Motorvehicle University of Emilia-Romagna

Master Degree in Advanced Automotive Electronic Engineering

MPC for Dynamic Obstacle Avoidance

Technical report on the group project activity

Course of Compliance Design of Automotive Systems

Group memebers:
Gianvincenzo Daddabbo
Gaetano Gallo
Alberto Ruggeri
Martina Tedesco
Alessandro Toschi

a.y. 2020-2021

Abstract

Every year 1.25 million people die and as many as 50 million are injured in road traffic accidents worldwide, according to United Nations statistics. Human error is involved in about 95% of all road traffic accidents in the EU, and in 2017 alone, 25300 people died on the Union's roads [1]. Autonomous cars can improve road safety and through new technologies it is also possible to reduce traffic congestion and CO2 emissions.

The aim of this report is to present the steps that we have followed as a group in order to implement a system for dynamic obstacle avoidance using an adaptive Model Predictive Control. In the simulated environment, the vehicle model is expected to encounter and safely avoid the dynamic obstacles along its way. On the other hand, when an obstacle is not present ahead of the autonomous vehicle, it is expected to follow a predetermined path.

Contents

List of Figures

List of Tables

1 Introduction

The task of avoiding obstacles is one of the key issues when it comes to the vehicular scenario and it is more difficult to perform it here than it is in static environments. A vehicle with an obstacle avoidance system is equipped with sensors that measure the distance between the car itself and the obstacle in the same lane. If an autonomous car encounters an obstacle, it is expected to move temporarily to another lane and move back to the original lane once it has driven past the obstacle. This type of control can be implemented using a Model Predictive Control (MPC). An MPC is an advanced control method that works in discrete time and uses a model of the system to make predictions about the system's future behavior. MPC solves an online optimization algorithm to find the optimal control action that drives the predicted output to the reference [2]. In other words, from a set of state values, and with respect to a model, it optimizes a problem around an objective and gives a sequence of control signals as outputs. The first set of control values are then used as inputs to the system plant, and after a short period, set as the *system time step*, the new state values are measured and the process is repeated. The overall objectives of the MPC are:

- prevent that input and output constraints are violated;
- optimize some input variables, while other outputs are kept in specified ranges,
- prevent the input variables from having excessive variations.

Figure 1 shows a block diagram for a Model Predictive Control.

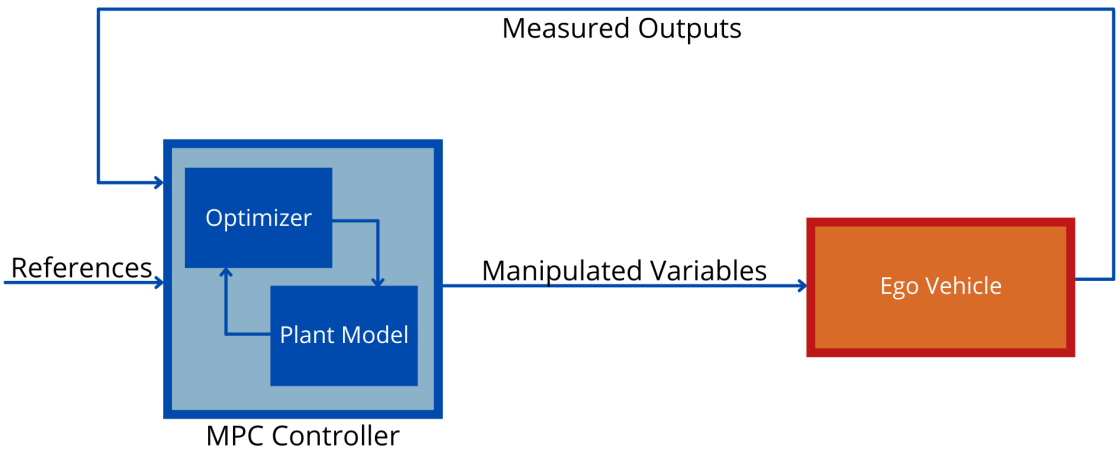


Figure 1: MPC Block Diagram

An MPC controller has two main functional blocks: the optimizer and the plant model. The dynamic optimizer allows to find the optimal input that gives the minimum value of the cost function taking into account all the constraints. Generally, a non linear model is used for the validation of the controller, while the plant model used for the MPC is a linearized version of the actual plant.

Our project is based on the use of an *adaptive* MPC which means that the plant state has to be measured again to be adopted as the linearization point for the next step of the predictive control. When the plant state is re-sampled, the whole process computes again the calculations starting from the new current state.

1.1 Model-Based Design

When developing a project, especially concerning embedded systems, it is crucial to follow a process model which illustrates the high-level activities and their phasing during development.
As stated in [3] *"Process models provide high-level perspective that helps team members understand what activities to do and what progress has been made on each of those development activities"*.
The development process of our system is based on the V-model shown in Figure 2. Starting from the general V-model we have tailored our own V-model in order to meet our needs; doing so we have managed to skip some unnecessary steps in the development process while still being compliant with the definition of the V-model itself.

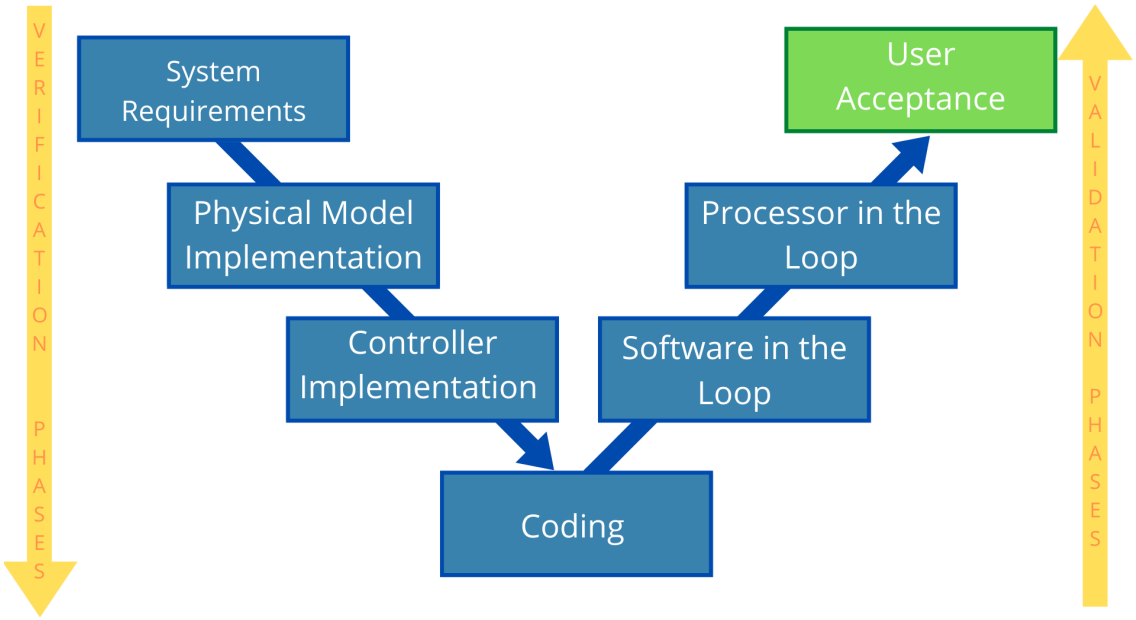


Figure 2: System Development V-model

As shown above the V-model is a representation of system development process that highlights verification steps on one side and validation steps on the other of the 'V'. The left side of the 'V' identifies the verification phase, containing the steps that lead to code generation while the right side identifies the validation phase which ends with the user acceptance. Below a brief description of the steps which constitute our V-model:

- **System requirements**: this is the first step of each project, it consists in the requirements gathering from the customer;
- **Physical model implementation**: this phase contains the system design of our vehicle in terms of sensors required and dynamic and kinematic model;

- **Controller implementation**: this phase is the focus of our project, designing an adaptive MPC.
- **Coding**: exploiting some specific tools, we can automatically generate code for embedded deployment and create test benches for system verification, saving time and avoiding the introduction of manually coded errors;
- **Software in the Loop**: once generated, the code is tested in a simulation environment;
- **Processor in the Loop**: during this phase the code is uploaded on a demo board and than it is tested again;
- **User Acceptance**: in this phase the final product is tested in order to verify the compliance with the initial costumer requirements.

1.2 Required tools

The project is based on MATLAB, a proprietary multi-paradigm programming language and numeric computing environment developed by *MathWorks*, and Simulink, a MATLAB-based block diagram environment for multi-domain simulation and Model-Based Design. The following list represents the software requirements for the project, including all the applications needed for the proper execution and test:

- **MATLAB R2019a or newer**;
    1. **Curve Fitting Toolbox**: needed to make the reference map gentler;
    2. **Automated Driving Toolbox**: needed to get a reference map from real world data;
    3. **Model Predictive Control Toolbox**: needed to have all the useful resources to develop the MPC;
- **Simulink 9.3 or higher**;
    1. **Simulink Test**: used to perform automated tests and requirements verification;
    2. **Embedded Coder**: used for automatic code generation.

2 System Requirements

Defining the project requirements is an essential and crucial step to accomplish in the starting phase of the project. Indeed, from both the perspective of the customer and of the project developers, detailed requirements are necessary to deliver exactly what is needed.
Firstly, the requirements are defined by the customer at a high level, then "translated" by the developers to a lower and technical level and later on validated to ensure that the project requirements are correct, free of defects/bugs, and meets the needs of the users.
For the sake of this project we have assumed to receive the requirements from a hypothetical customer. Namely, there are six high level requirements given for this obstacle avoidance implementation that are mainly related to the accuracy of the system and to the driver comfort:

1. Maximum lateral error from reference of $0.75m$;
2. Detection of obstacles within $100m$ ahead of vehicle;
3. Move on left lane within a predetermined safe zone from the obstacle[1];
4. Once passed the obstacle, come back on right lane with no less than $10m$ but no more than $50m$ ahead of it;
5. Maximum lateral acceleration of $2m/s^2$;
6. All previous requirements satisfied in speed range from $10km/h$ to $100km/h$.

[1] Third and fourth requirement are requested when an obstacle is detected

3 System Implementation

As briefly discussed in the *Introduction* chapter, nowadays one of the smartest techniques deployed for autonomous vehicles control is the Model Predictive Control, hence our decision to develop a controller in order to accomplish the path following and obstacle avoidance task. The design of such a controller needs a deep analysis of the project requirements and a smart system partitioning to identify different tasks that can be developed autonomously and independently from others to be then merged in order to satisfy the control problem.

3.1 Analysis of requirements

Starting from the detailed system requirements described in Section 2, we can obtain low level requirements suitable for the controller implementation; each of these is associated with a draft of the validation method needed to assert each of them. Table 1 shows the system requirements, low level requirements and the corresponding validation methods.

System requirements	Low level requirements	Validation methods
Maximum lateral error from reference of $0.75m$	Inequality constraint: $e_y = abs(Y_{reference} - Y_{vehicle}) <= 0.75m$	Check in each point that the difference between the actual vehicle position and the reference path is not greater than $0.75m$
Detection of obstacles within 100 m ahead of vehicle	Sensor fusion able to provide useful data in the range $0 - 100m$ from the vehicle	Drive toward a static obstacle and confirm that is detected when is $100m$ far
Move on left lane within a predetermined safe zone from the obstacle	Inequality constraint when obstacle detected: $dist = position_{vehicle} - position_{obstacle} \in SafeZone$	Confirm that when changing lane the distance from the obstacle measured by the sensors is greater than the predetermined safe zone
Once passed the obstacle, come back on right lane with no less than $10m$ but no more than $50m$ ahead of it	Inequality constraint when obstacle passed: $10m <= dist <= 50m$	Confirm that when changing lane the distance from the obstacle measured by the sensors is greater than the predetermined safe zone
Maximum lateral acceleration of $2.0m/s^2$	Constraint on Model Predictive Control input: $a.max = 2.0$	Check that the maximum value registered by the IMU−accelerometer is not greater than $2.0m/s^2$
All previous requirements satisfied in speed range from $10km/h$ to $100km/h$	Verify all previous constraint when simulating at speed range $10 - 100km/h$	Verify all the above methods with vehicle travelling from $10km/h$ to $100km/h$

Table 1: System requirements and possible validation methods

3.2 Partitioning

Regarding the system partitioning, which as already said in Section 3 is a crucial part of the implementation of a system, we have decided to base the implementation of our

project on the tasks shown in Figure 3.

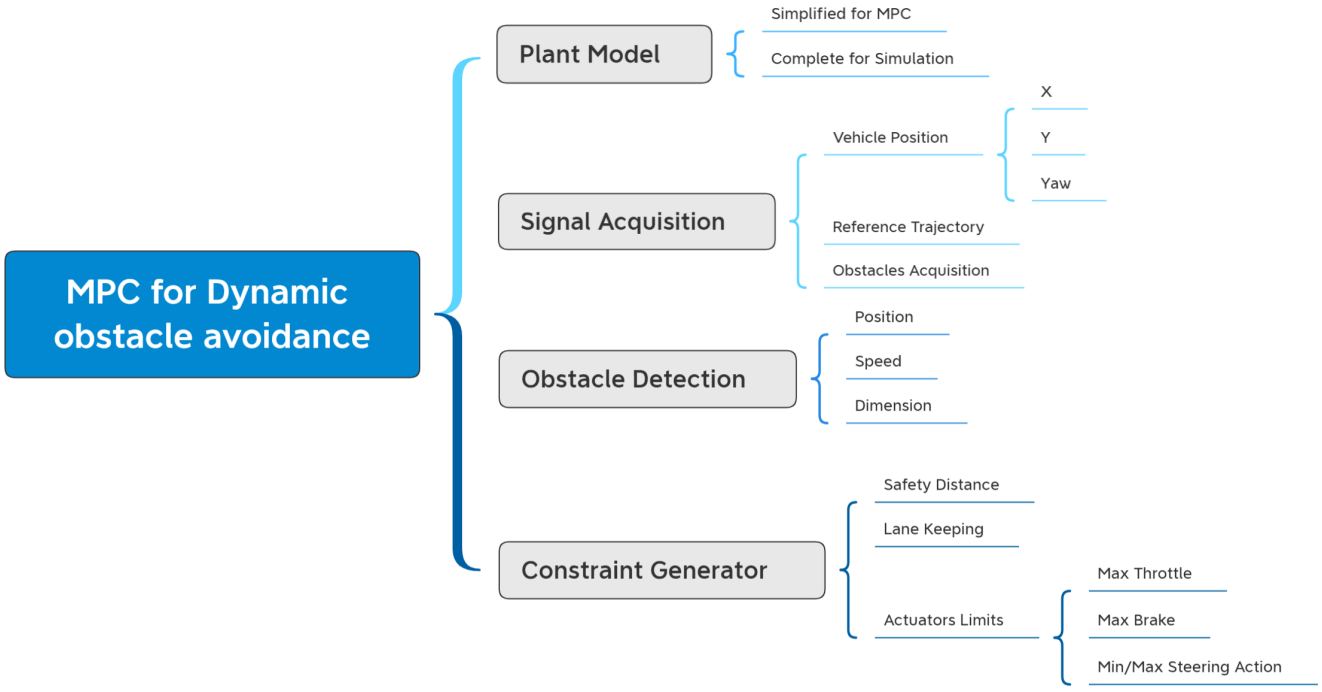


Figure 3: System Partitioning

As shown in the figure above, our project can be divided into four macro-areas:

- Plant Model: is the mathematical model of the vehicle which represent the development environment of our project. We have decided to adopt two models, a simplified one to be used in the MPC and a more accurate one to be used in the simulation;
- Signal Acquisition: refers to the gathering of data related to the vehicle position, trajectory, and obstacle acquisition, all coming from the sensors mounted on our vehicle;
- Obstacle Detection: here the properties of the obstacle such as position, speed, and dimension are evaluated;
- Constraint Generator: in this phase we set the constraints which the MPC needs to comply with. These constraints are necessary in order to make our model as realistic and safe as possible.

These sections will be described in depth in the following chapters.

4 Plant Model

The starting point for Model Based design is to develop and implement a plant model, so a model representing the Physics of the system considered.
Different models are available in literature to model a vehicle. Since the target of the project is to develop a lateral controller, one of the most suitable model is the so called *Bicycle Model*.
The kinematic bicycle model is described by the following non linear system:

$$\dot{X} = V cos(\psi + \beta) \tag{1}$$

$$\dot{Y} = V sin(\psi + \beta) \tag{2}$$

$$\dot{\psi} = \frac{V cos(\beta)}{l_f + l_r} \left(tan(\delta_f) - tan(\delta_r)\right) \tag{3}$$

$$\beta = tan^{-1}\left(\frac{l_f tan(\delta_r) + l_r tan(\delta_f)}{l_f + l_r}\right) \tag{4}$$

Where:

- X and Y are the coordinates of the body of the vehicle in the global reference frame,
- ψ is the yaw (orientation angle),
- β is the slip angle of the body,
- $\beta + \psi$ is the body speed direction known as *Course Angle*,
- δ_f is the front steering angle,
- δ_r is the rear steering angle,
- V is the magnitude of the body speed,
- l_f is a geometric parameter which indicates the distance of the CoG from the front wheel,
- l_r is a geometric parameter which indicates the distance of the CoG from the rear wheel.

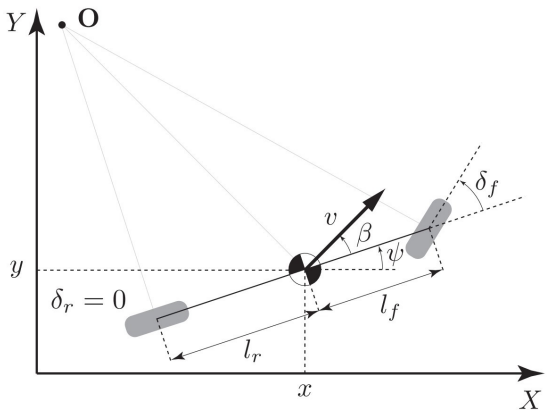


Figure 4: Bicycle kinematic vehicle model

This model can be further simplified assuming that, as in the most of commercial vehicles, only the front wheels are able to steer, so the rear steering angle δ_r is constant and equal to 0.
Standard MPC controller works on linear systems, so we decided to implement a linearized version of the Bicycle Model to feed the controller. This linearized model is described by the following State Space equations:

$$\dot{x} = Ax + Bu \tag{5}$$

$$y = Cx + Du \tag{6}$$

where:

$$x = \begin{bmatrix} X \\ Y \\ \psi \\ v \end{bmatrix} \quad A = \begin{bmatrix} 0 & 0 & -Vsin(\psi) & cos(\psi) \\ 0 & 0 & Vcos(\psi) & sin(\psi) \\ 0 & 0 & 0 & \frac{tan(\delta)}{l_r+l_f} \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & \frac{Vtan(\delta)^2+1}{l_r+l_f} \\ 1 & 0 \end{bmatrix} \tag{7}$$

$$u = \begin{bmatrix} Throttle \\ \delta \end{bmatrix} \qquad C = I^{4\times4} \qquad D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

These matrices are obtained evaluating the Taylor expansion of the non linear bicycle model, as explained in the MathWorks example [4].
A more complex vehicle model can be useful to test the performance of the controller to be developed. The *Dynamic Bicycle Model* [5] can be built starting from the previous non linear model, deriving the following second order derivative equations:

$$\ddot{x} = \dot{\psi}\dot{y} + Throttle \tag{8}$$

$$\ddot{y} = -\dot{\psi}\dot{x} + \frac{2}{m}\left(F_f cos(\delta) + F_r\right) \tag{9}$$

$$\ddot{\psi} = \frac{2}{I_z}\left(l_f F_f - l_r F_r\right) \tag{10}$$

$$\dot{X} = \dot{x}cos(\psi) - \dot{y}sin(\psi) \tag{11}$$

$$\dot{Y} = \dot{x}sin(\psi) + \dot{y}cos(\psi) \tag{12}$$

where m is the mass of the vehicle, I_z is the inertia of the vehicle with respect to the vertical axle passing for the CoG of it, while F_f and F_r are respectively the side slip forces acting on the front and rear wheels, and they can be evaluated as:

$$F_f = 2C_f\left(\delta - \theta_f\right) \tag{13}$$

$$F_r = 2C_r\left(-\theta_r\right) \tag{14}$$

with C_i side slip friction coefficient of the i-th wheel couple and θ_i side slip angle of the wheels.
Tyre side slip angles can be approximated by the following equations:

$$\theta_f = tan^{-1}\left(\frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) \tag{15}$$

$$\theta_r = tan^{-1}\left(\frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right) \tag{16}$$

The models introduced are, more or less, independent of the longitudinal dynamics of the vehicle, since they link this dynamic with a simple coefficient, that is the $Throttle$. In our model, this coefficient should be considered as an acceleration of the vehicle, and it can be both positive (driving) or negative (braking). Range of values for this parameter is dependent on lots of variables, of course, such as the engine power, the vehicle mass and inertia, the ground type, the rubber of the wheels and so on. Our assumption is to give a fixed interval for this parameter, to simulate a small commercial vehicle on dry asphalt, which can have a maximum acceleration of $4m/s^2$ and a maximum braking deceleration of $-0.8g$. To summarize:

- $Throttle \in [-7.85, 4.00]m/s^2$

Parameters considered for the bicycle model are taken from real vehicle data [6] and are reported in the following table:

ID	**Vehicle name**	**Wheel base** $[m]$	**l_r** $[m]$	**l_f** $[m]$	**Mass** $[kg]$	**Inertia** $[kg \cdot m^2]$
1	*Hyundai Azera*	2.843	1.738	1.105	1200	1000
2	*BMW 325i*	2.570	1.369	1.201	1251	2027
3	*Ford E150*	3.505	1.634	1.871	2995	6536
4	*Suzuki Samurai*	2.032	0.870	1.162	1229	1341
5	*Volkswagen Beetle*	2.408	0.996	1.412	857	1289

Table 2: Vehicle data considered for development and validation

Values reported in the table 2 are stored in a MATLAB file and can be accessed through the *loadParameters* function, which take as input the vehicle ID and returns the parameters associated with that vehicle. To avoid to call improperly this function, a default parameters set as been provided with the following values:

Vehicle name	**Wheel base** $[m]$	**l_r** $[m]$	**l_f** $[m]$	**Mass** $[kg]$	**Inertia** $[kg \cdot m^2]$
Default	2	1	1	1000	1000

We used data of vehicle 1, *Hyundai Azera*, for the development phase, where mass and inertia are not the real ones of the vehicle, but are given with realistic values, while other data of the previous table 2 are meant to be used in the validation phase, to test the controller with different vehicles. For what concern the side slip friction values, we considered two fixed values for all the vehicles that are:

- $C_f = 1.0745 \times 10^5$ N/rad
- $C_r = 1.9032 \times 10^5$ N/rad

while in the *"default"* condition they are both 10^5 N/rad.

4.1 Model comparison

As specified in the subsection 3.2 we have decided to use two models, a simplified and linearized model for the MPC and a more complex one for the simulation of the model. The former is defined as *Kinematic Bicycle Model* and the latter as *Dynamic Bicycle Model*. Hence we have decided to test both of these systems to show how they behave with different throttle and steering inputs using the *Simulink Test* tool. Thus, the tests performed are:

- **Free evolution test**: this test has been performed considering a constant steering of 0° and a constant throttle of 0 m/s^2;
- **Only throttle test**: this test has been performed keeping the steering angle constant and equal to 0° and varying the throttle as shown in Figure 5;
- **Constant steering test**: this test has been performed keeping the throttle constant and equal to 0 m/s^2 and the steering angle constant and equal to 2°;
- **Ramp steering test**: this test has been performed keeping the throttle equal to 0 m/s^2 and giving a ramp steering angle signal (varying linearly from 0° to 36°);
- **Small sinusoidal steering test**: this test has been performed keeping the throttle constant and equal to 0 m/s^2 and giving a sinusoidal steering angle signal with frequency 0.2 Hz and amplitude 5°;
- **Big sinusoidal steering test**: here we have performed the same test as before (sinusoidal steering input) but with a larger amplitude of the sine wave (15°);
- **Combined test 1**: this test has been performed keeping the steering angle constant and equal 2° and varying the throttle as shown in Figure 5;
- **Combined test 2**: this test has been performed keeping the throttle equal to 0.2 m/s^2 and giving a ramp steering angle signal (varying linearly from 0° to 36°).

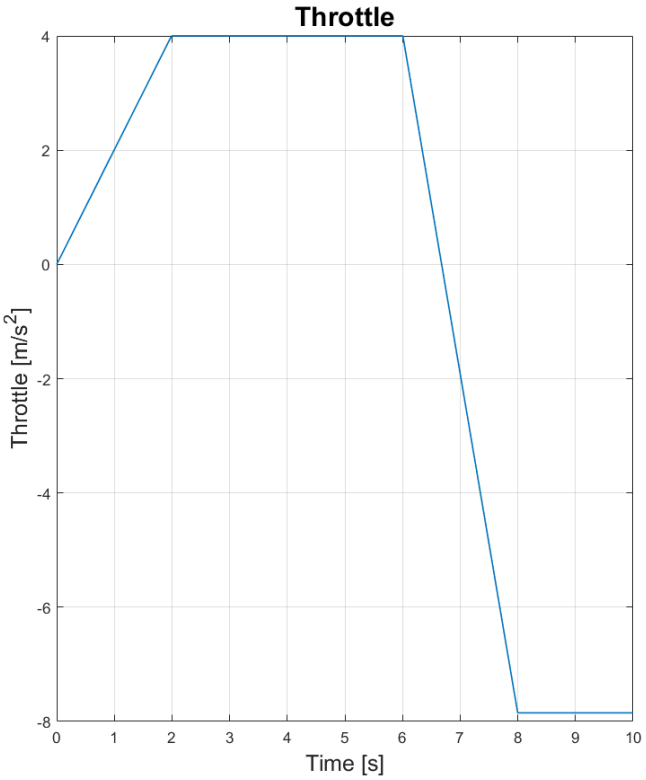


Figure 5: Throttle Input used in *Only throttle test* and *Combined test 1*. Maximum and minimum values are coherent with the range described in Section 4 in order to explore the whole set of throttle values.

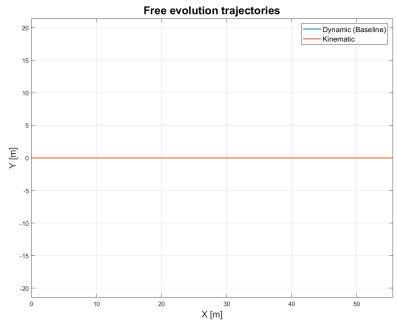
4.1.1 Simulink Test

The above-mentioned tests have been performed using the *Simulink Test* tool implemented by MathWorks. Simulink Test provides tools for authoring, managing, and executing systematic, simulation-based tests of models, generated code, and simulated or physical hardware. It includes simulation, baseline, and equivalence test templates that let you perform functional, unit, regression, and back-to-back testing using software-in-the-loop (SIL), processor-in-the-loop (PIL), and real-time hardware-in-the-loop (HIL) modes. With Simulink Test you can create non-intrusive test harnesses to isolate the component under test. You can define requirements-based assessments using a text-based language, and specify test input, expected outputs, and tolerances in a variety of formats. [7]

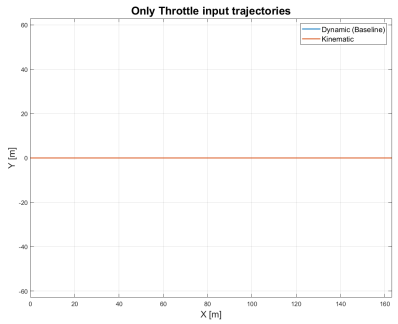
4.1.2 Results analysis

Exploiting Simulink Test, we have created two test harnesses : *Dynamic* and *Kinematic*. Those are referred to the models described at the beginning of this section. Then, we have executed an Equivalence Test which allowed us to make a comparison between two simulations. In particular, the Dynamic Model has been considered as the *baseline* which our Kinematic Model has been compared to in the test. The baseline represents the expected output being more accurate than the Kinematic model, which in turn is called *Compare to* model inside the Simulink Test automatic report generator[2]. Since these tests aim to give a qualitative analysis of the two models and compare them, we have not included equivalence criteria. The only parameter we have set up is the relative tolerance assigning to it a value of 1% in order to ignore the negligible offsets between the dynamic model and the kinematic model.
Figure 6 shows the results of the simulations we have carried out. As shown in Figures 6a and 6b respectively, as expected the two models behave in the same way when no inputs are present or when only the longitudinal input (throttle) is changed.
Another relevant result is shown in Figures 6e and 6f: as long as the amplitude of the sinusoidal input is fairly small, the two models behave in a similar fashion but the greater the amplitude of the sinusoidal input, the greater the deviation of the kinematic model from the dynamic one.
The other images shown below, underline as well the offset between the two vehicle models, which is due to the fact that when using the dynamic model we take into account lateral slips which are not considered in the kinematic model.

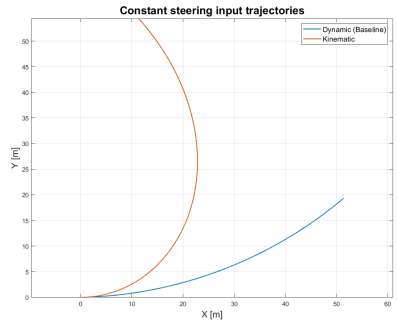
[2]The "Model Comparison - Test Report" file generated by Simulink Test is included in the /Documentation/Test Reports/ file path



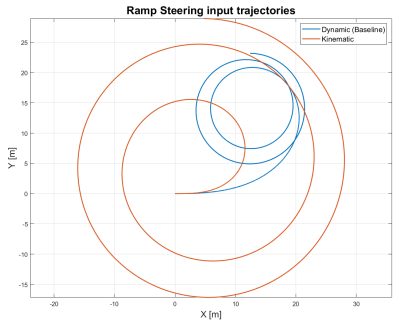
(a) Free evolution test - trajectories



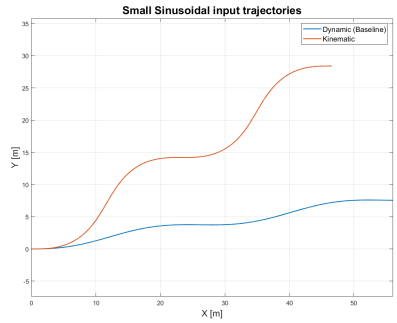
(b) Only throttle test - trajectories



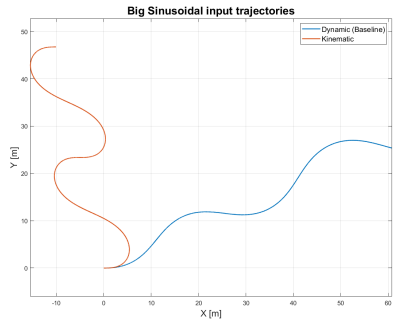
(c) Constant steering test - trajectories



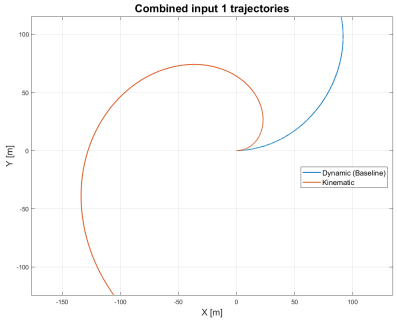
(d) Ramp steering test - trajectories



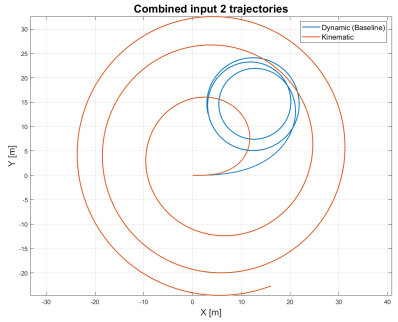
(e) Small sinusoidal test - trajectories



(f) Big sinusoidal test - trajectories



(g) Combined 1 test - trajectories



(h) Combined 2 test - trajectories

Figure 6: Output trajectories of the test set

After the analysis of the results, we decided to provide to the MPC the linearized bicycle model for the vehicle plant, because the controller is usually updated with the current state in some fractions of a second, and for this reason the differences between the model and the *"Real World"* system, that in our case is represented by the dynamic model, shouldn't affect too much the behaviour of the controller itself. As a matter of fact, by looking at the images above it is clear that the difference between the two models output is negligible at the first time steps of the simulation.
Moreover, as already discussed in this section, the performances of the two models are very similar if compared with small inputs, and this condition is fulfilled if we consider to use our controller in normal driving scenarios, such as, for instance, an highway.
Keeping in mind these assumptions, we go on with the project, considering to apply a change in the vehicle model for the MPC if we are not able to achieve good results with this one.

5 Signal Acquisition

All autonomous vehicles must have a sufficient knowledge of the surrounding environment and obtain accurate location information. For this purpose different sensors need to be adopted on a single vehicle, as a matter of fact different sensor have different limitations which can be overcome through sensor fusion. This approach is able to compensate the limitations of one sensor with the strengths of another.

5.1 Sensors used in obstacle avoidance

When dealing with autonomous driving, three are the core functions of interest:

- perception: visual perception (e.g., camera) and radar perception (e.g., LIDAR);
- planning: GPS, Inertial Navigation System (INS), and HD Maps;
- control: image identification, deep learning and artificial neural networks.

5.2 Perception: what a self-driving car sees

The three primary autonomous vehicle sensors for perception are camera, radar and lidar. Working together, they provide the car visuals of its surroundings and help it to detect the speed and distance of nearby objects, as well as their three-dimensional shape. [8]

5.2.1 Camera

Cameras are the main type of sensors for creating a visual representation of the surrounding world, as a matter of fact autonomous vehicles rely on cameras placed on every side to stitch together a 360 degrees view of their environment. Though providing very accurate images up to 80 meters, cameras have limitations:

- the distance from target objects needs to be calculated in order to know exactly where they are;
- mechanical issues: mounting multiple cameras as well as keeping them clean;
- issues related to low visibility conditions (e.g., rain, fog, nighttime);
- heavy graphic processing needed.

5.2.2 Radar

Radars can overcome cameras limitations in low visibility scenarios and improve object detection. The working principle of radars is based on radio pulses emitted by a source. Once these pulses hit an object they travel back to the sensor providing information about its speed and location. The maximum distance range for radar-based sensors is 250 meters for Long-Range Radars, 100 meters for Medium-Range Radars and 30 meters for Short-Range Radar.

5.2.3 Lidar

The name LiDAR stands for Light Detection and Ranging and its working principle is analogous to the radar's (Sub-section 5.2.2) but using a different part of the electromagnetic spectrum. Radars use radio waves or microwaves while lidars use light near the visible spectrum. This type of sensors make it possible for an autonomous car to create a 3D point cloud map and they provide shape and depth of the surroundings and its actors.

5.3 Planning: know where a self-driving car is going

"As the core of the planning layer, localization and navigation technologies including GPS, INS, and HD maps aim to assist self-driving cars in planning routes and navigating in real time." [9]

The above-mentioned technologies which have to be always active, allow the vehicle to go from point A to point B following the optimal route, and must re-compute in real-time the path if the optimal one has any unexpected diversions using a well tuned controller.

5.3.1 GPS

GPS stands for Global Positioning System. A GPS receiver is able to compute the current position of the vehicle together with the current time, acquiring and analysing signals received from at least four of the constellation of over 60 low-orbit satellites. The accuracy on the position information is 1 meter.

5.3.2 INS

INS stands for Inertial Navigation System and allows the accurate sensing of high-precision 3D position, velocity, and attitude information of the car via the inertial measurement unit (IMU). It can make up for the deficiency of GPS localization, which is not real-time enough.

5.3.3 HD Maps

HD Maps are crucial when it comes to Self-Driving Vehicles, these high-definition 3D maps are highly accurate and contain details not normally present on traditional maps. Such maps can be precise at a centimetre level. HD Maps consist of two map layers: static and dynamic HD Map. The former is usually collected in advance and contains lane models, road information, and road attributes while the latter, stored on a cloud platform, contains real-time traffic information. Moreover, dynamic HD Maps can be updated in real-time through information collected from vehicles and infrastructures on the cloud.

5.4 Our Assumptions

Our model is based on several assumptions regarding data acquisition from the sensors and how these data are translated into useful information. Below the list of assumptions we have made:

- we suppose no error in the information collected from the sensors;
- our car knows all the information related to the road model, such as lane width and number of lanes by acquiring them from cameras installed on the vehicle and HD Maps;

- the vehicle knows its position in the XY reference frame, through the use of GPS and INS;
- the route planning done by our controller exploits the latitude and longitude information to plan out a complete route based on the HD Maps, GPS, and INS. Moreover, while driving, the car also compares the HD maps with the perception information from sensors to get changing environment information for dynamically planning routes and making decisions;
- we suppose to have both a Long-Range Radar and Lidar mounted on the vehicle, hence to be able to detect obstacles in the range of 100 meters;

Though we include all the above information on the MATLAB code, we suppose to collect them using sensors, as previously described.

5.5 Reference trajectory

After implementing the vehicle model and keeping in mind the assumptions made above, we have generated and imposed a reference trajectory to the vehicle in order to have a path to follow during the development and the testing phases. To do so, we have taken the desired scenario from a real map using the open source website *OpenStreetMap*[10] (Figure 7). The website allows the user to export any route as a *.osm* file containing the global coordinates of the area together with some useful metadata such as street name, number of lanes, speed limits, etc.

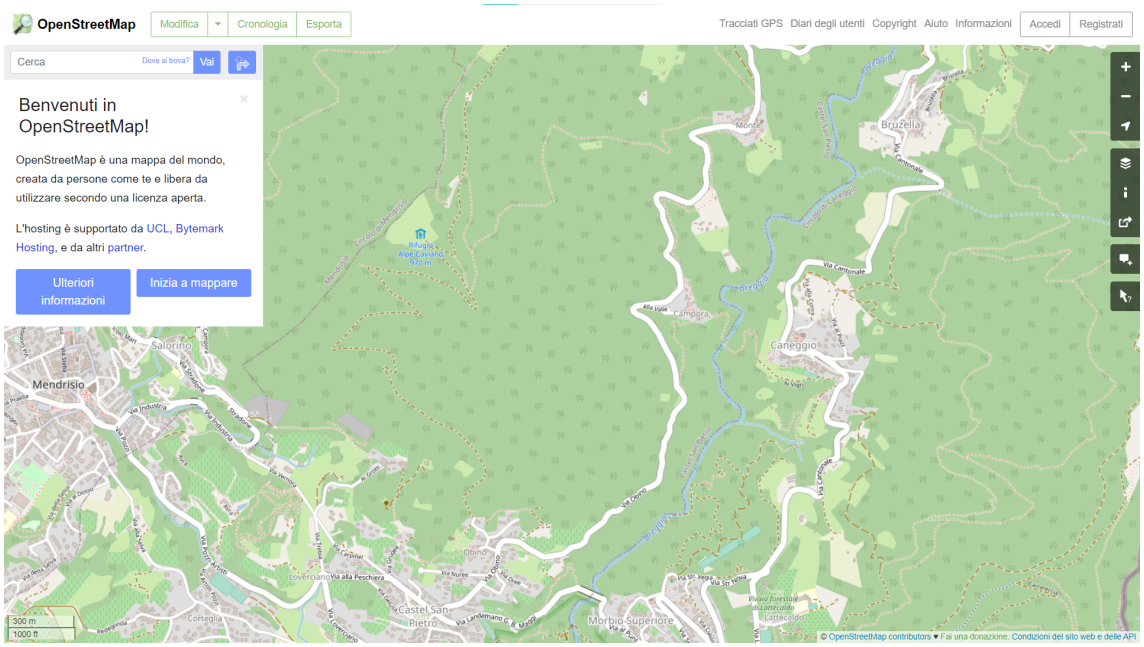


Figure 7: Selecting the area of interest in *OpenStreetMap*

Next, we have imported the *.osm* file in MATLAB exploiting the *Driving Scenario Designer* tool (Figure 8) provided as part of the Automated Driving Toolbox. The Driving Scenario Designer tool splits the whole selected route in as many pieces as the number of different streets names present. Then, the user can select only the pieces of road he/she is interested in and export its waypoints in a *.mat* file.

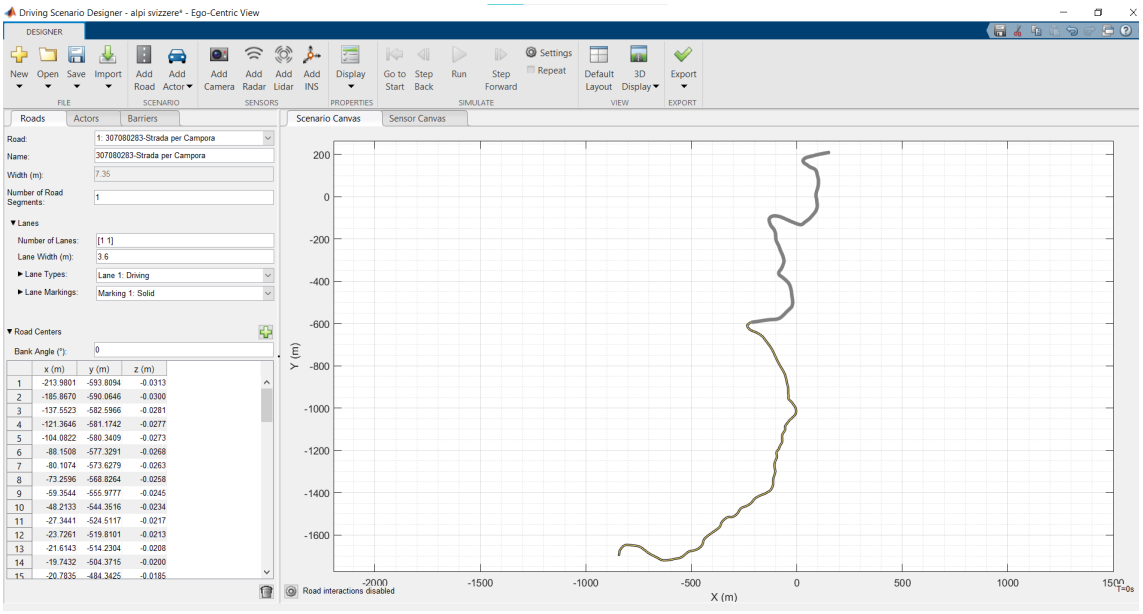


Figure 8: Importing maps on MATLAB using *Driver Scenario Designer*

Afterwards, the *.mat* file is used to generate the corresponding map in the XY reference frame. Since the thus generated map is composed by an insufficient number of waypoints for our applications, we have decided to create a function *ScenarioLoading* able to do an up-sample of the map because a higher density of points on it optimizes the path following algorithm. The up-sampling function interpolates linearly the original map with a fixed step depending on the reference speed and the sampling time selected for the controller. It is worth mentioning, as shown in Figure 9, that the up-sampled path (in orange) and the original waypoints (in blue) slightly deviate in some sections of the route due to the smoothing applied in order to obtain a more gentle trajectory.

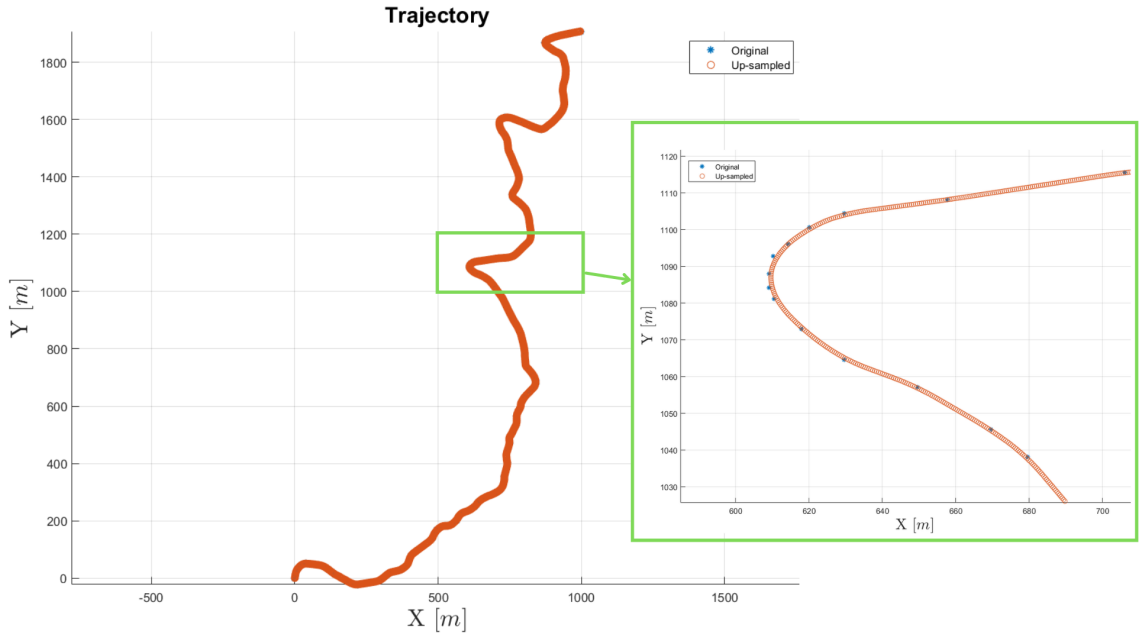


Figure 9: Comparison between original and up-sampled path

6 Constraints

In this section we are going to introduce the constraints we have used for the project. A plausible set of constraints is fundamental in order to perform a realistic simulation, which is required in order to have reasonable simulation outcomes.

6.1 Actuators constraints

The vehicle model presented in the Section 4 assumes that the inputs δ_f (front steering angle) and Throttle (driving and braking) can be controlled directly. In practice, however, low-level controllers are used to transform the aforementioned commands into physical control signals.

6.1.1 Steering angle constraints

Regarding the steering angle control, we suppose that our MPC directly affects the Electronic Power Steering (EPS) system by sending to it the target steering wheel position. Then, the EPS control unit calculates the optimal steering output based on the target steering wheel position received and sends the information to an electric motor to provide the necessary action on the wheels. For the project, we have assumed this link to be ideal and so our MPC directly control the angle of our front wheels. The values used are reported in the Table 3 and they have been chosen according to real data[11].

	min	**max**
Steering angle	-36 deg	36 deg
Steering rate	-60 deg/s	60 deg/s

Table 3: Steering constraints

6.1.2 Throttle constraints

As stated in Section 4 the vehicle model used in our simulation is, more or less, independent of the longitudinal dynamics of the vehicle. Moreover, the set of simulation we carry out are to be performed at constant speed, as close as possible to the target speed value. Physical limits on the actuators and comfort requirements impose bounds on the throttle and its rate of change according to the Table 4.

	min	**max**
Throttle	-7.85 m/s^2	4 m/s^2
Throttle rate	-20 m/s^3	8 m/s^3

Table 4: Throttle constraints

6.2 Output/State constraints

Tbd

7 Controller Setup

Matlab's Model Predictive Control Toolbox provides functions, an app, and Simulink blocks for designing and simulating controllers using linear and nonlinear model predictive control (MPC) [12]. Thanks to the toolbox we specified the plant model, horizons, constraints, and weights. By running closed-loop simulations, we evaluated controller performance and adjusted its behavior by varying weights and constraints at run time.

7.1 MPC Initialisation

Initialisation phase requires the definition of all the parameters needed for the execution of the controller. First of all we have created a MPC object passing as input the discretized state-space model of the plant, the prediction horizon and the control horizon.
Assuming that the car is already traveling at a speed V, the initial conditions for the other states is set taking the first point in the reference path, and for the two inputs it is set equal to zero. Exploiting the function *odometer* we computed the total distance covered by the car following the scenario in order to let the simulation terminates exactly when the road ends.
Subsequently we defined the constraints on inputs (Table 3 and 4) and the weights for the cost function, which correctly tuned, guarantee that the vehicle follows the reference without going off the road i.e. stays within the lane width limits and is capable to avoid an obstacle when is inside the sensors detection range.

7.2 MPC Algorithm

Once the controller has been initialised, all is ready to let the simulation begin.
At each time−step, different instructions are repeated in a *for* loop to simulate the controller in a closed−loop fashion. Figure 10 graphically shows the general concept behind a model predictive control; below it's described the Matlab algorithm that follows the same scheme.

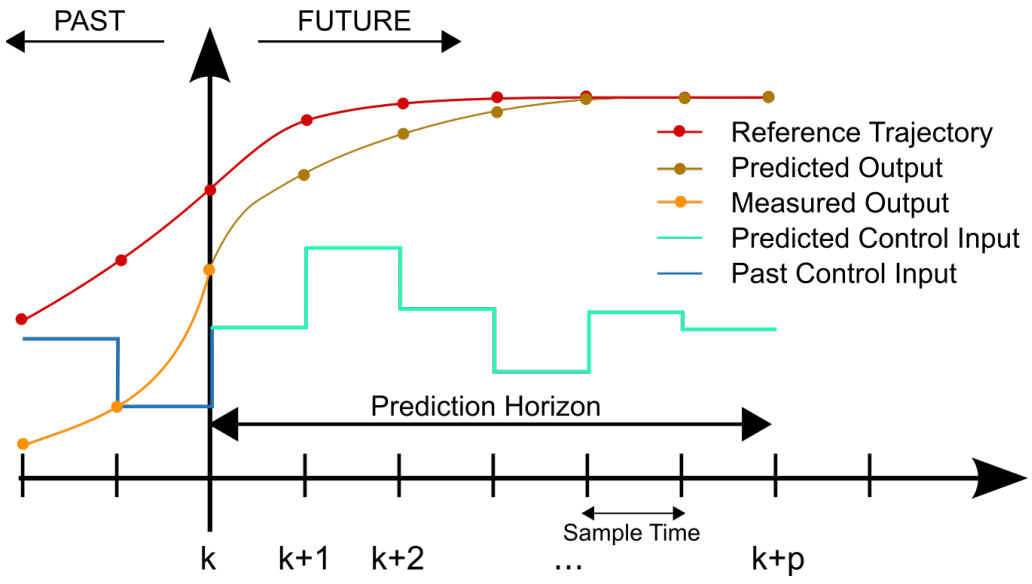


Figure 10: MPC algorithm scheme

First, the current state and the inputs are used in the function *obstacleVehicleModelDT*

to generate a discrete−time model, using the Zero−Order Hold (ZOH) method, and consequently to generate the nominal conditions for the discrete plant and the measurements. Then, the reference path is updated: starting from an index that is increased at each cycle, the trajectory data are read along the prediction horizon length. Once the new plant model and nominal conditions are calculated, they are used as inputs for the *mpcmoveAdaptive* Matlab command that computes the Adaptive MPC control action at current time. This same control action is subsequently used as input to the *VehicleModelCT_DYN_ode* function that outputs the differential equations relative to the vehicle dynamic model; these equations are integrated from 0 to the time−step using the *ode45* function.
In the end, the loop is concluded assigning to the next state[3] the result of the integration. The pattern described above is repeated n times, where n represent the ratio between the set simulation duration and resolution.

7.3 Simulink Model

In order to run closed-loop simulations a Simulink model has been designed because of the availability of many useful tools to test and validate the model. For the sake of clarity in reading and debugging the whole system, some of the Simulink blocks have been grouped in subsystems and a mask applied to some of them.

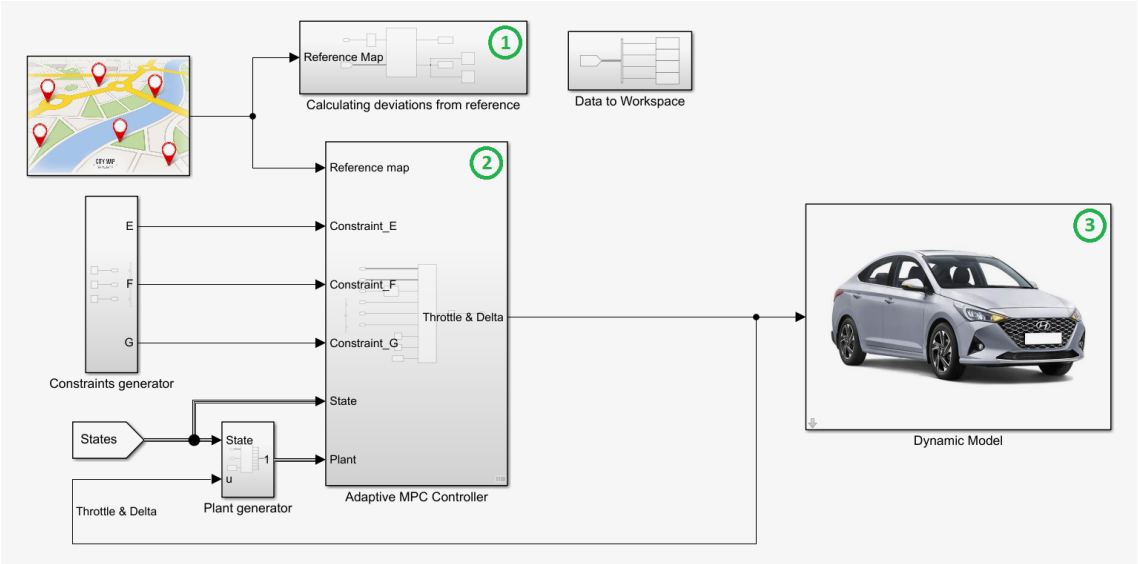


Figure 11: Simulink main

In Figure 11 the main system is depicted: the process simulated by these blocks is coherent with the MPC algorithm described in the previous section and the the numbers in green enumerate the three major subsystems and will be analyzed in the following.

[3]The *next* state in the current cycle will correspond to the *current* state in the successive cycle as the index is increased

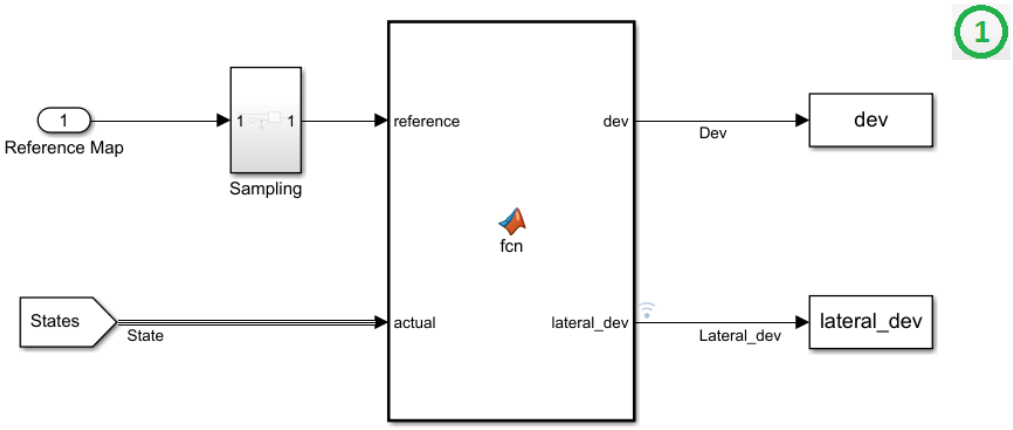


Figure 12: Simulink subsystem to calculate deviations

Namely, the content of the subsystem numbered 1 is displayed in Figure 12: the main purpose of these blocks is calculating a deviation from the reference in order to test controller and will be analyzed in details in the next Chapter **??**; in order to pass the proper number of reference way-points, a sampling subsystem is used as a input to the Matlab function block. A similar sampling subsystem is used also as input to the reference of the simulink block *Adaptive MPC*: in Figure 13 below indeed, are showed the blocks composing the subsystem *Adaptive MPC Controller* and in the blue rectangle is enlighted how the content of the reference map is sampled[4].

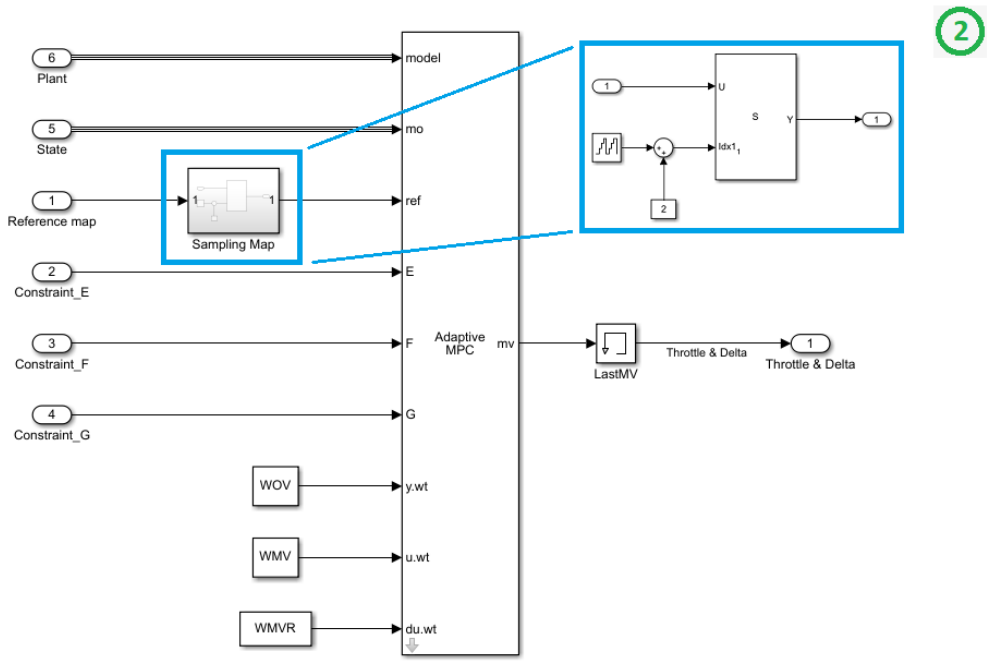


Figure 13: Simulink subsystem: Adaptive MPC controller

[4]The sampling subsystems displayed in Fig.12 and Fig.13 are both structured in the same way (selector and counter basically) but are different in the sampling rate

The subsystem *Dynamic Model* displayed in Fig.11 has as a mask the picture of an Hyundai Verna as this is the vehicle whose parameters are used in the simulation. Also for these blocks − Fig.14, the main purpose is calling an already designed Matlab function that in this case is the *VehicleModelCT_DYN* one, previously described in Chap.4.

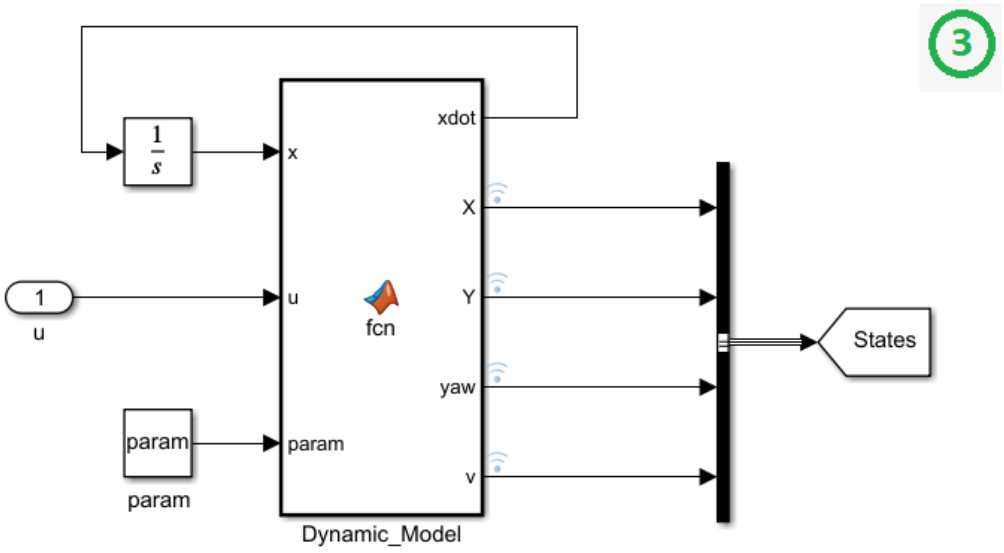


Figure 14: Simulink subsystem: Dynamic Model

7.4 Partitioning of the MPC functionalities

The developed MPC controller is supposed to carry out three main tasks:

- **Path following**: the car must be able to follow a given reference path without exceeding a predetermined threshold distance;
- **Static obstacle avoidance**: the car must sense and dodge any kind of stationary obstacle present on the road i.e. road construction sites, holes, trees, car accident, etc.;
- **Dynamic obstacle avoidance**: the car has the ability to perform overtaking maneuvers in the event that other moving vehicles, cyclists, pedestrians or animals are present in the same lane. The avoidance maneuver must be accomplished whether they are traveling in the same direction as the vehicle or in the opposite one.

Figure 15 shows how the ego-car (red) performs the overtaking maneuver: firstly it analyze the surrounding thanks to the on-board sensors and if it has enough space to achieve the lane change accordingly to the position and speed of the two moving obstacle (white cars) it plans the trajectory to follow to achieve the avoidance (blue path). Once the ego-car has planned a feasible path it starts to steer and accelerate arriving to the middle lane overtaking the car in the rightmost lane.

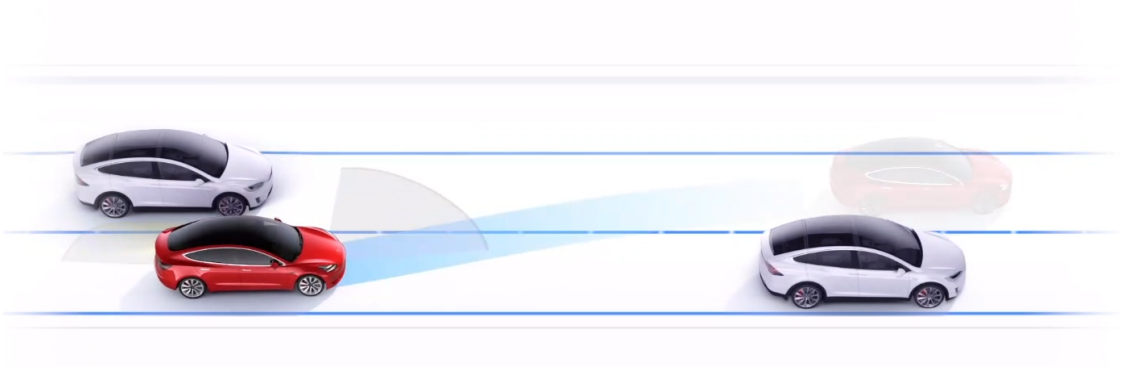


Figure 15: Dynamic obstacle avoidance maneuver on a three lanes road

In order to perform the path following task the MPC collects at each timestep the successive waypoints of the map for the duration of the prediction horizon and elaborates an optimal trajectory and control action to minimize the a cost function of the controller. Things get harder when obstacles are present on the road, either they are static or moving: we assumed to know "a priori" the position of the static obstacle or the trajectory that the moving obstacle is following, in such way thanks to the MATLAB function *"detectionFun"* we augmented the algorithm used for the path following with the capability to detect obstacles within the sensing range of the vehicle, thus allowing to promptly start and execute the avoidance maneuver always being outside the so called *"safe zone"* of the obstacle.
Figure 16 plots an example scenario for static obstacle avoidance, where:

- the **ego car** is represented by the green dot with the black boundary;
- the **horizontal lanes** are represented by the dashed blue lines;
- the **static obstacle** is represented by the red x with the black boundary;
- the **safe zone** is highlighted by the dashed red boundary.

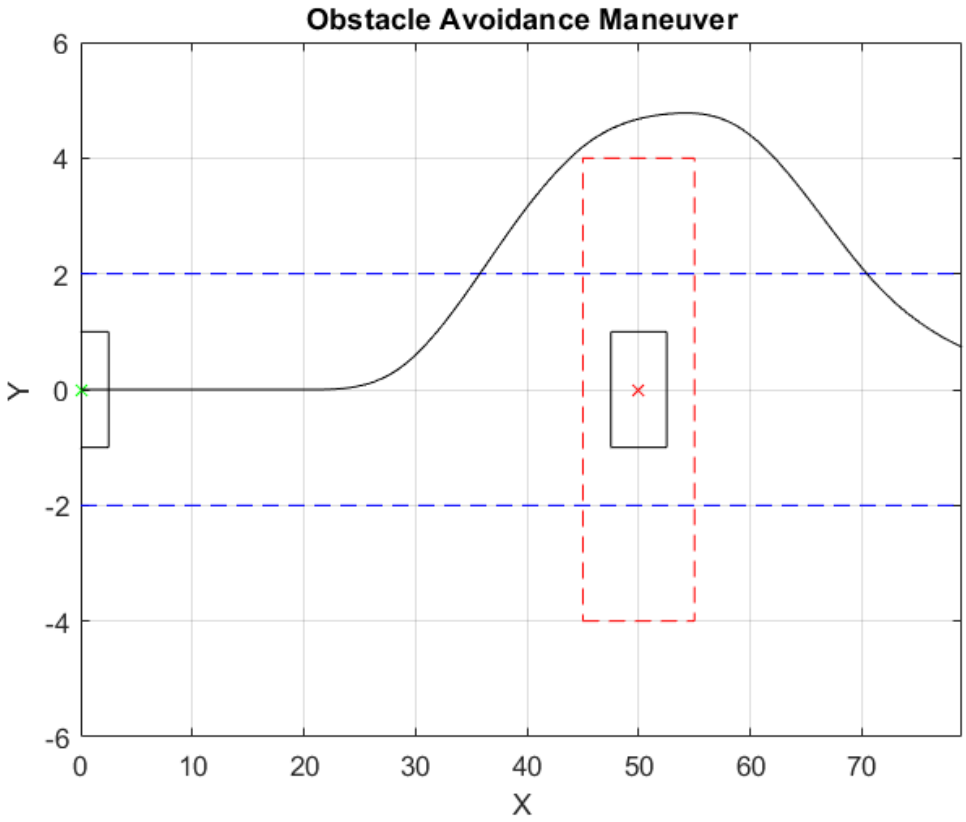


Figure 16: Example of static obstacle avoidance maneuver

References

[1] Roberta Frisoni, Andrea Dall'Oglio, Craig Nelson, James Long, Christoph Vollath, Davide Ranghetti, Sarah McMinimy, and Steer Davies Gleave. Research for tran committee–self-piloted cars: The future of road transport? *European Parliament, Directorate-General for internal policies, policy department B: Structural and Cohesion Policies, Transport and Tourism*, 2016.

[2] MathWorks Inc. Understanding model predictive control. `https://it.mathworks.com/videos/series/understanding-model-predictive-control.html`.

[3] Kim R. Fowler. Chapter 1 - introduction to good development. In Kim R. Fowler and Craig L. Silver, editors, *Developing and Managing Embedded Systems and Products*, pages 1–38. Newnes, Oxford, 2015.

[4] MathWorks Inc. Obstacle avoidance using adaptive model predictive control. `https://it.mathworks.com/help/mpc/ug/obstacle-avoidance-using-adaptive-model-predictive-control.html`.

[5] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1094–1099, 2015.

[6] Gary J. Heydinger, Ronald A. Bixel, W. Riley Garrott, Michael Pyne, J. Gavin Howe, and Dennis A. Guenther. Measured vehicle inertial parameters-nhtsa's data through november 1998. *SAE Transactions*, 108:2462–2485, 1999.

[7] MathWorks Inc. Simulink test. `https://www.mathworks.com/help/sltest/index.html?s_tid=srchtitlel`.

[8] Katie Burke. "How Does a Self-Driving Car See?". `https://blogs.nvidia.com/blog/2019/04/15/how-does-a-self-driving-car-see/`, 2019.

[9] Qian Luo, Yurui Cao, Jiajia Liu, and Abderrahim Benslimane. "Localization and navigation in autonomous driving: Threats and countermeasures". *IEEE Wireless Communications*, 26(4):38–45, 2019.

[10] Openstreetmap. `https://www.openstreetmap.org`.

[11] Garrick J Forkenbrock and Devin Elsasser. An assessment of human driver steering capability. *National Highway Traffic Safety Administration DOT HS*, 809:875, 2005.

[12] MathWorks Inc. Model predictive control toolbox. `https://www.mathworks.com/products/model-predictive-control.html`.