (New York NY), pp. 547–550, ACM, 2002.

[3] A. Liniger and J. Lygeros, "A noncooperative game approach to autonomous racing," *IEEE Transactions on Control Systems Technology*, vol. 28, no. 3, pp. 884–897, 2020.

[4] E. Alcalá, V. Puig, J. Quevedo, and U. Rosolia, "Autonomous racing using linear parameter varying-model predictive control (lpv-mpc)," *Control Engineering Practice*, vol. 95, p. 104270, 2020.

[5] J. Hu, S. Xiong, J. Zha, and C. Fu, "Lane detection and trajectory tracking control of autonomous vehicle based on model predictive control," *International journal of automotive technology*, vol. 21, no. 2, pp. 285–295, 2020.

[6] U. C. H. F. team, "F1/10 autonomous racing - montreal grand prix winning team." `https://youtu.be/ctTJHueaTcY`, 2019.

[7] M. Brunner, U. Rosolia, J. Gonzales, and F. Borrelli, "Repetitive learning model predictive control: An autonomous racing example," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 2545–2550, 2017.

[8] M. Morari and J. H. Lee, "Model predictive control: past, present and future," *Computers Chemical Engineering*, vol. 23, no. 4, pp. 667–682, 1999.

[9] V. Sezer and M. Gokasan, "A novel obstacle avoidance algorithm: "follow the gap method"," *Robotics and Autonomous Systems*, vol. 60, no. 9, pp. 1123–1134, 2012.

[10] J.-C. Latombe, *Robot motion planning*, vol. 124. Springer Science & Business Media, 2012.

[11] M. Lepetič, G. Klančar, I. Škrjanc, D. Matko, and B. Potočnik, "Time optimal path planning considering acceleration limits," *Robotics and Autonomous Systems*, vol. 45, no. 3, pp. 199–210, 2003.

[12] S. G. Tzafestas, "11 - mobile robot path, motion, and task planning," in *Introduction to Mobile Robot Control* (S. G. Tzafestas, ed.), pp. 429–478, Oxford: Elsevier, 2014.

[13] M. Althoff, M. Koschi, and S. Manzinger, "Commonroad: Composable benchmarks for motion planning on roads," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 719–726, 2017.

[14] J. Liu, P. Jayakumar, J. L. Stein, and T. Ersal, *A Nonlinear Model Predictive Control Algorithm for Obstacle Avoidance in Autonomous Ground Vehicles within Unknown Environments*. 2015.

[15] K. Florianski, "f1tenth-racing-algorithms." `https://github.com/TheCodeSummoner/f1tenth-racing-algorithms`, 2021.

[16] S. A. I. L. et al., "Stanford artificial intelligence laboratory et al.." `https://www.ros.org`, 2018.

[17] Y. Li and J. Ibanez-Guzman, "Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems," *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, 2020.

[18] F1TENTH, "F1tenth racing car (image)." `https://f1tenth.readthedocs.io/en/stable/_images/f1tenth_NX.png`, 2021.

[19] M. Himmelsbach, T. L. A. Mueller, and H. J. Wuensche, "Lidar-based 3d object perception," in *Proceedings of 1st International Workshop on Cognition for Technical Systems*, 2008.

[20] M. Sualeh and G.-W. Kim, "Visual-lidar based 3d object detection and tracking for embedded systems," *IEEE Access*, vol. 8, pp. 156285–156298, 2020.

[21] F. Chenavier and J. Crowley, "Position estimation for a mobile robot using vision and odometry," in *Proceedings 1992 IEEE International Conference on Robotics and Automation*, pp. 2588–2593 vol.3, 1992.

[22] K. S. Chong and L. Kleeman, "Accurate odometry and error modelling for a mobile robot," in *Proceedings of International Conference on Robotics and Automation*, vol. 4, pp. 2783–2788 vol.4, 1997.

[23] F1TENTH, "F1tenth simulator." `https://github.com/f1tenth/f1tenth_simulator`, 2021.

[24] A. Jain, M. O'Kelly, P. Chaudhari, and M. Morari, "BayesRace: Learning to race autonomously using prior experience," in *Proceedings of the 4th Conference on Robot Learning (CoRL)*, 2020.

[25] A. Sinha, M. O'Kelly, H. Zheng, R. Mangharam, J. Duchi, and R. Tedrake, "Formulazero: Distributionally robust online adaptation via offline population synthesis," *arXiv preprint arXiv:2003.03900*, 2020.

[26] M. O'Kelly, H. Zheng, A. Jain, J. Auckley, K. Luong, and R. Mangharam, "Tunercar: A superoptimization toolchain for autonomous racing," 2020.

[27] C. A. R. M. A. (CARMA), "Carma 1tenth." `https://github.com/usdot-fhwa-stol/carma-1-tenth`, 2020.

[28] R. Rajkumar, M. Gagliardi, and L. Sha, "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation," in *Proceedings Real-Time Technology and Applications Symposium*, pp. 66–75, 1995.

[29] F1TENTH, "F1tenth simulator's simplified graph of nodes." `https://f1tenth.readthedocs.io/en/latest/_images/sim_graph_public1.png`, 2021.

[30] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, 03 2014.

[31] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33–38, 1998.

[32] M. Farah, "yq." `https://github.com/mikefarah/yq`, 2021.

[33] T. Preston-Werner, P. J. H. C. Wanstrath, and S. Chacon, "Github." https://github.com/, 2021.

[34] S. Lucia, A. Tătulea-Codrean, C. Schoppmeyer, and S. Engell, "Rapid development of modular and sustainable nonlinear model predictive control solutions," *Control engineering practice*, vol. 60, pp. 51–62, 2017.

[35] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

[36] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[37] P. V. Craven, "Python arcade." https://github.com/pythonarcade/arcade, 2021.

[38] P. Norouzi, C. H. H. Zheng, and Y. Wu, "Racing using sqn reinforcement learning." https://github.com/pnorouzi/rl-path-racing, 2020.

[39] H. Z. J. Betz and M. Ellerbach, "F1tenth racing tracks." https://github.com/f1tenth/f1tenth_racetracks, 2020.