

POLITECNICO DI TORINO

Master Degree in Mechanical Engineering



Master Thesis

*Validation of a coupled Lane Keeping and Adaptive
Cruise Control inside a high fidelity driving simulator
scenario*

Supervisors:

Prof.ssa Daniela Misul
Ing. Stefano Ballesio
Ing. Salvatore Calanna

Candidate:

Nicolò Simonini

Luglio 2021

Abstract

Since the beginning, humans have been trying to evolve and improve their lifestyle; transport is for sure one of the most relevant factors influencing standards of living.

Since the first cars were developed, automotive industries have been focusing on two main aspects: improving efficiency, thus reducing costs and increasing power generation (considering environmental issues as well), and trying to reduce as much as possible accidents, fatalities and injuries. These two objectives led to a new need, which nowadays belongs to the process of development of new cars: simulations.

Being able to virtually reproduce what would happen in the real world allows engineers and technicians to reduce testing time and to avoid possible accidents during the design process (e.g. under-estimation of loads), which, as a consequence, means reducing costs.

Simulators are used in many different ways. They can be used with (Driver In the Loop) and without a human driver, the so called Software/Hardware In the Loop testing. With SIL (Software in the Loop) the user tests a certain Electronic Control Unit (ECU) in a totally virtual environment which reproduces the future working system; this is performed by means of computers only, thus requiring low costs. On the other hand, HIL (Hardware In the Loop) involves some physical parts like sensors, actuators and so on. This leads to a higher cost, but sometimes (in particular in the automotive field) these tests are mandatory. Some examples of ECUs that are tested in this way are ADAS (Advanced Driver Assistance Systems), which are gaining more and more importance in the automotive field reducing accidents and fatalities on the road, as well as Engine Control Unit, Airbag Control Unit and many others.

The whole world of simulators relies on software packages which must be able to generate scenarios for virtual tests. In order to design a simulation scenario, different aspects are to be considered:

- Vehicle editing: the car to be tested should be designed in the most precise way possible, starting from the kind of fuel-based vehicle, up to the definition of vehicle dynamics settings and so on.
- Simulation environment editing: both from a merely graphical point of view (urban, extra-urban, highway and so on) and from a traffic point of view (the addition of Non Played Characters, like other cars or pedestrians, could be necessary).
- Software integration: sometimes the user could need to perform a simulation using Matlab/Simulink or other programs. Software integration is therefore a very relevant aspect.
- How can the user log the simulation data? As simulation outputs are relevant and very important, the way of obtaining them, both in real-time and afterwards, is crucial both in terms of which data are available to be logged and how they can be logged (e.g. does it require additional machines or software?).
- Is there the possibility of implementing ADAS logics? Are there sensors and cameras which can retrieve information from the simulation?

As usual, the higher the cost of the software, the better it will be, thus leading to a higher number of editable features and a much easier-to-use interface. As there are many possibilities available, the user can choose the best one depending on his/her needs.

In this Thesis Project, two driving simulator software packages have been analysed: LGSVL, offered by LG, and WorldSim, belonging to the VI-Grade world (one of the leading companies dealing with driving simulation software).

LGSVL is open source, thus not requiring any license. However, as a drawback, it is not very user-friendly. The editing of vehicles and simulation scenarios is quite poor and not straightforward, requiring some programming skills and, even in case of simple changes, the access to a Unity license, thus introducing a cost. On the other hand, WorldSim requires a commercial license; this leads to a much easier-to-use simulation software on all aspects that have been briefly introduced before.

As the reader may realize, WorldSim is by far the best between the two. Addfor S.p.A, which has hosted my curricular stage, has a partnership with Danisi Engineering that, as an advanced automotive company, offered me the possibility of using and analysing WorldSim on their static simulator at Nichelino (TO).

After some weeks of debugging and analysing the software, I had the chance of validating a control model of a vehicle, coupling the Lane Keeping Assist (LKA) (lateral control) to the Adaptive Cruise Control (ACC) (longitudinal control). Thanks to the Radar and Lane detector sensors which have been added to the user vehicle, by streaming information from/to the simulation via UDP, the Simulink model is able to act on the steering wheel angle and on the throttle and brake pedals.

After several months inside the world of driving simulators and after having applied an ADAS on one of them, I could realize the growing importance of this software. It is now more than clear that automotive industries should think about investing in this field, because it can lead to a strong reduction in costs and a huge improvement in terms of passenger safety, thus leading to more and more possible customers.

Acknowledgement

Vorrei ringraziare in primis la mia famiglia per il supporto che mi ha sempre fornito. Ringrazio tutti i miei amici, sia vecchi che nuovi, che a loro modo hanno contribuito a portare a termine questo percorso. Un particolare ringraziamento va infine a Nicolò, senza il quale non sarei mai riuscito a superare tutti gli ostacoli e le sfide.

Contents

1	Introduction	4
1.1	Simulation in the world of engineering	4
1.1.1	The need of a simulation process	4
1.1.2	Driving simulators	5
2	Soft part of driving simulators	8
2.1	Terminologies and definitions	8
2.2	Features of a Driving Simulator Software	10
2.2.1	Degree of accuracy in editing of simulation scenario	10
2.2.2	Degree of accuracy in editing of Ego vehicle	12
2.2.3	Data logging and Real-Time analysis	13
2.2.4	Integrability of the software	13
2.2.5	Possibility of ADASs' tests	14
2.3	Vehicle models used for simulations	17
2.3.1	Kinematic models	17
2.3.2	Dynamic model	17
3	LGSVL vs VI-WorldSim	19
3.1	Generalities	19
3.2	LGSVL	19
3.2.1	Additional requisites	20
3.2.2	Simulation setup	21
3.2.3	Data Logging	27
3.2.4	Integration with other platforms: path planning and route editing with Apollo 5.0	28
3.3	VI-WorldSim	31
3.3.1	Soft architecture of the Static Simulator at Nichelino Plant	31
3.3.2	VI-WorldSim Studio	32
3.3.3	Integration of the software	37
3.3.4	Real-Time analysis and Data Logging from the sensors	37
3.4	Final comparisons	38
4	ADAS	40
4.1	The need of a Driving Simulator	40
4.2	Why ADAS	40
4.2.1	Crashes reduction	41
4.3	Classification	42
4.4	Euro NCAP	44
4.4.1	NCAP tests before ADAS introduction	44
4.4.2	NCAP tests after ADAS introduction	44
4.5	State of Art: Stanley method	45

5	Control model	48
5.1	General functioning	48
5.1.1	Lane Keeping Assist	48
5.1.2	Adaptive Cruise Control	49
5.2	Control model inside WorldSim	49
5.2.1	LKA	49
5.2.2	ACC	51
5.3	Lateral Control	52
5.3.1	Reference generation	53
5.3.2	Steer angle generator	56
5.3.3	Steer torque computation	58
5.4	Longitudinal control	58
5.4.1	Reference speed generator: IDM	58
5.4.2	Throttle and Brake computation	60
5.5	Vehicle model	61
5.5.1	VI Car Real Time	61
5.5.2	Vehicle model for simulation	63
6	Validation of the control model: experimental testing	64
6.1	Simulation setting on a Real-Time Concurrent machine	64
6.1.1	Creation of the simulation scenario with WorldSim Studio	64
6.1.2	SimWorkbench and DriveSim	66
6.2	The static simulator	68
6.2.1	Immersion of the driver	69
6.2.2	General architecture	70
6.2.3	Communication SimWorkbench - MATLAB/Simulink	72
6.3	Modified architecture for the experimental process	76
6.3.1	UDP Communication	77
6.3.2	Steering Wheel	79
7	Results and conclusions	84
7.1	Results from the control model and the simulation	84
7.1.1	Adaptive Cruise Control	84
7.1.2	Lane Keeping Assist	85
7.2	Conclusions	86
7.3	Next possible steps	87
A	Simulink Model	88

List of Figures

1.1	Advantages of simulations	4
1.2	Driving simulators	5
1.3	Hardware In the Loop	6
1.4	Hardware In the Loop	6
1.5	ADASs' typical functions on a vehicle	7
2.1	Simulation scenario examples	8
2.2	Sensors examples	9
2.3	Urban scenario generated with VI WorldSim.	10
2.4	Extra-urban scenario generated with VI WorldSim.	10
2.5	Highway scenario generated with VI WorldSim.	11
2.6	Simulink model of a vehicle; here the most external blocks.	12
2.7	Data logging from a simulink model.	13
2.8	GPS sensor	14
2.9	Radar sensor	14
2.10	LiDAR sensor	15
2.11	Depth camera.	15
2.12	Semantic segmentation camera	16
2.13	Ultrasonic sensor effect.	16
2.14	Kinematic model of a vehicle.	17
2.15	Dynamic model of a vehicle.	18
3.1	Example scenario from LGSVL.	19
3.2	Unity Interface.	20
3.3	Graphical User Interface of Apollo.	21
3.4	Web User Interface	21
3.5	Adding new map to the User Interface	22
3.6	LGSVL Simulator contents: Borregas Avenue	22
3.7	LGSVL map examples	23
3.8	Graphic User Interface of LGSVL Simulator	24
3.9	Editing possibilities	25
3.10	Ego vehicle configuration on the WebUI	26
3.11	Monitor for data streaming check.	29
3.12	Apollo Dreamview	30
3.13	Rack with all machines assessing for Static Simulator correct functioning	31
3.14	WorldSim MCity map	32
3.15	WorldSim Neighborhood map	33
3.16	WorldSim extraurban map	33
3.17	WorldSim highway map	34
3.18	Triggers examaple	35
3.19	GUI of VI-CRT: aerodynamic forces setting	35
3.20	WorldSim sensor configuration	36
3.21	Sensor addition to the ego vehicle	36
3.22	Resuming table.	39

4.1	Forward collision warning statistics	41
4.2	Forward collision warning with automatic braking statistics	41
4.3	Rear automatic braking statistics	41
4.4	Percent of new passenger vehicles available with ADASs	42
4.5	Classification of driving automation	43
4.6	“Stanley”	45
4.7	Stanley scheme	46
4.8	Large heading error case	47
4.9	Large positive track error case	47
5.1	Lane Keeping Assist	48
5.2	Adaptive Cruise Control	49
5.3	Right lane curvature computation block	50
5.4	Schematic representation of Lateral Control functioning	52
5.5	Reference information.	53
5.6	Reference information.	54
5.7	Ego vehicle on the right of the desired trajectory.	54
5.8	Ego vehicle on the left of the desired trajectory.	55
5.9	Steering wheel angle generator scheme	56
5.10	Stanley method	57
5.11	Control torque computation	58
5.12	Speed desired	59
5.13	Longitudinal control	60
5.14	Feedback contribution	61
5.15	Rigid parts of CRT vehicle model	62
6.1	Ego vehicle configuration	64
6.2	Lead vehicle configuration	65
6.3	SimWB Test	66
6.4	VI-DriveSim GUI	67
6.5	VI-Controller GUI	68
6.6	Static simulator.	68
6.7	Example of scenario on the curved screen.	69
6.8	Flat screen assessing for the car mirror.	69
6.9	Control room.	71
6.10	Static simulator architecture.	71
6.11	RTDB library inside Simulink.	72
6.12	Build tab.	72
6.13	Simulator connection by means of SimWB Toolkit.	73
6.14	Code generation by means of SimWB Toolkit.	74
6.15	Net I/O on SimWB.	74
6.16	Add new messages on I/O Mappings	75
6.17	UDP communication Library in Simulink.	75
6.18	Modified architecture of the simulator for experimental tests.	76
6.19	Data streamed towards the concurrent	78
6.20	Gear shifting LUT	78
6.21	Steering wheel	79
6.22	Joystick Input block	80
6.23	Demux of Axes from the Controller	81
6.24	Demux of Buttons from the Controller	81
6.25	PID scheme	82
6.26	PID search for optimum tuning	83
6.27	New PID output	83
7.1	Adaptive Cruise Control effect	84
7.2	Lane Keeping Assist effect	85

7.3	HIL process	86
7.4	ADAS	86
A.1	Curvature computation block.	88
A.2	Cross track error computation.	88
A.3	Reference generator (LKA).	89
A.4	Reference steering wheel angle generator.	89
A.5	Stanley block.	90
A.6	Control torque computation.	90
A.7	Ego raw speed desired computation.	90
A.8	Ego speed desired computation.	91
A.9	Feedforward part for commands computation (ACC).	91
A.10	Feedback loop for commands computation (ACC).	91

Chapter 1

Introduction

In this chapter I will briefly introduce the main subjects of my Thesis, in order to allow you to have a better understanding of the reasons that led me to investigate this world.

1.1 Simulation in the world of engineering

1.1.1 The need of a simulation process

In the latest decades, the threshold of computers scapabilities has been moved forward and forward. This has affected not only the informatic world, but also the mechanical one. In fact, nowadays we have no longer a clear separation between those fields, which are actually becoming more and more interconnected.

Let us think at the whole development of any mechanical system: it requires a design procedure, in which a lot of parameters have to be taken into account like the materials to be used, static and dynamic loads, the operative environment and so on.. All these requirements, which can be defined as *mechanical*, need to comply with two other important features: the time we have to obtain the final product and the available budget (€) for the project.

The development and the birth of new technologies, both from hardware and software point of view, have been (and will be) very important because of the **reduction** they led to **in terms of money and time**.

As a general rule, changes at the first steps of the design of a product cost (both in terms of time and money) a lot less than the ones performed at the final stages.

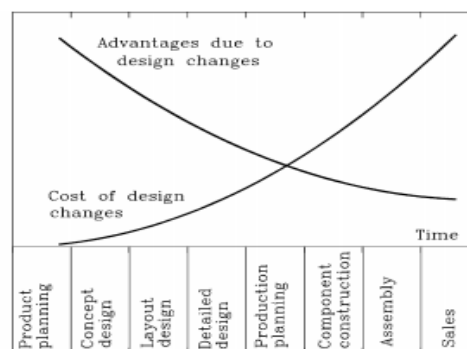


Figure 1.1: Advantages of simulations

In the figure 1.1 the difference, from a qualitative point of view, in changes at the different steps of the production is perceived. If we wanted to consider the effect of the simulations in the design of a product, we would have to cut the part of graph to the right of the “Production planning” since a proper simulation procedure leads to avoid (almost completely) the possibility of changes at those stages.

1.1.2 Driving simulators

Previously we have described the main reason why simulations are so important in production and manufacturing fields.

Now we want to focus our attention more in specific on *driving simulators*. As the name can suggest, they are used to reproduce and simulate the behavior of a vehicle. Several types of simulators exist, depending on the **fidelity**, which represents the potential of the simulator (basically it defines how much the simulation is realistic): starting from a simple desktop (as visual), steering wheel and pedal input devices (as depicted in figure a) up to the most complex ones like the dynamic simulator with multiple degrees of freedom represented in figure b).



(a) Desktop simulator with steering wheel and input pedals



(b) Dynamic simulator at Nichelino (TO) plant of Danisi Engineering

Figure 1.2: Driving simulators

The complexity of the system, the huge amount of variables (materials, environment, traffic and so on) that can affect the vehicle and the many possible behaviors that can be analyzed (dynamic, aerodynamic, and so on) lead to the fact that driving simulators are becoming more and more important in the automotive industry.

Help in the design and manufacturing though is not the only field of interest for driving simulators. They are also used for drivers' training, both for competitive races (motorsport field, figure a) and for licenses (e.g. truck driving simulators in figure b); in the advantages we have less cost, higher safety, less stress during lessons and shorter learning times. Then we can not forget about entertainment (videogames) and the new virtual competitions.



(a) Formula 1 driving simulator



(b) Truck simulator

DIL, SIL and HIL The main application, in terms of automotive manufacturing, of driving simulators is the testing and validation of Electronic Control Units: ADAS, Engine Control Unit, Airbag Control Unit and many others are then tested in this way.

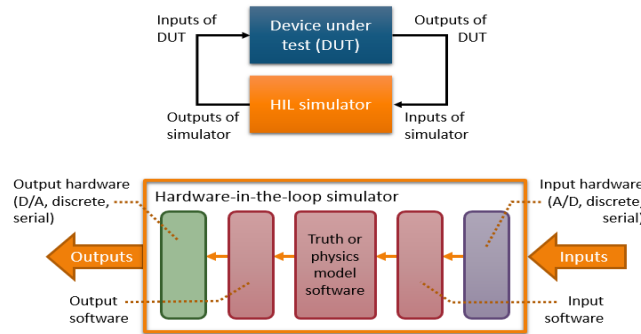


Figure 1.3: Hardware In the Loop

Basically these units are tested in a totally (Software In the Loop) or partially (Hardware In the Loop) virtual scenario which reproduces the future working environment. In this way mechanical components of the vehicle can be stressed and tested at very early stages of the product design:

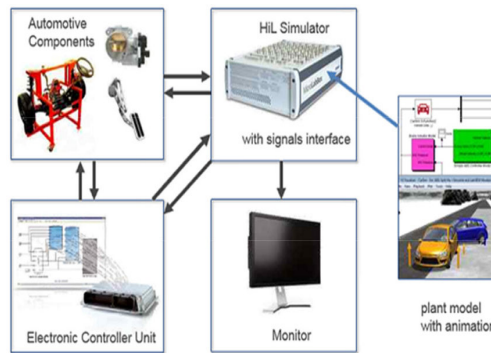


Figure 1.4: Hardware In the Loop

ADAS - Advanced Driving Assistance System As already introduced, one of the ECUs that are tested on Driving Simulators are ADASs. As the name suggests, their aim is to help the human driver into driving and parking functions (it is important to underline that up to now they are not meant for substituting him/her).

Most of the road accidents are due to human errors and ADAS's aim is to minimize these errors, thus leading to less fatalities during driving.

We will explain more in detail how an ADAS actually works, but for now let us describe its functioning by just saying it is based on the coupling between *cameras and sensors* and *algorithms*. The firsts are able to retrieve information of the environment (outside but also inside of the vehicle), while the others use those informations as input to generate an output to control a particular function of the vehicle.

The list of actions that an ADAS can perform is quite long; we can find parking cameras and sensors, the Cruise Control function, the Lane Departure Warning System (LDWS), the Emergency Braking (EB) and so on.. In the following picture you can see some of the possible functions that ADASs can provide.

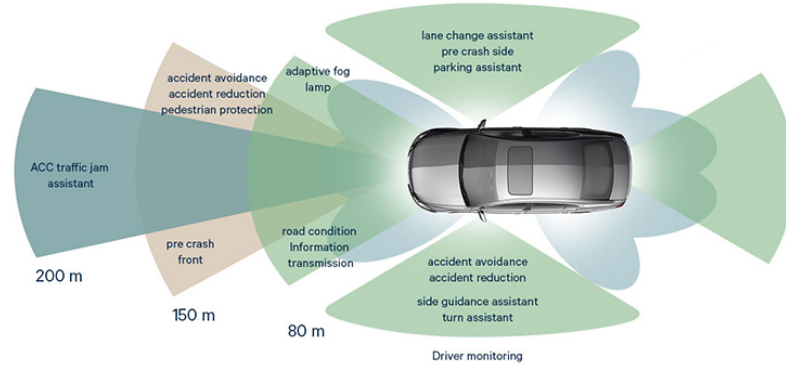


Figure 1.5: ADASs' typical functions on a vehicle

As you can imagine, some of them are not life-depending actually (e.g. parking cameras), while others (like the EB or the LDWS) could be fatal if not properly designed. For this reason the study and analysis of these system can not be performed on the road with actual vehicles and human drivers, thus leading to the necessity of Driving Simulators and softwares.

Chapter 2

Soft part of driving simulators

In this chapter, the *soft part* of a driving simulator is described. In particular, the requirements for the simulation scenario are analyzed, from the editing of the environment, to the extraction of simulation data and the dynamic models that can be used to represent vehicles.

2.1 Terminologies and definitions

First of all we need to give some definitions of terms that will be used throughout this documentation.

- **Scenario:** it represents the map and the environment of the simulation. We can have many possible scenarios for performing a simulation and each of them offers the chance to analyze different situations: you may find a city, a part of a city, an highway, a non-urban scenario, a parking lot and many others. In the following pictures we have reported a couple of examples:

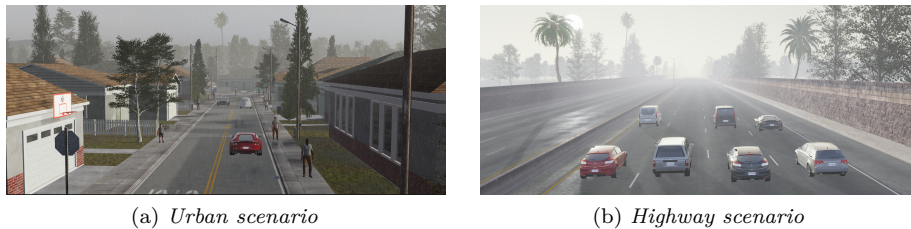
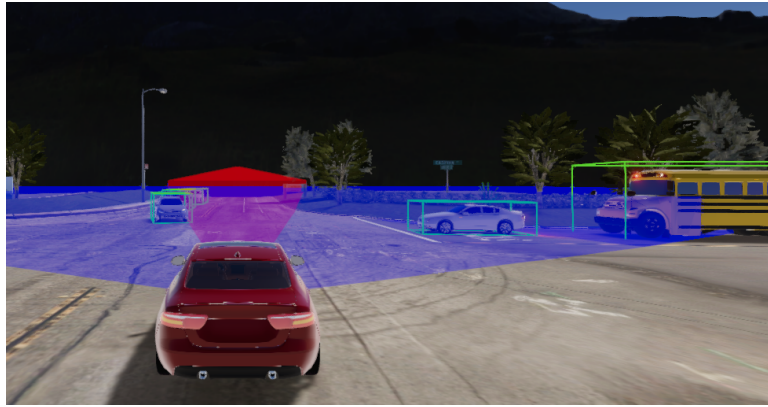


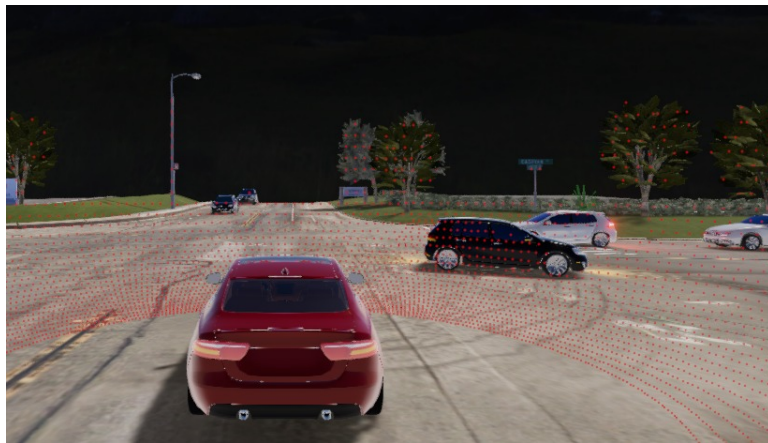
Figure 2.1: Simulation scenario examples

- **Ego vehicle:** this is the vehicle that is controlled by the user, regardless the way of doing it. Several control ways are available, from the use of physical controllers (steering wheels, joysticks and others) up to the automatic control by means of scripts or simulink models.
- **NPC vehicles:** Non Player Character vehicles are the ones which constitute the “traffic” driving around the scenario. Their behavior is determined anyway by the user, who can even choose their trajectory for example.
- **Agent:** in general it defines an active element of the simulation, thus vehicles, pedestrians and animals.
- **Obstacle:** anything that is still and, because of that, becomes an impediment for some agents in the simulation.

- **Sensors and Cameras:** they are the representation of the physical sensors and cameras present on the vehicle. Let us think for example at the *Radar* (figure a), the *Lidar* (figure b), the GPS etc which are used in almost every ADAS, but in general on most cars.



(a) *Radar camera*



(b) *Lidar camera*

Figure 2.2: Sensors examples

2.2 Features of a Driving Simulator Software

When you deal with a DSS and you want to make sure it fits your needs, some features and functionalities are usually checked.

2.2.1 Degree of accuracy in editing of simulation scenario

For sure one of the most important feature inside a simulation is the *environment* in which it is performed. In the following you will find the three mainly used environments and one possible application for each of them:

- **Urban:**

This scenario is usually populated with cars and pedestrians. This allows everything that involves these agents, like emergency braking in case some pedestrian suddenly crosses the street or the interaction at junctions or roundabouts (with or without the presence of blind spots during the drive) and so on.



Figure 2.3: Urban scenario generated with VI WorldSim.

- **Extra-urban**

This is mainly used for ADASs applications. One of the most analyzed case is the overtaking of a vehicle which suddenly brakes or which is seen from the driver at the last minute; the aim of the ADAS in this case is to help the human driver in avoiding the collision either by simply braking or braking and steering. An example is depicted below:

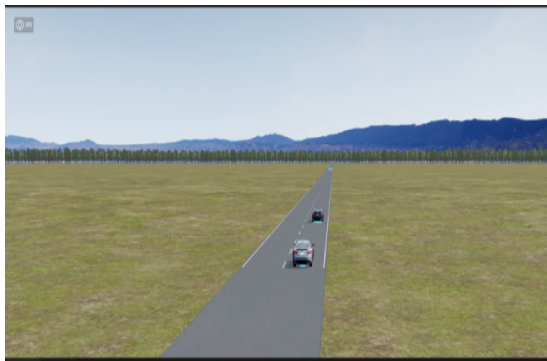


Figure 2.4: Extra-urban scenario generated with VI WorldSim.

- **Highway**

If high speeds have to be tested, then this is the scenario which is usually used (figure 2.5). Interaction between vehicles at different speeds requires different environments and maps. Let us think at Lane Keeping systems or analysis of a Cruise Control performances: both these things could require long tracks and high speeds, which an highway is able to satisfy.



Figure 2.5: Highway scenario generated with VI WorldSim.

Actually the environment is not the only feature that is important in the realization of a simulation's scenario. *Weather conditions and time of the day* are important too, in order to be as much realistic as possible.

This is particularly important for ADASs' sensors and cameras, which obviously will not always encounter perfect conditions in terms of light and possible interferences (rain, fog etc..) which may cause problems in the detection of agents, lane markings, and so on.

NCAP The European New Car Assessment Programme (Euro NCAP) is a voluntary vehicle safety rating system. Since the birth of first ADASs to help human drivers in order to reduce as much as possible (or even prevent) accidents, NCAP has started to consider these systems too in the rating of a car safety factor.

When we talk about ADASs, up to now Euro NCAP tests the following ones (which are the most used ones at the moments): *Autonomous Emergency Braking (AEB)*, *Lane Keeping Assist (LKA)*, *Adaptive Cruise Control (ACC)* and *Monitoring systems* (check for safety belt, check for degree of attention of the driver and so on).

The first two requires some specific scenarios for tests and of course simulations take the lead even here. In particular up to now 5 scenarios are the mainly required ones:

- **AEB:**

- Ego vehicle which approaches a standing still vehicle.
- Ego vehicle which approaches a vehicle with a lower constant speed.
- Ego vehicle driving behind a vehicle which suddenly (actually different values of deceleration are tested) brakes down to zero speed
- Ego vehicle which has to turn left, with an approaching vehicle in front of it.

- **LKA:**

- Ego vehicle has to maintain a certain direction in order to keep going within the road's markings.

These scenario are therefore tested before on the simulator and then, when the results are good enough, are tried on the road.

2.2.2 Degree of accuracy in editing of Ego vehicle

One of the multiple purposes of a driving simulation is to investigate the functioning of a specific part of a vehicle inside a virtual world, before actually test it on road; this is very helpful in terms of development of vehicle's parts.

Therefore the editing of the vehicle is a key element during the setting of a simulation. Let us say we need to test for example an electric car or an internal combustion engine one, or maybe we need to act on some wheel friction parameters or in general the dynamic and aerodynamic structure of the car.

The accuracy in the editing of the vehicle depends on the model the software is based on. Many softwares are automatically integrated with MATLAB& Simulink and are characterized by a Simulink model (an example is depicted in figure 2.6); the number of blocks is of course representative of the degree of accuracy.

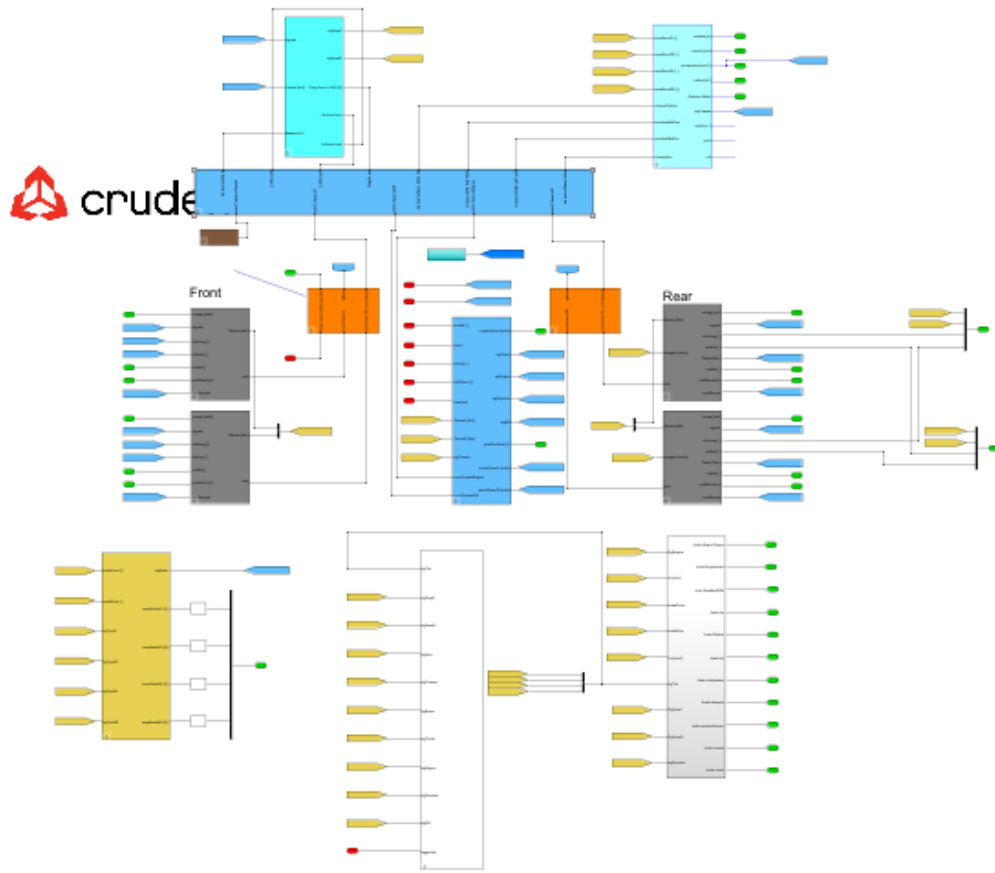


Figure 2.6: Simulink model of a vehicle; here the most external blocks.

2.2.3 Data logging and Real-Time analysis

The whole concept of the simulation relies of course on the real-time analysis or on the post-processing of some parameters, which are controlled in order to understand if the design is correct or if it has to be modified.

In order to do so, we need to be able to log the data of the simulation. The importance of this procedure leads to the need of a fast and (if possible) straightforward way of receive (and eventually process) these data; furthermore the amount and the types of parameters that can be obtained are considered too in the definition of the “quality” of logging of a software.

Let us make an example. If the vehicle dynamic behavior is based on a Simulink model, then the user can easily plot data by means of scope or save them in order to be able to perform a post process. In the following an example of retrievable data is shown.

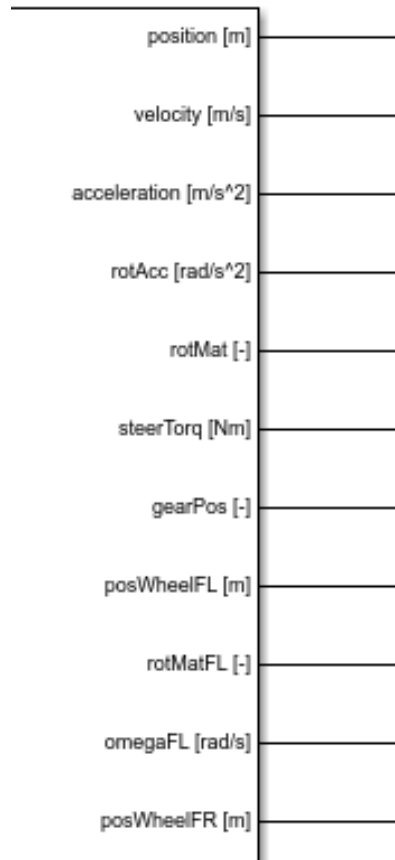


Figure 2.7: Data logging from a simulink model.

2.2.4 Integrability of the software

The chance to connect and integrate the simulation’s software with second and third parts is another important characteristic which is usually checked.

As a matter of fact, before, during and after the simulation, we may have the need to make the software communicate with others. Let us think at implementing control logics with Matlab& Simulink, or using an external vehicle model, or again an external traffic model and so on. The possibility and degree of difficulty of doing so is therefore critical and, as we will see, it could be a great deal.

2.2.5 Possibility of ADASs' tests

Lastly we find the possible need of performing some tests on ADASs, in order to check control logics before testing them directly on the road. This obviously requires the use of some cameras and sensors in order to detect agents of the scenario and obtain data from the simulation. In the following you find a list of some of the mainly used systems when we deal with ADAS:

- **GPS:** the position in time of the ego vehicle is obtained.

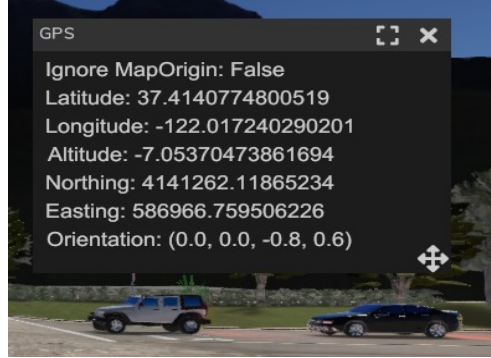


Figure 2.8: GPS sensor

- **Radar:** it stands for “Radio Detection and Ranging” and it is a detection system which is made up by a *transmitter* which produces electromagnetic waves (in radio and microwaves domains), an antenna which at the same time sends the waves and receives them back. According to the power of the Radar, it can detect objects at different distances and in general it is able to detect values like displacements, velocities, accelerations and even the type of object that has been hit.

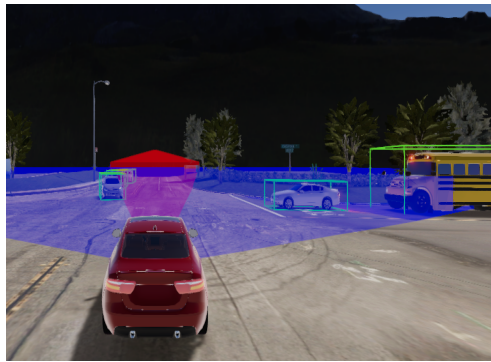


Figure 2.9: Radar sensor

The working principle of radar is to send and receive (after the reflection on the target) electromagnetic waves being able to compute, knowing the time of the response, the distance and the velocity of the hit object. The main problem with radar is that only metallic materials reflect properly the wave, while non metallic ones aren't able to do so at all (or in a imperfect way) so only some agents can be detect in this way. The problem is solved using LiDAR (which is investigated at the next subsection), which using a different wave is able to detect far more materials in a more precise way and smaller objects: the reason behind this is the wavelength (small wavelength is able to detect small objects and viceversa).

- **LiDAR:** it stands for “Light Detection and Ranging” and it works basically as the Radar with the difference in the kind of wave that is used, which now is a Laser. The density and the wavelength of the laser, as previously said, solve the problem of radar. In the following picture, the effect of LiDAR is represented and as you can see it is shown in a different way with respect to the Radar.



Figure 2.10: LiDAR sensor

But then the question is: why is Radar still used? There are many reasons behind this choice which are not investigated here: just to give a very brief explanation, let us consider that the LiDAR is relatively more expensive and it has the feature of detecting even very small objects, which can be a pro or a con: if we think at fog, snow, rain and so on, some LiDAR are able to detect these environmental agents, thus leading to an imperfect detection of what is really important.

A coupling of the two would be the perfect option, but this could not be possible for some reasons (cost, room on the vehicle and others).

- **Depth Camera:** it is one of the many possible cameras we can find on a ADAS. It returns an image which is made up only by shades on the gray scale, where the darker the gray the closer is the object to the camera, and viceversa.



Figure 2.11: Depth camera.

- **Semantic segmentation Camera:** it returns an image where each agent in the simulation's scenario (vehicles, pedestrians, animals, trees, roads and so on) are represented by a certain color.



Figure 2.12: Semantic segmentation camera

- **Virtual camera:** the name is quite general. It usually describes a camera which is able to detect obstacles, lane markings and traffic signs.
- **Ultrasonic sensor:** as the name can suggest its working principle is based on the use of sound waves, which are produced and received. Considering the amount of time that the sound takes to come back, the distance is measured.

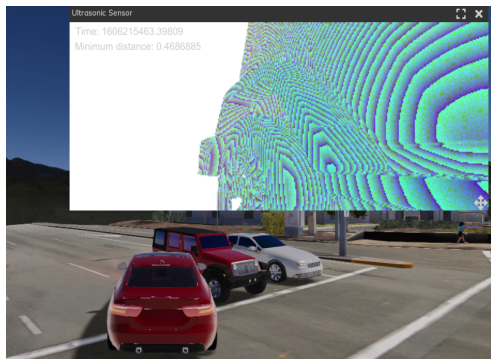


Figure 2.13: Ultrasonic sensor effect.

This sensor is usually used for short distances and low speeds, like parking situations.

2.3 Vehicle models used for simulations

Before getting involved with driving simulations, we need to understand which are the possible ways to represent the vehicle inside a simulation.

The first and main choice to be made is between a *kinematic* model and a *dynamic* one. The first one does not take into account the inertia of the vehicle and therefore is effective at low speeds only. On the contrary the dynamic model takes into account the inertia, thus leading to a more complicated, but even accurate, model which is able to handle more realistic dynamics.

2.3.1 Kinematic models

Let us consider a car with a velocity $v(t)$. Its kinematic motion can be described by two parameters, which are described in the following picture:

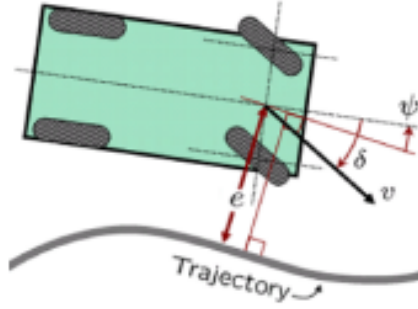


Figure 2.14: Kinematic model of a vehicle.

- The cross-track error $e(t)$, which represents the distance of the front wheels axle middle point (see figure 2.14).
- The difference in angle between the front wheels and the nearest segment of trajectory to be followed:

$$\psi(t) - \delta(t) \quad (2.1)$$

where $\psi(t)$ is the yaw angle and $\delta(t)$ is the angle between the front wheels and the vehicle.

These are the only values to be considered when we need to model the kinematic behavior of a vehicle and, for example, when we need to apply some control (ADAS) on it.

2.3.2 Dynamic model

The dynamic model keeps into account even the inertia of the vehicle. In particular, the effect of the tire slip and the steering servo motor are considered. The general scheme is shown in the following:

The effect of *tire slip* is considered in the following way: each tire is modeled to produce a force which is perpendicular to the rolling direction and proportional to the angle α , which is the inclination of the local velocity with respect to the forward velocity of the vehicle:

$$\begin{cases} F_{yf}(t) \approx -C_y \alpha_f(t) \\ F_{yr}(t) \approx -C_y \alpha_r(t) \end{cases} \quad (2.2)$$

where C_y is the cornering stiffness of the tire. Furthermore we have:

$$\begin{cases} \alpha_f(t) = \arctan\left(\frac{U_y(t) + r(t)a}{U_x(t)}\right) + \delta(t) \\ \alpha_r(t) = \arctan\left(\frac{U_y(t) - r(t)b}{U_x(t)}\right) \end{cases} \quad (2.3)$$

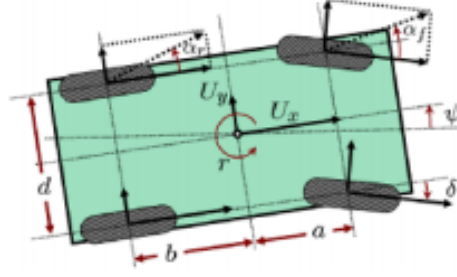


Figure 2.15: Dynamic model of a vehicle.

where $U_x(t)$ and $U_y(t)$ are longitudinal and lateral velocities of the vehicle. The differential equations of motion are:

$$\begin{cases} m(\dot{U}_x(t) - r(t)U_y(t)) = F_{xr} + F_{xf} \cos(\delta(t)) + F_{yf} \sin(\delta(t)) \\ m(\dot{U}_y(t) + r(t)U_x(t)) = F_{yr} - F_{xf} \sin(\delta(t)) + F_{yf} \cos(\delta(t)) \\ I_z \dot{r}(t) = -aF_{xf} \sin(\delta(t)) + aF_{yf}(t) \cos(\delta) - bF_{yr}(t) \end{cases} \quad (2.4)$$

Dynamic models: different degrees of freedom Let us recall that in the case example seen before, a bicycle model was considered.

Different models are characterized by a different number of degrees of freedom thus leading to different the accuracy of the model and therefore in the limited possible applications. At the same time we need to take into account the specific application. If, for example, only the lateral behavior of the vehicle is to be analyzed, then even a simple two degrees of freedom bicycle model is enough; on the other hand, if the longitudinal dynamic of the car has to be taken into account, then we need an additional degree of freedom (in this case the bicycle model is still enough).

Many different dynamic models are therefore available depending on the application, up to the most accurate ones which have to investigate in deep the dynamic behavior of the car, being therefore characterized by more than 10 dofs (VI Car Real Time, as we will see, exploits a vehicle model based on 14 degrees of freedom).

Chapter 3

LGSVL vs VI-WorldSim

3.1 Generalities

During my stage and my thesis project I had the chance of analyzing and studying two driving simulator softwares: LGSVL, which has been developed by LG and which stands for LG Silicon Valley Lab, and VI-WorldSim, the ultimate simulation’s software belonging to VI-Grade group.

In the following sections, I will describe these softwares, not from a “how-to-use” point of view, but investigating the main features (which has been discussed in the previous chapter) which characterize them. In this way the reader can have an understanding of how the same things can be performed in such different ways and, more importantly, how this affects the performances (and therefore the evaluation) of a software.

Before going on with the documentation, a crucial difference has to be underlined: LGSVL is an open-source software, while VI-WorldSim is obtained and used by means of a license. As you may have already realized, this affects a lot the possibilities and the degree of difficulty of use of the software.

As you will see, this chapter is organized in the following way: in the first two sections you will go through an explanation of how to set the scenario and perform a simulation, first with LGSVL and then with VI-WorldSim; after that, a summary is proposed, in order to be able to understand the actual differences between the two softwares.

3.2 LGSVL

As already mentioned, LG developed this software (release 2020.06) in order to allow users to perform simulations inside virtual scenarios in a totally free way. Of course this leads to some limitations, as you will understand.



Figure 3.1: Example scenario from LGSVL.

3.2.1 Additional requisites

Before actually explaining the procedure to properly set up a simulation, we need to mention two additional parties, which are automatically integrated with LGSVL and whom usage is mandatory (or at least for some applications of the software): *Unity Editor* and *Apollo*.

Unity Editor

Unity is a real-time development platform, which is used in the video-games field and LGSVL exploits its graphic engine.

LGSVL comes for free with some environments which had already been implemented and developed by means of Unity. These maps therefore can be edited with the same program and you can even create new ones from scratch. Even though Unity is not an open source software, these basic operations can be performed using the Student or Personal license, which comes for free.

Of course the fact that the Student version of Unity is more than sufficient for being able to *edit/create* maps, vehicles and so on does not cut out the main problem: the necessity of particular skills in the usage of this software. In the following figure is represented the Unity Editor interface.

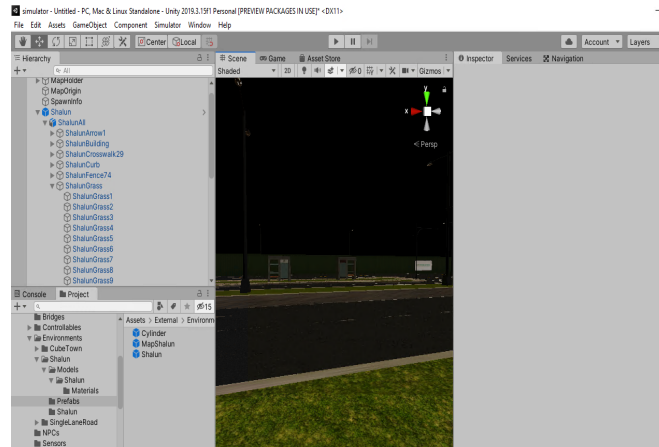


Figure 3.2: Unity Interface.

Apollo

Apollo is an open source platform which is used (along with Autoware, not investigated here) during the *testing of ADAS* applications; path planning and route editing are possible, but also the activation of sensors and their visualization and so on.



Figure 3.3: Graphical User Interface of Apollo.

Apollo comes inside a Docker container, which means, after the cloning of the repository is performed, the container has to be started and entered each time we want to use the platform.

3.2.2 Simulation setup

Now we are ready to go through all the steps which are needed to set up the simulation. In this way you can actually perceive which are the limitations and the weak points of a free-to-use simulator software.

The simulation is set up by means of the Web User Interface, which appears like this:

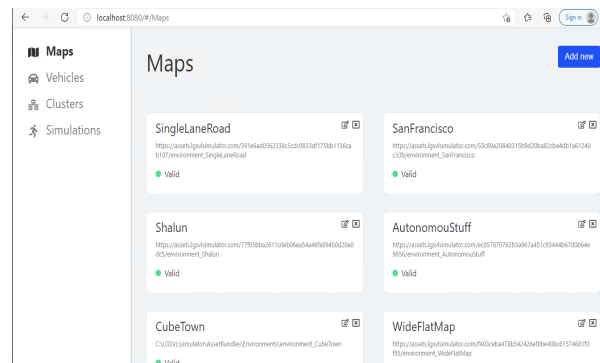


Figure 3.4: Web User Interface

On the left of the picture 3.4 you can select “Maps”, “Vehicles” and so on in order to set that feature of the simulation.

Before actually starting with the setting of the simulation, the term **AssetBundle** (AB) has to be introduced: it is a file which needs to be uploaded on the WebUI in order to be available for the simulation (we are talking about vehicles, maps, sensors etc). Some assetbundles are already available at <https://content.lgsvlsimulator.com/> so we just need to add them on the WebUI, but we may need to generate others from scratch by means of Unity Editor (procedure which, as already stated, requires non-basic skills though).

Getting the correct map

The map or environment plays, of course, a crucial role in the simulation. As introduced in the previous chapter, we can distinguish 3 main environments: the urban scenario, which basically represents the city (traffic lights, stops, traffic around you, roundabout and so on), the extra-urban scenario, representing a possible road out of the city (which is not an highway) and lastly the highway scenario.

The scenario's choice depends of course on what we need to reproduce and simulate, so we need to understand if we need traffic around the ego vehicle and, if present, the kind of traffic, the speeds (city environment of course is characterized by lower speeds with respect to the highway one), if it is needed pedestrians action and so on.

Furthermore, as mentioned in the chapter 2, different standard scenarios are required to be analyzed when we are dealing with safety rating of a vehicle according to Euro NCAP. This reflects in different maps and environments to be used.

From the WebUI, under the "Maps" tab, you can add new maps (new ABs of maps) by clicking "Add new":

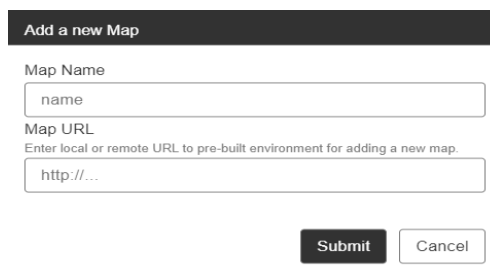


Figure 3.5: Adding new map to the User Interface

You need to decide the *Map Name* and to add the *url* of the assetbundle. The latter can be a remote url, if we simply take it from the contents of LGSVL Simulator which are available at the official web site, or a local url which indicates the location (the path) of the assetbundle on the machine you are using: this is the case in which you have generated new assetbundles from Unity Editor and save them in a specific folder.

If you want to take the AB from <https://content.lgsvlsimulator.com/maps/>, you need to select the map we desire at the site. In the new page then, you right-click "Asset Bundle" and copy and paste the url in the specific line in the WebUI.

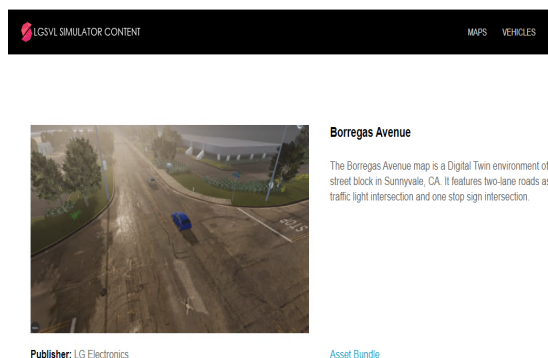


Figure 3.6: LGSVL Simulator contents: Borregas Avenue

LGSVL is characterized by a few urban scenarios, the Indianapolis circuit and the basic straight road for simple tests (like the one we have introduced talking about NCAP); these are the ones which comes for free from the content site. Of course, if needed, one can think about adding more maps, but this is possible in two ways: buying new maps downloading them from Unity Store, thus involving costs, or creating new ones from scratch, thus requiring advanced skills as Unity project developer.

Real **problems** come when the user needs to perform an **editing** of the map (changing road dimensions, changing position of traffic lights and so on). Let us say for a moment that we have quite advanced skills as a Unity developer, we still need the AssetBundle of the map we want to edit, since inside it there are all the necessary files (textures etc) for performing the editing on Unity Editor, and this is not always available. The necessary files are found on GitHub and actually only three maps are available there with all the necessary files: *Shalun* (a non-digital twin version on Taiwan Car Lab Testing Facility) and *SingleLaneRoad*. This means the two mentioned maps, represented below, are the only ones (at the moment) that are available to be edited.



(a) *Shalun*



(b) *SingleLaneRoad*

Figure 3.7: LGSVL map examples

Editing of the scenario

Once we have obtained and select the correct environment for our simulation, we need to make sure the scenario satisfies all our requirements.

When we talk about a simulation scenario, we take into account all the agents surrounding the Ego Vehicle, the weather conditions and the time of the day at which performing the simulation (thus resulting in different lighting conditions).

LGSVL Simulator's GUI allows the user to change all these conditions in a very easy and fast way, even if the traffic is characterized simply by a on/off condition, thus from the interface is not possible to choose exactly the path followed by NPC vehicles (and pedestrians) for example.

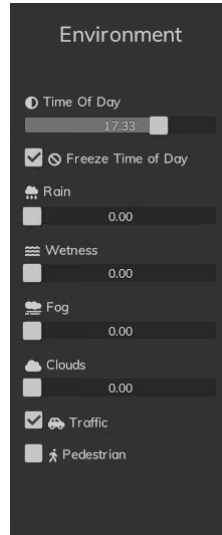


Figure 3.8: Graphic User Interface of LGSVL Simulator

Editing the Ego vehicle

Once the environment and the scenario is properly set, we need to add the correct ego vehicle, in terms of vehicle dynamics, vehicle aerodynamics, engine type (electric, hybrid, internal combustion engine, etc) and so on. Of course the user would like to have the most editable vehicle possible: the higher the degree of editability, the better, since more configurations are available to be analyzed in a simulation.

The general procedure that has to be followed is pretty much the same as for maps: we need to add vehicles at the WebUI by knowing the url to the AssetBundle. Some vehicles are totally for free and their AB can be obtained at the content site: here we find a bunch of vehicles, but only one of them can be found on GitHub (so only one AB can be downloaded actually) thus only one vehicle can be edited on Unity Editor.

Then the thing is always the same: if needed you can download additional vehicles from the Store, but that requires money. Furthermore, the JaguarXE2015 (the editable vehicle) is a pretty simple model and the parameters that can be modified are shown in the following:

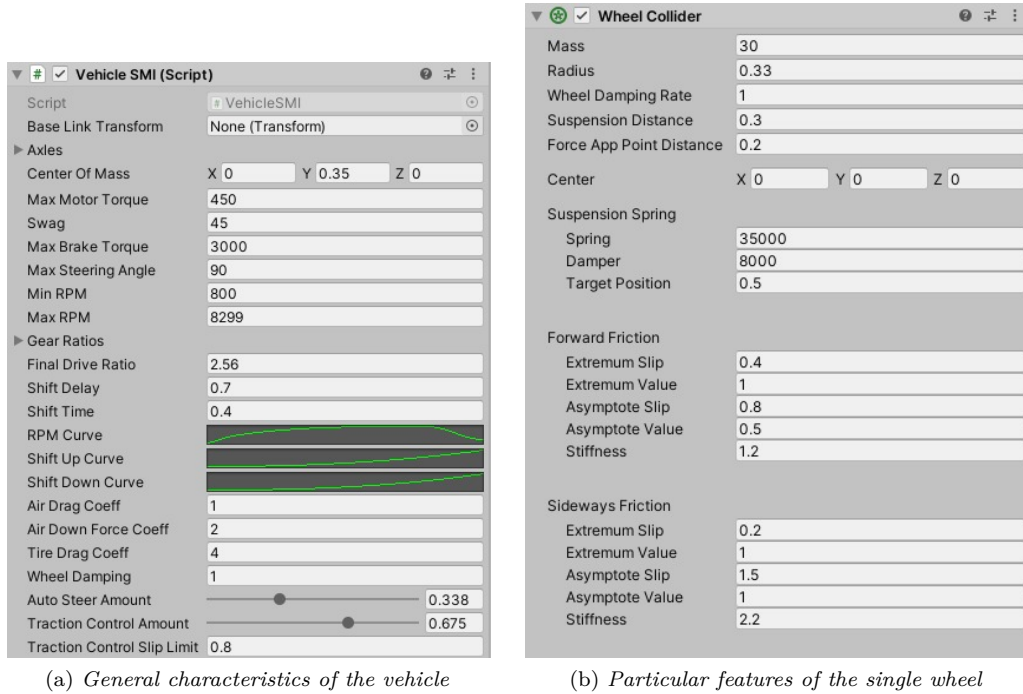


Figure 3.9: Editing possibilities

In figure a some general characteristics of the vehicle can be edited, like the total mass, the position of the center of mass and so on; in figure b instead, particular features of the single wheel can be edited (stiffness, damping action and so on). Pay attention to these parameters because they are the only ones that can be edited for free inside LGSVL, up to now.

Of course different vehicle's model would have different (and maybe more) parameters that can be set.

Functional Mock-Up Interface The FMI (Functional Mock-up Interface) is a free standard which is compatible with many softwares; it has been developed on purpose to have compliance between different platforms and being able, for example, to link dynamics models with fluid-dynamics or aerodynamics ones and so on.

LGSVL offers the opportunity of using personal dynamic models for the vehicle using an FMU (Functional Mock-up Unit). Actually though, at the moment, this function does not work properly and LGSVL support told me the bug is going to be fixed in the next release of the software, but still the possibility is quite interesting. Simulink model could be implemented and used, which would improve the degree of accuracy in the editing of the ego vehicle and the potentiality of LGSVL.

Adding sensors and cameras

Once the vehicle is properly edited, we can generate its assetbundle on Unity and add it at the WebUI. Other than the room for the url and the name of the vehicle, we can select the *bridge type* which states the kind of platform we are going to use to perform path planning and route editing (Apollo or Autoware) and the *sensor configuration* to be added at the vehicle: it is basically made up by a json code (figure 3.10), in which we need to specify the sensors we want and their features (position on the vehicle, etc).

Lexus2016RXHybrid (Autoware) Configuration

Bridge Type

ROS

Sensors

Requires json format

```
[{"type": "GPS Device", "name": "GPS",
  "params": {"Frequency": 12.5, "Topic":
"/nmea_sentence", "Frame": "gps", "IgnoreMapOrigin":
true},
  "transform": {"x": 0, "y": 0, "z": 0,
"pitch": 0, "yaw": 0, "roll": 0}}, {"type": "GPS
Odometry", "name": "GPS Odometry",
  "params": {"Frequency": 12.5, "Topic":
"/odom", "Frame": "gps", "IgnoreMapOrigin": true},
  "transform": {"x": 0, "y": 0, "z": 0,
"pitch": 0, "yaw": 0, "roll": 0}}, {"type": "IMU",
"name": "IMU",
  "params": {"Topic": "/imu_raw", "Frame":
"imu"},
  "transform": {"x": 0, "y": 0, "z": 0,
"pitch": 0, "yaw": 0, "roll": 0}}, {"type": "Lidar",
"name": "Lidar",
  "params": {"LaserCount": 32,
"MinDistance": 0.5, "MaxDistance": 100,
"RotationFrequency": 10, "MeasurementsPerRotation":
360, "FieldOfView": 41.33, "CenterAngle": 10,
```

Submit Cancel

Figure 3.10: Ego vehicle configuration on the WebUI

The json code needs to be written down properly (this is mandatory), following the structure reported here below:

```
[
SENSOR,
SENSOR,
SENSOR
]
```

Let us remember though that sensors can always be added, taken away or modified later on by means of the Python API.

3.2.3 Data Logging

LGSVL comes with the possibility of using a **Python Application Programming Interface (API)** which can be used to edit the configuration of a loaded simulation: we can add agents, make them move around the map (that is the only way to do so), change scenario's conditions and more importantly retrieve sensor configuration and data and more.

In the following you find an example of a very simple code written in python, thus exploiting the API:

```
import lgsvl
import os

sim = lgsvl.Simulator("localhost", 8181)
sim.load("BorregasAve")

spawns = sim.get_spawn()
state = lgsvl.AgentState()
state.transform = spawns[0]
ego = sim.add_agent("Jaguar2015XE (Apollo 3.0)", lgsvl.AgentType.EGO, state = s

c = lgsvl.VehicleControl()
c.throttle = 0.2
ego.apply_control(c, True)

sim.run(5)

sensors = ego.get_sensors()
for s in sensors:
    if s.name == "GPS":
        dati_gps = s.data
```

Basically what is done here is to load the scene, choose the “Borregas Avenue” as map, add the “Jaguar2015XE” as Ego vehicle in a specific position and give it a certain speed; then knowing that the ego car had a GPS sensor, we take its data in order to be able to post process them.

We need to underline the fact that using the Python API is the only way to set a certain NPC path or anything like that. This means a (basic) knowledge of programming language is required and the documentation with all classes and functions has to be available each time we need to set the simulation. This can be quite annoying, but we have to remember that this software is completely open source, thus we can expect something like this.

Finally, let us recall that the only data which are available for the data logging by means of the API are positions, velocities and accelerations of the vehicles. Other information may be retrieved by means of the Apollo GUI (see in the next chapter), but for a sake of time I did not have the chance of studying its functioning.

3.2.4 Integration with other platforms: path planning and route editing with Apollo 5.0

LGSVL allows the integration of the Simulator with Autonomous Driving (AD) platforms, in particular Autoware and Apollo. During my stage I analyzed **Apollo** (in particular the version 5.0), but despite of the platform (both are characterized by a WebUI just like LGSVL) they require a *Linux OS*.

The procedure could seem long and complicate, but it is actually quite straightforward. After the setup and installation procedure (Apollo is designed to run out of docker containers), which is not reported here and can found at the official site of LGSVL, the steps to be followed are:

1. Navigate to the folder `apollo-5.0/docker/scripts` and open a terminal from it.
2. Start the container in which you have Apollo by typing:

```
./dev_start.sh
```

3. Enter the container:

```
./dev_into.sh
```

4. Start Apollo:

```
bootstrap.sh
```

5. Launch the bridge:

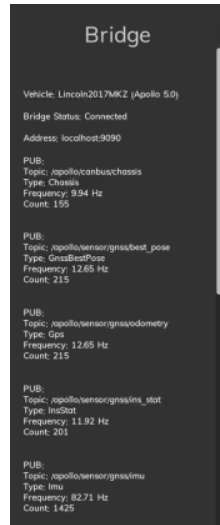
```
bridge.sh
```

Right now the end of the terminal should look like this:



```
[ OK ] Finished setting up Apollo docker environment. Now you can enter with:
bash docker/scripts/dev_into.sh
[ OK ] Enjoy!
addfor@apollo-5.0/docker/scripts$ ./dev_into.sh
addfor@ln_dev_docker:/apollo$ bootstrap.sh
nohup: appending output to 'nohup.out'
nohup: appending output to 'nohup.out'
Launched module monitor.
nohup: appending output to 'nohup.out'
Launched module dreamview.
dreamview is running at http://localhost:8888
addfor@ln_dev_docker:/apollo$ bridge.sh
WARNING: Logging before InitGoogleLogging() is written to STDERR
t1204 09:46:02.408560 43002 global_data.cc:150] [cyber bridge] host ip: 10.0.1.8
```


Furthermore if you select the Simulator window, in the Bridge tab you should see the connection has been established:



We can even open a new terminal, enter the apollo docker and see the published channels by running:

```
cyber_monitor
```

The result should be something of the type:

Channels	FrameRatio
/apollo/canbus/chassis	0.95
/apollo/control	0.00
/apollo/control/cmd	0.00
/apollo/drive_event	0.00
/apollo/hmi/auto_capture	0.00
/apollo/hmi/status	0.00
/apollo/localization/mf_status	0.00
/apollo/localization/pos	0.00
/apollo/monitor	2.00
/apollo/monitor/system_status	0.00
/apollo/navigation	0.00
/apollo/perception/obstacles	0.00
/apollo/perception/traffic_light	0.00
/apollo/planning	0.00
/apollo/precognition	0.00
/apollo/relative_map	0.00
/apollo/routing_request	0.00
/apollo/routing_response	0.00
/apollo/sensor/camera/front_12m/image	12.00
/apollo/sensor/camera/front_12m/image/compressed	12.94
/apollo/sensor/camera/front_30m/image	14.00
/apollo/sensor/camera/front_30m/image/compressed	14.00
/apollo/sensor/camera/left_fisheye/image	0.00

Figure 3.11: Monitor for data streaming check.

- Launch the simulator, click Open Browser and setup the simulation. At first you need to use a vehicle which is characterized by a **CyberRT** bridge type (allows connection to Apollo 5.0); then there is to set properly the **Bridge Connection string** you find next to the vehicle name in the dedicated part where you select the car for your simulation: it has to be "localhost:9090".
- Open **Dreamview** (the WebUI of Apollo); this can be done directly from the terminal, by opening the link you have seen printed out right after launching Apollo.
- Inside the application select the vehicle and map you have chosen for the simulation.

9. Now, selecting the **Module Controller** tab, we can add whatever thing we would like, for example:

- Localization
- Transform
- Perception
- Traffic Light
- Planning
- Prediction
- Routing
- Control

In case you chose the Lincoln2017MKZ in BorregasAve, with those elements in Dreamview you should see something like:

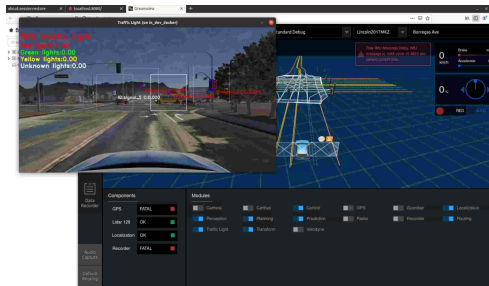


Figure 3.12: Apollo Dreamview

10. Now you can exploit all Apollo 5.0 functionalities, like Route Editing and so on.
11. When you have finished, in order to stop the container you need to type (in a new terminal opened from the same folder of step 1):

```
./dev_start.sh stop
```

3.3 VI-WorldSim

VI-WorldSim is the ultimate driving simulator software which has been developed by **VI-grade**.

One of the strengths of the VI-Grade world is for sure the ramification in different software packages for the various aspects of the simulation. This actually introduces also higher costs, but as a final result the simulation can be edited and set up in a very precise and accurate way.

As all the other solutions offered by this company, WorldSim requires its own license which, added to others, makes up a quite complex but perfectly interconnected processes architecture. As you will realize in the following subsections, being WorldSim based on a commercial license leads to a straightforward procedure to set the simulation up and other advantages like the full integration with the VI-grade world, thus meaning, if present, the possibility of connecting this software with all the other platforms (Vi-CarRalTime, VI-Controller, VI-DriveSim and so on).

3.3.1 Soft architecture of the Static Simulator at Nichelino Plant

In order to be able to describe WorldSim, first we need to analyze the soft architecture of the simulator which has been used for my studies. It is made up by different processes, which are run on different machines (6 in particular). In the following picture the 6 total machines are represented:



Figure 3.13: Rack with all machines assessing for Static Simulator correct functioning

The hardware architecture will be described later on, for now let us just explain the soft part. Many processes are involved in the running of a simulation; these processes are added and edited on the so called Controller platform.

- Visual processes: one for each part (right, center, left) of the 7 meter curved screen which is used to project the simulation, plus one to display the scenario on an additional machine to check the simulation and perform several post processing operations.
- Mirrors processes: one for each mirror of the simulator (three mirrors for right, left and center)
- Sound process: the process assessing for the audio of the simulation (in this case it is run by the mirror's machine).

- Sensor process: the process in charge of retrieving information from WorldSim’s sensors.

As the reader may have perceived, this is a quite complex architecture; in addition to that, we need to account for power consumption of the overall simulator, which is quite high in terms of electricity. Despite these two drawbacks though, the result is quite impressive in terms of immersion of the driver inside the simulation.

3.3.2 VI-WorldSim Studio

WorldSim Studio is the GUI and it can be defined as very easy-to-use. It allows to set the scenario for the simulation in all its aspects.

Map

Up to now, we can find the following environments:

- **MCity**: it represents an urban scenario. It is characterized by the usual elements like crossings, traffic-lights, parking spots and so on, plus a roundabout (this last item has to be underlined since nowadays it represents one of the main problems when talking about ADASs, due to the fact that the Ego vehicle encounters other ones from different directions).



Figure 3.14: WorldSim MCity map

The only drawback of this scenario is that there are not *T junction* with blind spots; of course they can manually be generated by adding vehicles around the map which limit the sight of the driver, but still it represents a drawback.

- **Neighborhood:** it is another urban scenario, but it is different from a structural point of view. Now we have a district where all the streets are perpendicular to each other and houses are present too, thus resulting in the perfect scenario for simulating T junction of pedestrian crossings with blind spots caused by walls.

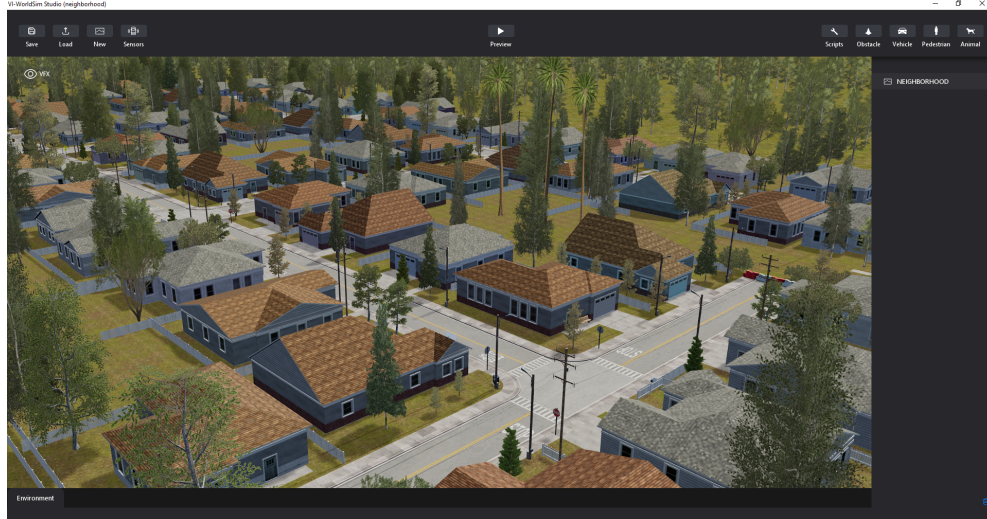


Figure 3.15: WorldSim Neighborhood map

- **Extra-urban road:** as the name may suggest it is the typical extra-urban road, which can be found in different versions (only straight road, straight road with perpendicular road and so on).

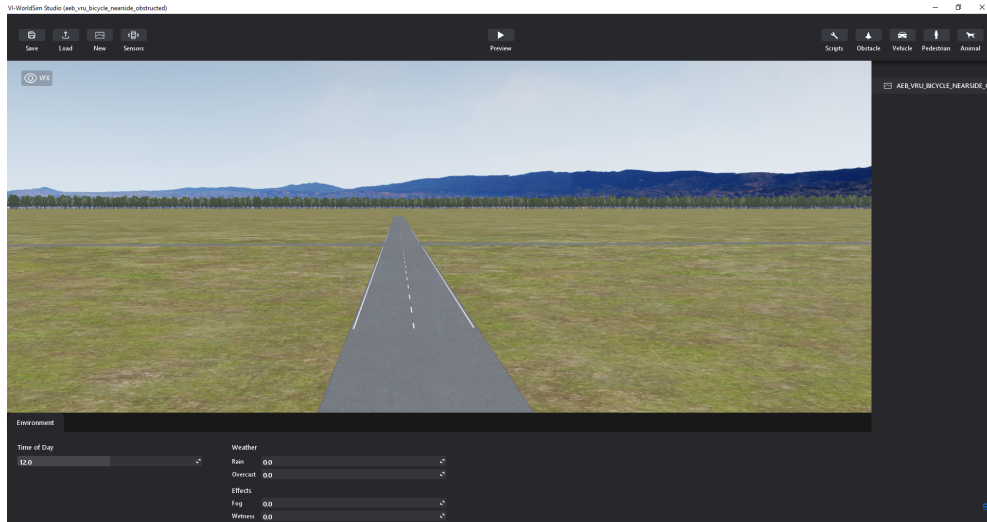


Figure 3.16: WorldSim extraurban map

This environment is perfect for testing the vehicle according to NCAP rules.

- **Highway:** finally we have the highway (the typical German *autobahn*) scenario, which can be exploited in different versions like the simple one, the one with an exit, the one with an entrance and so on, to be able to analyze the most important situations of the scenario.

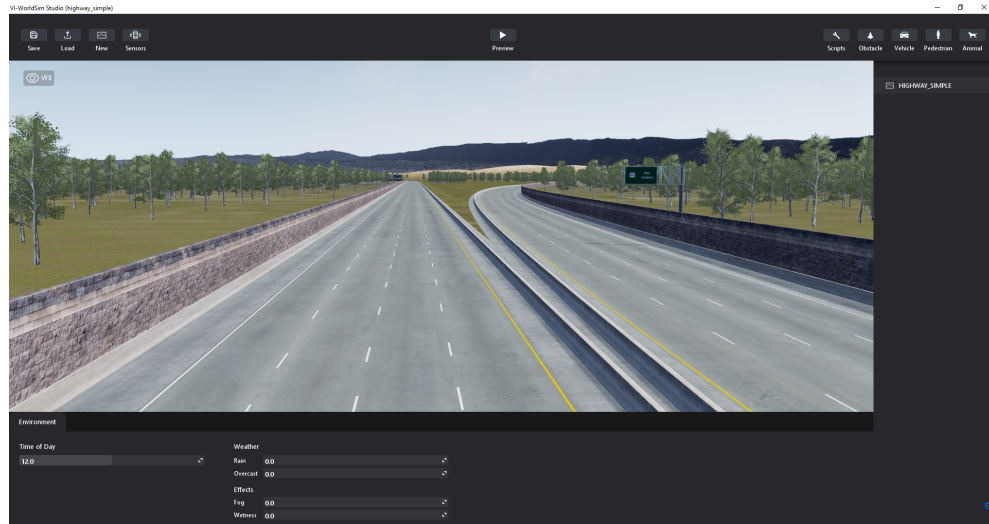


Figure 3.17: WorldSim highway map

One of the biggest drawbacks of WorldSim is that, as LGSVL, it has an external editor for maps and environments. Up to now RoadRunner is required (it comes with a license) if some modifications are needed, like changing number of lanes, changing their width, adding traffic-lights and so on, or if we want to create our own map.

Editing the scenario

As already introduced the GUI of WorldSim Studio is quite easy to use.

First of all, the **addition of agents** is very easy and is performed by a single click. Many types of vehicles, pedestrians, animals and even obstacles are present and for each of the dynamic agents the user has to decide the behavior: deactivated, wander mode, user-controlled mode, reference trajectory, destination route and teleport are available (the user-controlled behavior only for the vehicles of course).

As you may imagine, this leads to the fact that the behavior of each agent is decided and edited by means of the GUI, avoiding completely any programming skills, required for example in LGSVL Simulator software. Then different "Script' Components" are available and be careful that the name is only indicative, it does not mean the user has to write scripts of anything. In order to make an example, *triggers* can be used to perform actions of make some agent do something when that specific event occurs; then *way points* can be added to specify the route destination for vehicles or pedestrians or even animals.



Figure 3.18: Triggers example

The **weather conditions** and **time of the day** can be easily set on the GUI just like LGSVL Simulator did.

Finally the set of **sensors** present on the ego vehicle can be edited and added on the car. The user can select the type of sensor it needs and then modify some properties which depend on the sensor (like the position on the vehicle, its orientation, the pixel to be used in the two directions and so on).

Editing of the ego vehicle - VICRT

As previously introduced, VI-Grade has distributed in several software packages the different aspects of the simulation. In case of the editing of the vehicle, the user needs to make use of the aforementioned *VI-CarRealTime* software. This program allows a very specific editing of the car, thus leading to a higher degree of realism and, more importantly, to more solutions to be analyzed.

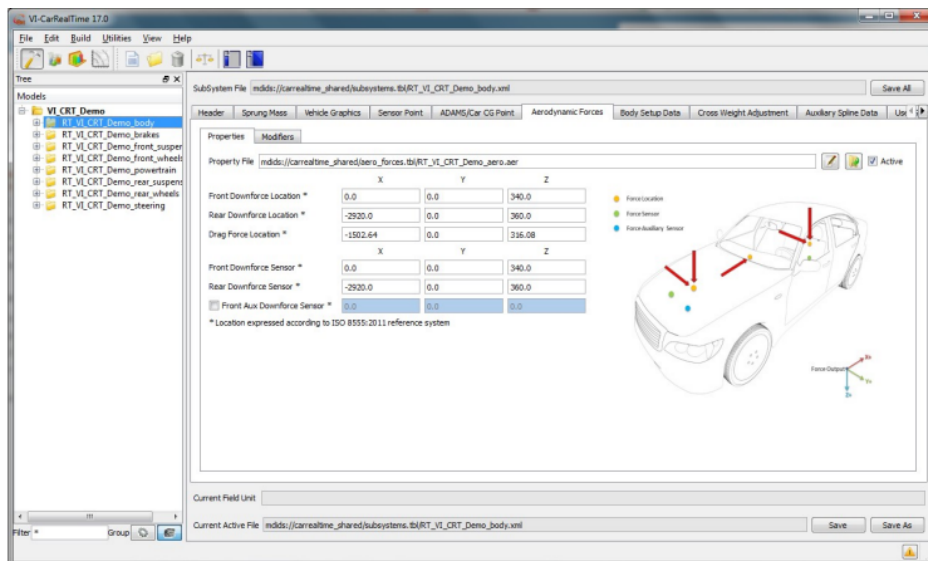


Figure 3.19: GUI of VI-CRT: aerodynamic forces setting

Adding sensors and cameras

The creation and addition of sensors are based on two steps.

1. First of all, you need to create a json file describing the sensors you want to add to your vehicle and attach the file to the ego car (in VI-WorldSim Studio, in the “sensor” section that comes out once you select the ego vehicle).

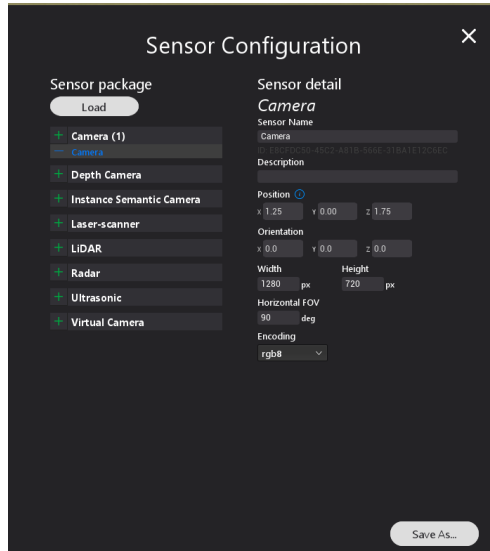


Figure 3.20: WorldSim sensor configuration

As you can see, the sensor is completely editable in terms of position on the vehicle; then other parameters can be set, starting from the name, up to the Field Of View and others (depending on the kind of sensor).

2. Once the json file of the sensor is created, you need to add it on the vehicle in the tab “Sensor” of the car.

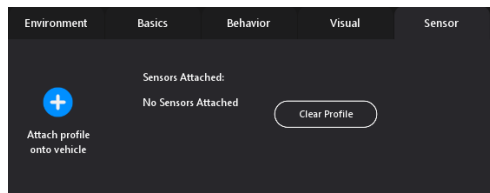


Figure 3.21: Sensor addition to the ego vehicle

3. As a last step, the user needs to add a Sensor process in the Controller (all the values of the process are specified in the documentation). In our case the process has been added at the Broadcast machine. It is recommended not to add it at machines running as visuals (so basically the ones accounting for the projectors and the mirrors), since this may lead to a drop in the performances (in the case of the visual broadcast we will allow a little drop in graphics).

Now WorldSim is able to create sensor’s signals on the machine on which the Sensor process is running and send this data by means of ZeroMQ messaging library over the network; these messages are then decoded by means of flatbuffers, a cross-platform serialization library.

By means of a Software Development Kit (SDK) is possible to receive and process sensors’ signals. Actually, there are three possible ways of doing so.

3.3.3 Integration of the software

Once the scenario is created and edited properly on WorldSim Studio, it can be tested and used.

The quality of the simulation of course depends on the settings of the simulator. VI-grade offers a wide list of possibilities, from the simple desktop-simulator with the use of a steering wheel and pedals only, up to the dynamic simulator which is a 9 degrees of freedom platform to simulate at best the loads on the vehicle and offers the best virtual driving experience possible.

Despite of the degree of accuracy in the simulation, the integration which is possible working inside the VI-grade world is so that the ego vehicle which is used inside the WorldSim scenario is taken directly from **VI CarRealTime** (if the license is present), thus leading to the possibility of exploiting the coupling between the two softwares (so the ego car can be edited in the best possible way).

At the same time the use of Controller coupled with DriveSim and WorkBench (software for the analysis, both in real time and afterwards, of the data logged by the sensors on the vehicle), thus meaning a complete control on the simulation and in particular the possibility of performing **Hardware-In-the-Loop (HIL)** analysis.

The user can even think at using compiled Simulink Models within the simulation, thus increasing the accuracy and the level of data logging. As the reader will see, a simulink model communicating via udp has been used in my thesis.

3.3.4 Real-Time analysis and Data Logging from the sensors

When the procedure of adding a sensor to the Ego vehicle is concluded, the extraction of data has to be implemented.

WorldSim is able to create sensor's signals on the machine on which the Sensor process is running and send this data by means of ZeroMQ messaging library over the network; these messages are then decoded by means of flatbuffers, a cross-platform serialization library.

By means of a Software Development Kit (SDK) is possible to receive and process sensors' signals. Actually, there are three possible ways of doing so.

Integrated SDK

The first (and more simple) way is exploiting the already integrated SDK inside the software Drive.

The only thing we need to do is open the RTDB Tool from the DriveSim GUI and select "WorldSim Signals". Inside it, you find the list of possible sensor that can be exploited and all their features: you need to select what you want from the list (only a subset of all the sensors are available in this way; possible changes will be made in the next releases) in order to add these signals to the RTDB channel. The RTDB signals can be analysed on SimWB in the tab Real-TimeViewer. As previously mentioned, this way allows you to receive and process signals from a subset of sensors. In particular:

- **GPS:** it retrieves information about the position (altitude, latitude and longitude) of the ego vehicle with time in meters.
- **Collision Sensor:** it generates a boolean, indicating 1 if a collision has occurred.
- **Virtual Camera:** this sensor is able to detect lane markings and generates the coefficients which are necessary for the lane fitting and informations regarding obstacles on the road; both these two will be investigated later on.
- **Radar:** it is a 64x7 that gets the first 64 targets in radar range. Each line of the matrix represents a point of the radar return and each point has 7 properties (columns); even this sensor will be investigated in the following chapters.

Pure SDK

This procedure requires some programming skills (in Python or C++).

Despite of the requirement in software development skills, it allows you to obtain signals from all the available sensors in WorldSim:

- *Camera*
- *Depth Camera*
- *Instance Semantic Camera*
- *Laser-Scanner*
- *LiDAR*
- *Ultrasonic*
- *Virtual Camera*

The user needs to run the main.py (in the case of Python) file which states where signals can be found and to edit the ExampleSubscriber.py properly, in order to get, for example, the RadarSubscriber.py. The “Subscriber” files are used to receive information from the net, while the “Publisher” ones can be exploited to publish something on the network.

Anyway, this procedure has not been investigated since the required knowledge in programming.

Robotic Operating System (ROS) Environment

A ROS is a group of software frameworks for Robot software development. It is open source and, since it is used a lot when talking about ADAS development, VI Grade decided to add a container which allows the user to have a direct plug-in towards ROS.

This container, therefore this environment, is used not only for the data logging and extraction in real-time, but allows even the visualization of the sensors data in runtime.

3.4 Final comparisons

Now that the two software packages have been described in terms of procedure for setting up the simulation, the reader may realize the actual difference between the two.

The license price for **VI-WorldSim** reflects in several **advantages**, which are reported below:

- *Required skills* for creating the scenario.
- *Time* for obtaining the scenario.
- *Ego vehicle editing potential* (VI-CRT).

The *integrability* the VI-grade world offers could be worth the cost. Nowadays, when we deal with automotive industries, simulations are becoming more and more relevant, therefore a structure of this kind could be of huge interest.

If, on the contrary, the user is looking for an open-source solution to test some simple ADASs logics (for example), then LGSVL Simulator's software could be more than enough if he/she is aware of the limitations.

In the following, a resuming table is shown:

\	<u>LGSVL (open source)</u>	WorldSim (€)
Editing environments	n/a --> Unity Editor	n/a --> RoadRunner (Mathworks) or other
Traffic (NPC)	Default (random behavior) or by means of Python API.	Default, reference trajectories, waypoints, triggers and so on (WorldSim Studio).
Weather conditions	Edited from the GUI when the simulation is running.	Edited from WorldSim Studio or from VI-Controller during the simulation.
Editing Ego vehicle	n/a --> Unity Editor, low level of accuracy in the vehicle model.	n/a --> VI-Car Real Time, accurate vehicle dynamic model
Sensors & Cameras	Added and edited by means of a json file.	Added and edited from WorldSim Studio. Only Radar, Virtual Camera and GPS require no additional skills.
Data Logging/Real-Time analysis	By means of Python API (poor) data logging is available.	Data logging and Real-Time analysis by means of SimWorkbench, csv files, ..
Integrability	Unity Editor, FMI (not working at release 2020.06)	VI-Grade world, Matlab/Simulink
Additional skills required	Advanced skills in Unity Editor for editing of Maps and Ego Vehicle, good programming skills for setting up properly the simulation.	Advanced programming skills (Software Development Kit is to be used) for extraction of data from some specific sensors and cameras (different from Radar, Virtual Camera and GPS). RoadRunner skills (not so relevant).
Additional packages required	Unity Editor (Student/Private license is enough), Apollo/Autoware (Open source).	VI-DriveSim, SimWorkbench, VI-CarRealTime, RoadRunner or other for the editing of environment.

Figure 3.22: Resuming table.

Chapter 4

ADAS

4.1 The need of a Driving Simulator

As already mentioned, driving simulators are used not only during the design phase of some vehicle's components, but even for the development of Advanced Driver-Assistance Systems (also known as *ADAS*).

The reason behind this is simple and is due to the architecture of these systems. They are based on some logics and algorithms which require time and, more importantly, many attempts before actually being ready to be used; of course, this is due to the importance these systems have and to the fact that an improper design could lead to very dangerous events.

Let us consider the Adaptive Cruise Control for example. This logic is able to choose and apply a certain speed for the user vehicle, depending on the speed of the leading car (and many other parameters which will be analyzed later on); you may imagine what a non perfect design could lead to (rear-end collision and other kinds of accidents). This leads to the need of performing a lot of tests, with different speeds and, possibly, different traffic scenarios. This would be impossible without the possibility of reproducing these experiments on a virtual simulator, both in terms of cost and level of risk of these maneuver.

4.2 Why ADAS

In order to understand the reasons behind the presence of ADASs in our cars, we should have a look at some statistics.

An analysis of the National Motor Vehicle Crash Causation Survey, conducted by the National Highway Traffic Safety Administration (*NHTSA*, part of the United States Department of Transportation), shows that road accidents and crashes are due to driver's errors in **94% of the times**. Even if we need to remember that many factors are involved when drivers' errors occur, a relevant role is played by distraction which contributes in the failure of recognizing hazards while driving.

The role of ADAS is to help humans in recognizing these hazards and, sometimes, even in reacting to them by replacing the driver itself; as we will analyze, there are lots of these systems and depending on their technology (so the level of help they can deliver).

Even if sometimes we do not recognize them, nowadays cars are sold with more and more ADASs, thus increasing the level of safety. Having said this, we should always keep in mind that these systems have been developed and are used only in order to help the driver, not to fully replace him/her.

4.2.1 Crashes reduction

In the following, the effects of some, simple, ADASs is shown, by looking at statistics of accidents in highway. The sources are the *Insurance Institute for Highway Safety* and the *Highway Loss Data Institute* (USA, 2020).

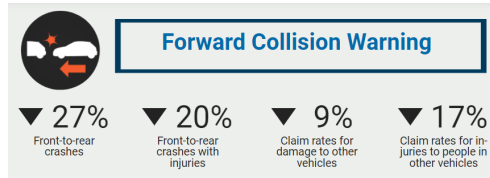


Figure 4.1: Forward collision warning statistics

As you can see from figure 4.1, the number of crashes has been decreased by 27% with a **Forward Collision Warning**, which basically warns the driver when there is an imminent forward collision. If you in addition consider the **auto-braking**, then the crashes are way less, with a drop of 50%:

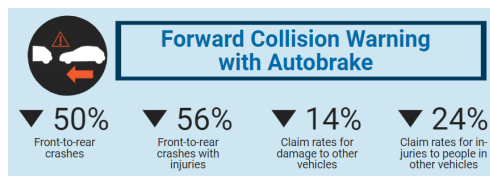


Figure 4.2: Forward collision warning with automatic braking statistics

Another system which affects a lot the percentage of crashes is the **Rear Automatic Braking**, with a reduction of more than the 75%:

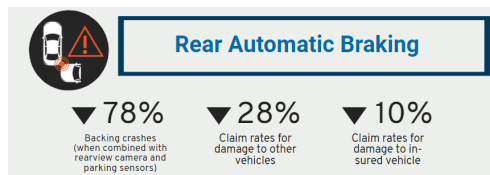


Figure 4.3: Rear automatic braking statistics

The positive results of having these kinds of systems on passenger cars, lead to the increase in the presence of ADAS on new vehicles. In the following we can see the trend between 2013 and 2018:

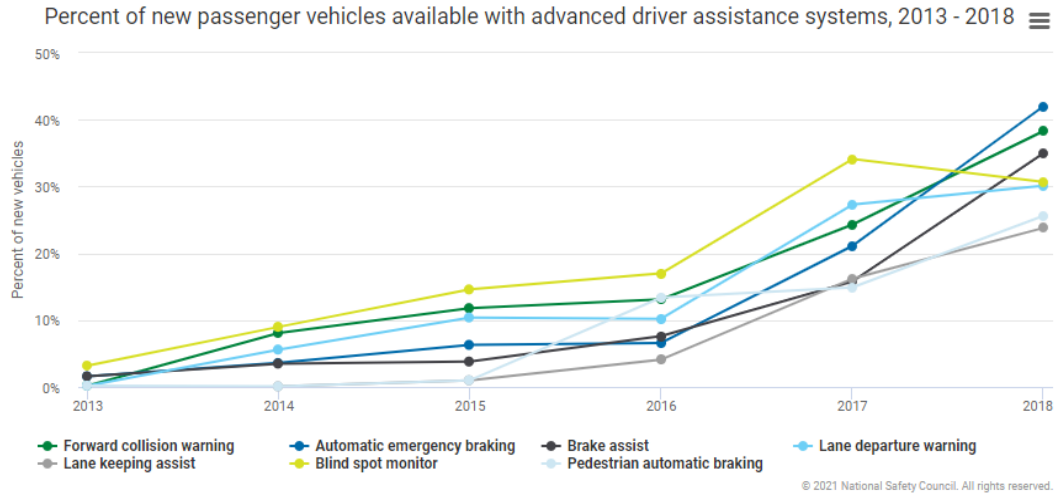


Figure 4.4: Percent of new passenger vehicles available with ADASs

4.3 Classification

As already mentioned, the amount and type of automated driving tasks of a passenger cars will define the *Level* of the vehicle; this means each car is assigned to a level depending on how much (and if) the driver has to control its demands which all together make the *dynamic driving task*.

The Society of Automotive Engineers (SAE) defines 6 levels of automation, from 0 to 5. Let us see the difference between them:

- **Level 0 (No driving automation):** most of the vehicles in the roads nowadays belong to this level. Here the dynamic driving task is provided entirely by the human driver, even if some systems to help the driver may be present: the emergency braking system, for example, can be present on a car, but, since it technically does not drive the car, it does not affect the level of driving automation actually, therefore the level of the vehicle keeps at zero.
- **Level 1 (Driver assistance):** if your car belongs to this class, then it means it has one system helping the human to drive. For example, cars with Adaptive Cruise Control only are Level 1 vehicles, since the driver still needs to act on the steering wheel and on the brake pedal.
- **Level 2 (Partial driving automation):** this is actually the starting class for cars equipped with the so called ADASs installed on. The vehicle performs both accelerating and steering by itself (if extra-ordinary situation are avoided), even if the human driver can take control whenever she/he wants. An example Level 2 automation is Tesla Autopilot.
- **Level 3 (Conditional driving automation):** the difference with the Level 2 can be perceived just from a technological point of view. Now the car is equipped with sensors and logics which can make decisions depending on the environment. One example scenario could be the one in which a vehicle in front of us is proceeding at a very low speed and, if there are the necessary conditions, we would like to overtake it: in Level 3 vehicles, the conditions of traffic (and in general of the environment) are detected and, still after the human has accepted the maneuver, the overtake is performed automatically.

- **Level 4 (High driving automation):** at this level, vehicles can even intervene if something goes wrong or if there is a system's failure. This is the first level in which vehicles can actually be set in *self-driving mode* since they do not require human interaction in most cases (but still, the driver can manually override the ADAS).

Unfortunately up to now legislations and traffic infrastructures are not ready yet, thus Level 4 vehicles can be used only in limited (urban) areas and at low speeds only (50/60km/h).

- **Level 5 (Full driving automation):** vehicles belonging to the highest level of driving automation are equipped with everything which is required to completely avoid human interaction with the car. This means the dynamic driving task is eliminated and no steering wheel or pedals will be present.

Up to now they are still under development and many researches are going on about this subject, but hopefully in the future completely autonomous vehicles will be seen in the streets.

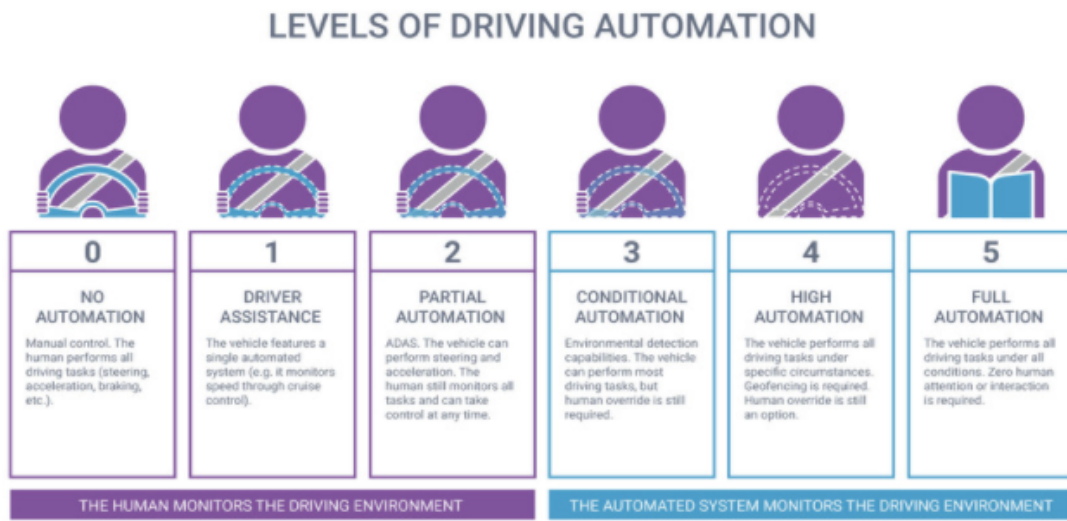


Figure 4.5: Classification of driving automation

4.4 Euro NCAP

The **European New Car assessment Program** has been founded in 1997 and is supported by the European Union.

Since then, it is responsible of the validation of new vehicles in terms of *passive safety*: this means a series of crash tests is performed on the vehicle, in order to see its reaction and how well it behaves in those critical circumstances. After the tests, a long period of evaluation and analysis of results begins, ending with the rating of the car.

4.4.1 NCAP tests before ADAS introduction

The rating system is based on different categories:

- **Adult Occupant Protection (AOP)**: frontal impact, lateral impact and whiplash tests are performed in order to evaluate the protection for adults in the car.
- **Child Occupant Protection (COP)**: in this class, the protection for children in the car is evaluated. Some tests are performed, in which the child restraint system's presence is checked and if it works properly; then the availability of provisions for safe transport of children in the car.
- **Vulnerable Road User (VRU) Protection**: safety of pedestrians (but in general other agents on the road) is evaluated here, in case a collision occurs. Head impact, upper leg impact and lower leg impact are tested.

4.4.2 NCAP tests after ADAS introduction

Since 2010, after the introduction of the Autonomous Emergency Braking system in the Volvo XC60 and the very positive results from a safety point of view which it led to, Euro NCAP decided to add the *active safety* tests, for validation of the vehicle in terms of ADASs and other features. This led to a new class in the rating system (and as a consequence new tests):

- **Autonomous emergency Braking Car-to-Car (AEB C2C)**: different situations are analyzed.
 - *Car-to-Car-Rear Stationary (CCRs)*: a vehicle collides with another one which is standing still (the frontal structure of one car hits the rear of the other).
 - *Car-to-Car-Rear Moving (CCRm)*: a vehicle collides with another one which is moving at a constant lower speed (the frontal structure of one car hits the rear of the other).
 - *Car-to-Car-Rear Braking (CCRb)*: a vehicle collides with another one which is braking (the frontal structure of one car hits the rear of the other).
 - *Car-to-Car-Front Turn-across-path (CCFtap)*: a vehicle turns across the path of an oncoming vehicle travelling at constant speed (the frontal structure of one car hits the front of the other).
- **Autonomous Emergency Braking Vulnerable Road Users (AEB VRU)**: a long list of tests is performed, in which the agent (pedestrian, cyclist etc), its state (still, moving, speed etc) and position changes.
- **Lane Support Systems (LSS)**: even here many scenarios are tested, changing the position of the ego car, the markings, the situation (overtaking, normal drive), the speeds of the cars and other parameters.
- **Speed Assist (SA)**: not only the Adaptive Cruise Control, but even warning signals and sounds if the speed overcomes the maximum one imposed by road's limitation and other systems linked to the speed regulation.

4.5 State of Art: Stanley method

In the following chapter, a control model coupling Lane Keeping Assist and Adaptive Cruise Control will be investigated. It is therefore important to set some ground rules which will help us in the description of the model. In particular, the Stanley method is investigated here. It is a *geometric path tracking controller* which is based on a kinematic model of the vehicle.

This method has been implemented by the Stanford Racing Team on a Volkswagen Touareg (named “Stanley”) which took part in the DARPA Grand Challenge 2005, a 132 miles autonomous off-road race. The general idea behind the development of this new system is that a change in the reference position could lead to different and possibly more desirable conditions for the control; in particular, now the reference point assessing for the vehicle is the center of the front axle, while other techniques used the center of gravity or the center of the rear axle.



Figure 4.6: “Stanley”

The working principle of the Stanley method is to compute a reference steering wheel angle on the basis of three main aspects:

- The **cross track error** which represents the distance from the vehicle to the nearest segment of lane [m].
- The **heading error** which is the inclination of the vehicle with respect to the direction of the road [rad].
- Certain values of **maximum steering wheel angle** and **minimum steering wheel angle**.

The desired result is to be able to compute a reference error, to be adjusted and followed by the control, which does not depend on the actual speed of the vehicle: more in specific, the speed will affect the time which is taken to adjust the error, but it does not affect the error itself.

In the following figure, the Stanley scheme is shown:

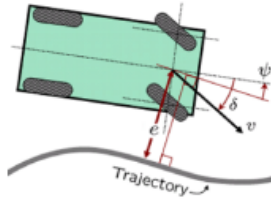


Figure 4.7: Stanley scheme

Let us see how this control was thought and how the three mentioned requirements are taken into account:

- First of all, the steering ($\delta(t)$) is adjusted in order to cover the whole heading error ($\psi(t)$):

$$\delta(t) = \psi(t) \quad (4.1)$$

- Now another contribution of steering wheel angle is obtained in order to account for the cross track error ($e(t)$):

$$\delta(t) = \arctan \frac{ke(t)}{v(t)} \quad (4.2)$$

where the steer is proportional to the error (k is a coefficient which has been experimentally evaluated), inversely proportional to the speed (as previously mentioned) and the arctan is used to limit the effect of large errors.

- The steering angle is limited:

$$\delta(t) \in [\delta_{min}, \delta_{max}] \quad (4.3)$$

As a result, the overall steering wheel angle will be defined as:

$$\delta(t) = \psi(t) + \arctan \frac{ke(t)}{v(t)} \quad (4.4)$$

Let us see some examples. If, as depicted in figure 4.8, the heading error of the vehicle is large and higher than the maximum allowed value, then the control will require the maximum steering wheel angle, until it decreases and the vehicle reaches the correct direction.

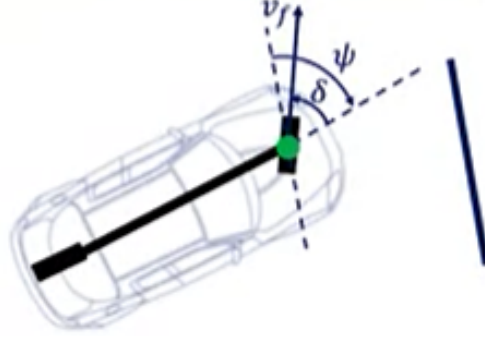


Figure 4.8: Large heading error case

For large positive track error instead, the value of $\arctan \frac{ke(t)}{v(t)}$ is near $\pi/2$ leading to have:

$$\delta(t) \approx \psi(t) + \pi/2 \quad (4.5)$$

and the vehicle will position itself perpendicular to the lane at first, decreasing the angle as the track error decreases.



Figure 4.9: Large positive track error case

Being the derivative of the cross track error in a kinematic vehicle model:

$$\dot{e}(t) = v(t) \sin(\psi(t) - \delta(t)) \quad (4.6)$$

If we substitute the value of 4.4 we obtain:

$$\dot{e}(t) = -v(t) \sin(\arctan(\frac{ke(t)}{v(t)})) \approx \frac{-ke(t)}{\sqrt{1 + (\frac{ke(t)}{v})^2}} \quad (4.7)$$

And finally for small track errors, we have that:

$$\dot{e}(t) \approx -ke(t) \quad (4.8)$$

This is relevant, since it has been proved that the decay rate ($\dot{e}(t)$) exponentially decreases to zero as the error decreases to zero and there is no dependency on speed, thus leading to the fact that faster vehicles will take more time to reach correct the trajectory, but the velocity does not affect the functioning of the control.

Chapter 5

Control model

The object of this chapter is to investigate and describe a couple of control logics, which are very common when dealing with ADASs: the *Lane Keeping Assist (LKA)* and the *Adaptive Cruise Control (ACC)*.

Let us remind you that these logics have been developed and integrated inside VI-WorldSim environment, exploiting its sensors and cameras. Having the chance of exploiting a license, the choice between the two analyzed software packages has been quite easy because of all the advantages that WorldSim has.

5.1 General functioning

5.1.1 Lane Keeping Assist

By means of a particular sensor, the vehicle is able to detect the road markings. Then the logic computes the curvature of the road and, by consequence, the direction the vehicle has to take; by knowing this (which basically means knowing the steering wheel angle the vehicle would like to have to remain inside the road) and the steering wheel angle at that precise instant, this logic computes the error, which actually is the amount of steering angle to be applied.



Figure 5.1: Lane Keeping Assist

Of course there are a lot of parameters affecting the maneuver, but still this is the basic functioning. As you may have noticed, the procedure to know and compute the correction to the steering wheel is actually the same a human driver usually does (depending on the road's curvature he/she automatically adjust the steering wheel angle to keep the car inside the lane).

The main sensor which is required for this control is the **Virtual Camera**, which is able to detect road markings and estimate in this way the curvature of the road (therefore the steering wheel angle which is needed).

5.1.2 Adaptive Cruise Control

When we are driving, the throttle and brake commands are defined depending on several parameters (even if we take some of them for granted sometimes), but the most relevant ones are the actual speed we are going at, the maximum allowed speed (according to traffic laws) and, if it is present, the speed of the lead vehicle.

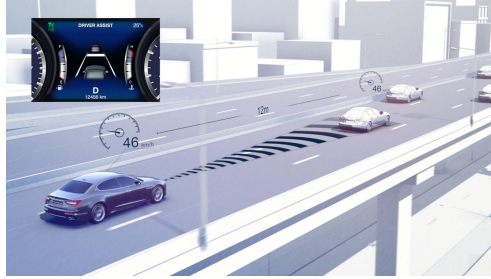


Figure 5.2: Adaptive Cruise Control

These three parameters are the foundation of the ACC, which basically reproduces the behavior of a human driver choosing the proper percentage of throttle and brake commands. Therefore the control is based on the information coming from the vehicle itself and the data coming from the leading one: this is obtained by means of a **Radar**.

5.2 Control model inside WorldSim

5.2.1 LKA

As already mentioned, the LKA is supported by a sensor which must be able to detect the road markings: in case of WorldSim, this sensor is defined as *Virtual Camera*. It is able to perform three actions (which can be checked or not, in case something is not needed): lane markings detection, obstacle detection and traffic signs.

The first thing we need to do is adding this sensor at the Ego vehicle while we are editing the simulation's scenario (the procedure has already been described in the dedicated chapter).

Once we have added the Virtual Camera and checked the *Lane markings* option, we are able to extract the following data from the simulation (in real time and afterwards):

- **Left Lane:**
 - **Distance:** the distance, in meters, from the virtual camera to the nearest left lane recognized.
 - **Status:** it is a boolean variable. 1 if the sensor is retrieving data, 0 if it is not.
 - **Polynomial:**
 - * $X_{min} - X_{max}$: this is the interval of longitudinal distance, in meters, from the sensor in which the polynomial is going to be computed.
 - * $c0, c1, c2, c3$: coefficients of the third order polynomial which is reproducing the road marking on the left (the so called **lane fitting** procedure); $c0$ has the same value of the data "Distance".

- **Right Lane:**

- **Distance:** the distance, in meters, from the virtual camera to the nearest right lane recognized.
- **Status:** it is a boolean variable. 1 if the sensor is retrieving data, 0 if it is not.
- **Polynomial:**
 - * $Xmin - Xmax$: this is the interval of longitudinal distance, in meters, from the sensor in which the polynomial is going to be computed.
 - * $c0, c1, c2, c3$: coefficients of the third order polynomial which is reproducing the road marking on the right; as before, $c0$ has the same value of the data “Distance”.

Curvature computation

Now that we have all the information we need, we can proceed in the computation of the curvature.

Since the control has been developed as a Simulink Model, the best way to perform this computation is the “MATLAB Function” block. As you can see in the following picture, the block is fed with the parameters of the right lane coming from the Virtual Camera and it outputs the curvature:

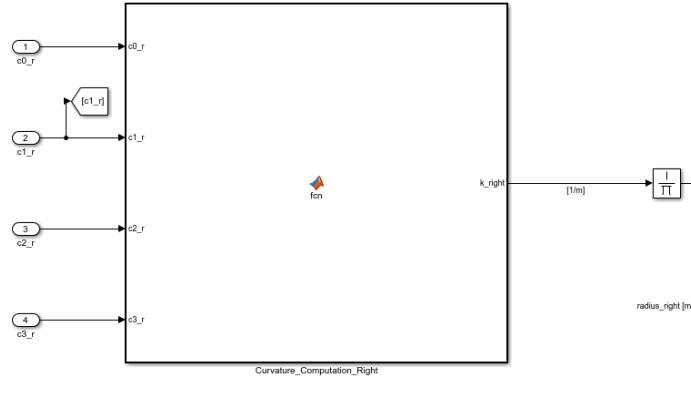


Figure 5.3: Right lane curvature computation block

The function which has been used is called *polynomialCurveCurvature* and it is able to compute the curvature of a parametrized polynomial curve. In our case, the polynomial is:

$$y = c_0 + c_1x + c_2x^2 + c_3x^3 \quad (5.1)$$

and has been parametrized as:

$$\begin{cases} x = t \\ y = c_0 + c_1t + c_2t^2 + c_3t^3 \end{cases} \quad (5.2)$$

This is mandatory since the mentioned matlab function accepts 3 arguments: the point in which the curvature has to be computed, the vector containing the coefficients of x's equation and the vector with the coefficients of y's equation (*polynomialCurveCurvature(point, xcoeff, ycoeff)*). In this case we had:

$$\begin{cases} point = c_0 \\ xcoeff = [0 \ 1 \ 0 \ 0] \\ ycoeff = [c_0 \ c_1 \ c_2 \ c_3] \end{cases} \quad (5.3)$$

This has to be done both for the right lane and the left lane; as outputs we will get k_right and k_left which are respectively the curvature of the lane to the right and left. What is needed though is the mean curvature and, in order to get it, we need to pass through the curvature's radii computation:

$$\begin{cases} radius_right = \frac{1}{k_right} \\ radius_left = \frac{1}{k_left} \end{cases} \quad (5.4)$$

Once we have the two radii, we first compute the mean radius:

$$mean_radius = \frac{radius_right + radius_left}{2} \quad (5.5)$$

And then if we perform its reciprocal, we get the mean curvature:

$$mean_k = \frac{1}{mean_radius} \quad (5.6)$$

Of course, this is performed at each simulation's step in real time, thus leading to have at each instant the mean curvature of the road.

Road's heading's computation

The virtual camera for lane markings is used for the computation of the road's heading too, which is basically the inclination of the road with respect to the longitudinal axis of the vehicle. The need of this value comes from the lateral control of the vehicle, which will be analyzed later on.

If we approximate the polynomial, cutting it down to first order, we get the following straight line:

$$y = c_0 + c_1 x \quad (5.7)$$

In order to get the angle, we just need to take the slope of the line (c_1 in this case) and then perform the arcotangent of that value:

$$heading_road = \arctan c_1 \quad (5.8)$$

5.2.2 ACC

The WorldSim environment allows the user to add a Radar on the ego vehicle. This sensor is able to detect up to 64 targets and, for each of them, retrieve informations:

- **id**: it is the unique identifier of the target.
- **yaw**: it defines the angle in z direction in the sensor reference frame in [radians] (let us recall that the reference frame is a right-handed coordinate system in which the x is along the longitudinal axis of the vehicle, the y is perpendicular to the x and point to the left and the z points up) .
- **range**: the distance from the ego vehicle in [m] .
- **power**: it is the power of return of the ray in [dB]
- **range rate**: first time derivative of the range, so the velocity of the target with respect to the ego vehicle in [m/s].
- **yaw rate**: first time derivative of the yaw of the target with respect to the ego vehicle in [rad/s].
- **status**: it is a boolean assessing for a proper functioning of the radar for that target (so basically it states if the target has a valid return).

5.3 Lateral Control

Let us focus for a moment on what a human driver (automatically) does on the road: as the vehicle proceeds, the driver realizes what steering wheel angle is needed depending on the road curvature and then applies it. As you should see, this procedure is made up by two steps:

1. Computation of a *required steering wheel angle* based on some reference.
2. Application of a specific *torque at the steering wheel* in order to obtain that required steering wheel angle.

The aim of the **lateral control** is to reproduce these two steps. In the following, a schematic representation is proposed:

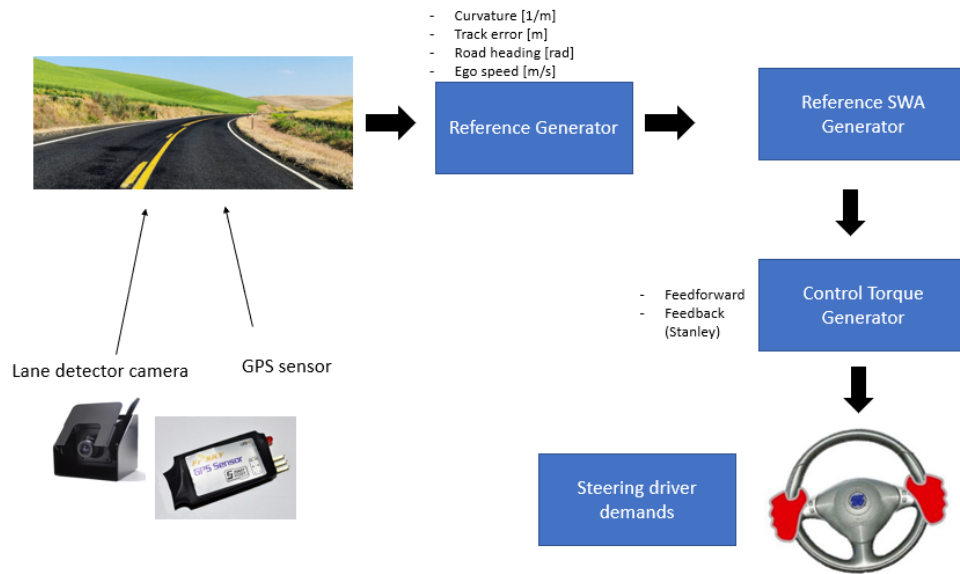


Figure 5.4: Schematic representation of Lateral Control functioning

5.3.1 Reference generation

In case of the lateral control, the reference is used to compute what is the required steering wheel angle and is based on information of the road coming from the sensors (in this case the Virtual Camera). It is made up by the following terms:

- The road *curvature* $[1/m]$.
- The distance from the road centerline $[m]$ (defined as e_{\perp} or track error).
- The *heading of the road* $[rad]$.
- The *first derivative of the heading of the road* $[rad/s]$.
- The *ego speed* $[m/s]$.

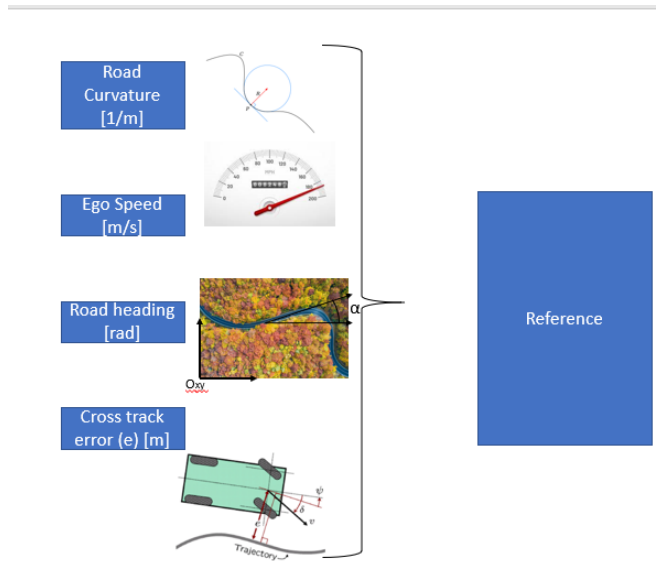


Figure 5.5: Reference information.

Cross track error computation

The two coefficients R_c0 and L_c0 involved in the computation of the third grade polynomial which reproduces the road markings (right and left lane) are also the distance (with sign), respectively, of the sensor (therefore of the ego vehicle) from the nearest right lane and left lane.

Recalling that the values are to be taken in absolute value it is then possible to compute the lane width and then the distance of the vehicle from an imaginary centerline of the lane, which has been chosen as the ideal trajectory to be followed.

As previously introduced, the values of L_c0 e R_c0 defines the distance of the sensor (therefore of the ego vehicle) respectively from the nearest left and right line. This distance is computed taking into account the reference frame, reported below:

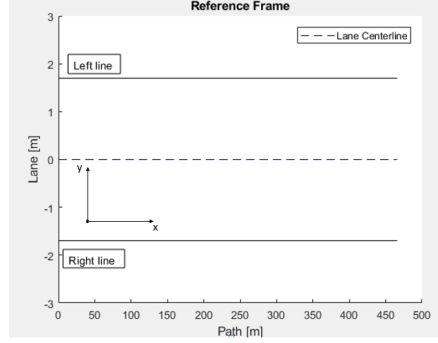


Figure 5.6: Reference information.

The lane width is therefore computed by taking into account the absolute values. Then the computation proceeds as:

- If the ego vehicle is on the right of the centerline (figure 5.7)

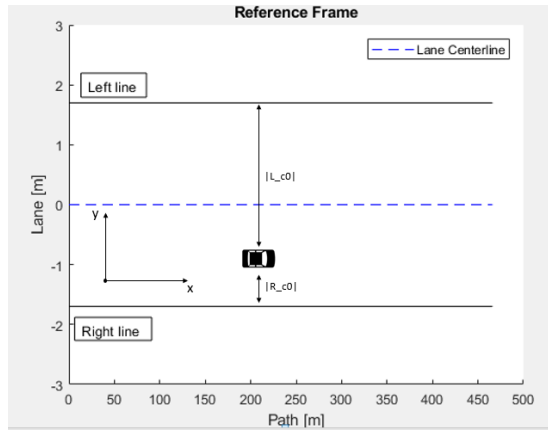


Figure 5.7: Ego vehicle on the right of the desired trajectory.

meaning:

$$|L_c0| > |R_c0| \quad (5.9)$$

Then the cross track error is:

$$e_{\perp} = |L_c0| - \frac{lane_width}{2} \quad (5.10)$$

- While if the ego vehicle is on the left of the centerline (figure 5.8)

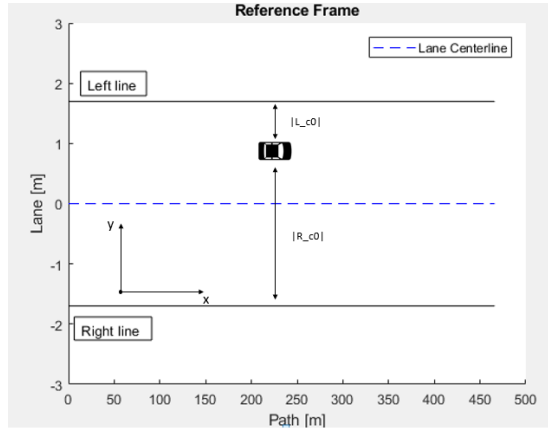


Figure 5.8: Ego vehicle on the left of the desired trajectory.

meaning:

$$|R_c0| > |L_c0| \quad (5.11)$$

Then:

$$e_{\perp} = |R_c0| - \frac{lane_width}{2} \quad (5.12)$$

Even though this e_{\perp} plays the main role, it will be slightly adjusted in order to take into account the errors which the control will make when previewing the curvature and the road heading; let us recall that even if these two terms are computed at each simulation step (with a sample time of 0.002 s), there is always an error since the curvature which is estimated at 1.002 s is different from the actual curvature at 1.004 s, even if the difference is almost irrelevant.

5.3.2 Steer angle generator

Once the reference has been generated, now the model should compute what is the steering wheel angle associated to it (defined as “swa_ref”).

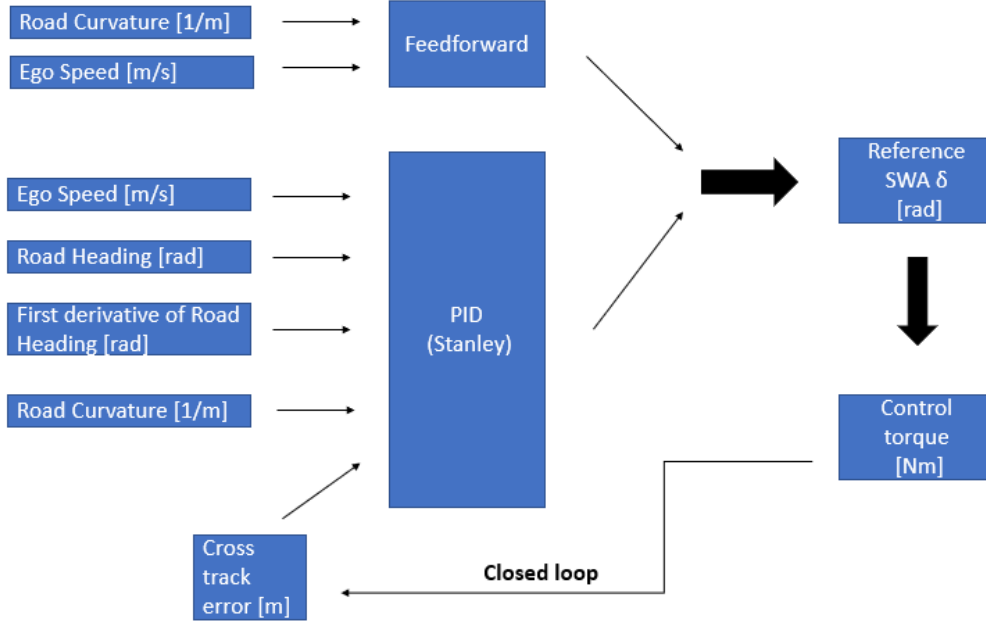


Figure 5.9: Steering wheel angle generator scheme

As you can see from figure 5.9, the control is made up by two contributions: one steering wheel angle coming from a closed loop based on a so called “Stanley method” plus a feedforward term.

Feedforward

The effect of the feedforward is known: it relies on information coming from the vehicle and it aims at guessing the steering wheel angle on the basis of those information. The mathematical representation of this is shown in the following:

$$swa_{FF} = \frac{K_{steer}}{R} [(l_f + l_r) + K_{ug} V^2] \quad (5.13)$$

where:

- K_{steer} is the steering ratio (assessing for the difference between the angle at the steering wheel and the one at the ground).
- R is the road curvature radius [m].
- l_r and l_f are the rear and front wheelbase lengths [m].
- V is the ego vehicle speed [m/s].
- K_{ug} is the understeer gradient (computed below) [rad/m/s²].

The **understeer gradient** defines the behavior of the vehicle at high speeds and values of lateral accelerations. If the gradient is positive as the speed during a curve (therefore the lateral acceleration too) increases, the vehicle requires an increasing steering wheel angle (with respect to the the case of the same curve but at low speeds); if, on the contrary, the gradient is negative, then it means that for higher values of lateral accelerations the required steering wheel angle is lower.

The understeer gradient is computed as follows (on the basis of the inverse of a bicycle model):

$$K_{ug} = m \left[\frac{l_r}{(l_r + l_f)c_f} + \frac{l_f}{(l_r + l_f)c_r} \right] \quad (5.14)$$

where:

- m is the vehicle mass [kg].
- c_r and c_f are respectively the cornering stiffnesses on the rear and front [N/rad].

Feedback: Stanley method

In order to adjust the contribution coming from the feedforward, a modified Stanley method is used inside a closed loop (checking the value of e_{\perp}).

As already discussed in chapter 4, the Stanley method was developed at Stanford and represents a solution for the path tracking of a vehicle. In other words, it is a way of computing the required steering wheel angle on the basis of information coming from the road and the vehicle itself.

In the following picture, the Stanley contribution computation is shown:

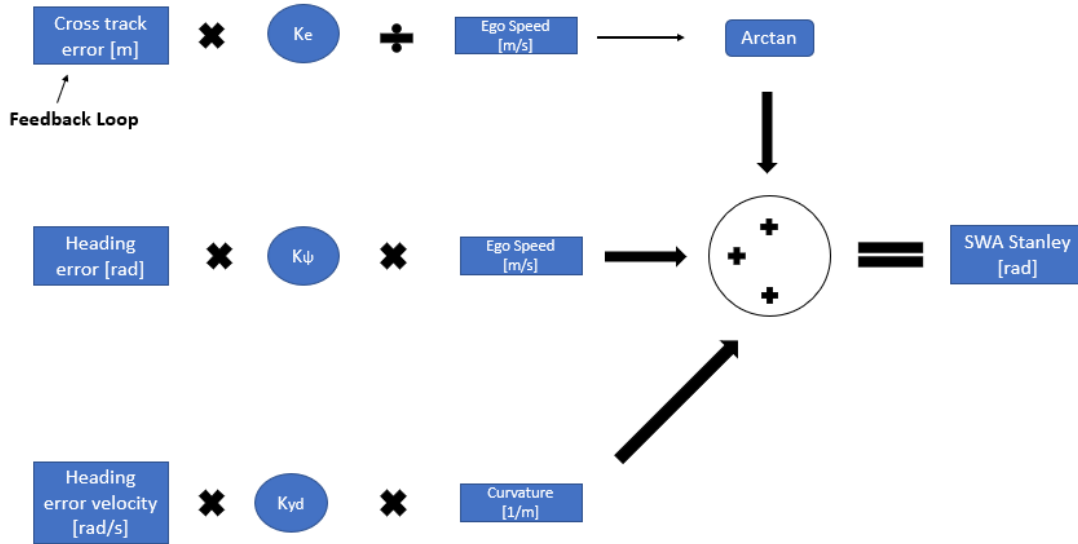


Figure 5.10: Stanley method

The mathematical representation is:

$$\begin{cases} swa_{Stanley} = \arctan \frac{K_e * e}{V} + K_{\psi} * V * \delta_{\psi} + \dot{\delta}_{\psi} * K_{yd} * \left| \frac{1}{R} \right| \\ e = V * T_{preview} * \sin \delta_{\psi} + e_{\perp} \\ \delta_{\psi} = \psi_{ref} - \psi \\ \dot{\delta}_{\psi} = \dot{\psi} - \frac{V}{R} \end{cases} \quad (5.15)$$

where:

- K_e , K_{ψ} and K_{yd} are coefficients for the tuning of the loop.
- $T_{preview}$ is the preview time [s].
- ψ is the vehicle yaw angle [rad].
- $\dot{\psi}$ is the vehicle yaw rate [rad/s].

5.3.3 Steer torque computation

Now the overall required steering wheel angle can be obtained:

$$swa_{req} = swa_{Stanley} + swa_{FF} \quad (5.16)$$

This steering wheel angle is compared with the actual steering wheel angle and their difference will be used as the error to be lowered down to 0 inside a **PID controller** (as shown in the picture 5.11).

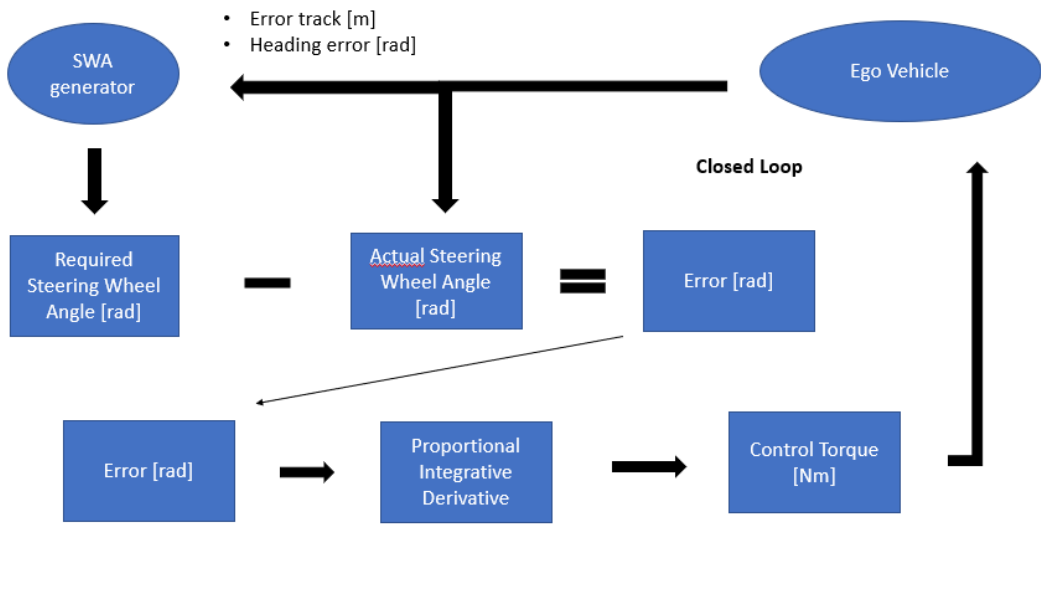


Figure 5.11: Control torque computation

The output of the block is therefore the torque to be applied automatically at the steering wheel, in order to obtain the desired trajectory of the ego vehicle.

5.4 Longitudinal control

As for the lateral control, the longitudinal one is based on the computation of a reference (speed) which has to be followed and obtained by means of a control. This reference speed here is computed by means of an approach based on an **Intelligent Driver Model (IDM)** which, considering information coming from the Radar and data of the ego vehicle itself, computed a target velocity.

5.4.1 Reference speed generator: IDM

The mathematical expression of the IDM which is used to compute the target acceleration (which will be integrated to get the target speed) is the following:

$$\begin{cases} \dot{v}_{ref} = a * \left[1 - \frac{v}{v_0} \delta - \frac{\tilde{s}(v, \Delta v)^2}{s} \right] \\ \tilde{s}(v, \Delta v) = s_0 + v * T_h + \frac{v * \Delta v}{2 * \sqrt{a * b}} \end{cases} \quad (5.17)$$

where:

- a = maximum comfort acceleration $[m/s^2]$.
- b = maximum comfort deceleration $[m/s^2]$.
- v = actual speed of the ego vehicle $[m/s]$.

- Δv = relative speed between the ego vehicle and the vehicle in front of it [m/s].
- s_0 = minimum distance between the ego vehicle and the vehicle in front of it [m].
- s = distance between the ego vehicle and the vehicle in front of it [m].
- T_h = minimum time to collision [s].
- δ = attenuation of acceleration when speed increases.
- v_0 = ego vehicle desired speed [m/s]. It is the speed the ego vehicle would like to have taking into account the speed limits only.

As a result of this computation, the reference acceleration is obtained. By simply integrating the value in time, we get the reference speed.

Desired ego speed

Let us for a moment focus on what is the ego vehicle desired speed and how it is computed.

At first a *raw desired ego speed* is computed; in this computation, the curvature is taken into account (the higher the curvature, the lower should be the speed to keep stability of the vehicle), in addition to values of maximum comfort lateral accelerations.

Once the raw desired speed is computed, then it has to be saturated. If the desired speed is higher than limitations due to possible lane keeping stability problems, then the vehicle should not accelerate up to that value. In picture 5.12 the look-up table is involved in this limitation: if the speed exceeds the limits (10 m/s and $\sqrt{AyLKASLimLeft/Right * R}$), then the ego desired speed is saturated.

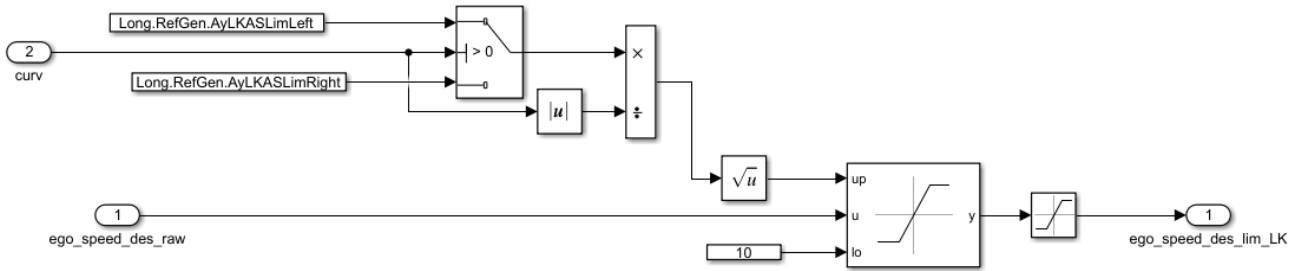


Figure 5.12: Speed desired

where $AyLKASLimLeft$ and $AyLKASLimRight$ are respectively the maximum values of left and right lateral accelerations according to limits for a stable lane keeping assis system.

5.4.2 Throttle and Brake computation

Once the reference speed and acceleration have been computed by means of the IDM, the model needs to determine the percentage of throttle or brake which is needed to obtain the correct value.

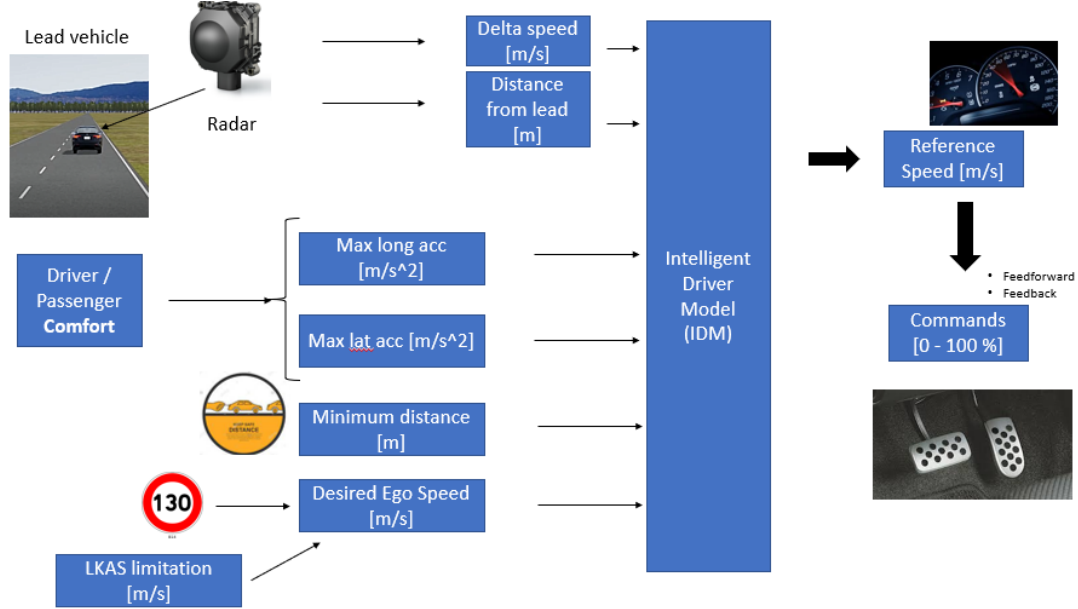


Figure 5.13: Longitudinal control

The simulink block in figure 5.13 is in charge of this operation: using as inputs the actual speed, the reference speed and the reference acceleration, it determines the commands (at first in the range $[0, 1]$, they are converted in percentage with the two gains).

As the other controls that are present in the model, even here it is made up by two contributions: one coming from the feedforward and the other from a feedback.

Control torque computation

As already occurred for other controls, the parameter to be evaluated is made up of two contributions: one coming from a feedforward part in which the model tries to predict the required commands on the basis of the reference acceleration obtained by means of the IDM and the other belongs to a feedback part (closed loop).

Starting from the *feedforward contribution*, it is computed as follows:

$$command_{FF} = K_{acc} \dot{v}_{ref} \quad (5.18)$$

The *feedback loop* is instead made up by a PID with the addition of an anti-windup to limit overshoots due to the integral part.

A schematic representation of the control is shown below:

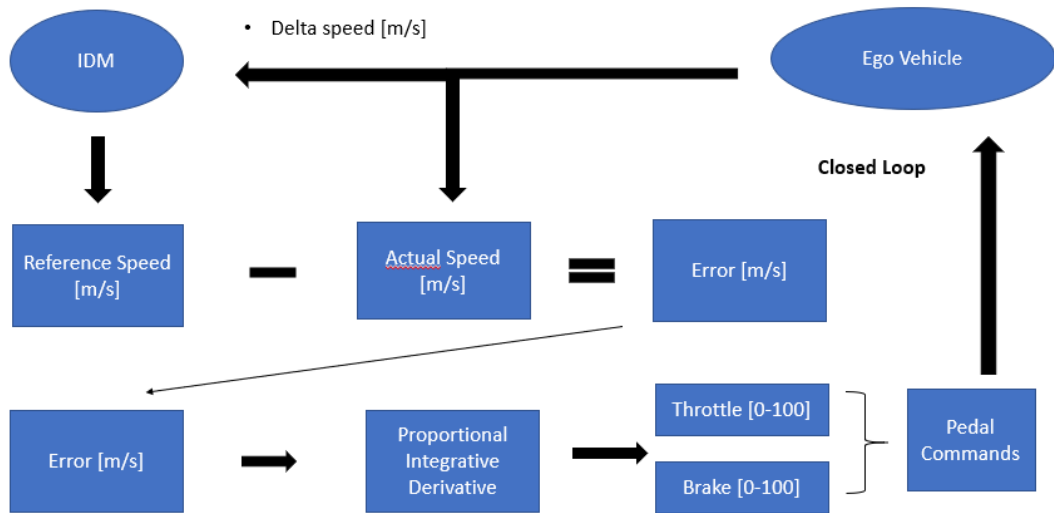


Figure 5.14: Feedback contribution

As a result, the commands for braking and throttling are obtained, as a function of the speed of the leading vehicle.

5.5 Vehicle model

The vehicle plays for sure a very important, if not the main, role when we talk about driving simulations. Even if it has not been crucial in my thesis and within the control model, in the following it is briefly described the model and which are the parameters that can (potentially) be edited.

5.5.1 VI Car Real Time

AS previously mentioned, VI-Grade offers the possibility of having all the different aspects of the simulation already integrated and implemented all together. The editing of the vehicle which will be used in a simulation (let us say inside a WorldSim scenario) is performed on VI Car Real Time.

VICRT contains a simplified model of a four-wheeled vehicle with 14 degrees of freedom (reduced degree-of-freedom model). The model includes 5 rigid parts:

- The **vehicle chassis**, sprung mass of the model.
- Four **wheels**, unsprung masses of the model.

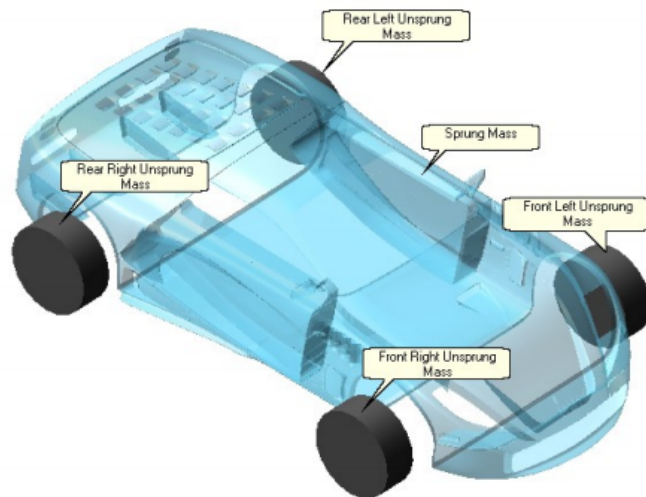


Figure 5.15: Rigid parts of CRT vehicle model

The vehicle chassis has 6 degrees of freedom (displacements and rotations in three directions) and the each wheel has 2 degrees of freedom assessing for the motion with respect to the vehicle body and the wheel spin.

The user is able to edit the following subsystems of the vehicle:

- Front suspension
- Rear suspension
- Steering
- Body
- Powertrain
- Front wheels and tires
- Rear wheels and tires
- Brakes
- Auxiliary systems

As the reader may realize, the level of accuracy in the editing of the vehicle (from a dynamic point of view) is quite advanced and even quite simple and straightforward (the use of the software is not investigated here).

5.5.2 Vehicle model for simulation

In my specific case, the vehicle which is used for the simulation is a Maserati Ghibli, with an internal combustion engine coupled with two electric motors on the front wheels.

In the following, the main parameters of the vehicle are shown:

- Wheelbase = 2600 mm.
- Front track = 1680 mm.
- Rear track = 1650 mm.
- Center of Gravity height = 490 mm.
- Vehicle mass = 1850 kg (including driver, fuel and passenger).
- Rolling inertia = 700 kgm^2 .
- Pitch inertia = 2600 kgm^2 .
- Yaw inertia = 2900 kgm^2 .

For commercial reasons the specific parameters regarding the dynamic behavior of the vehicle (suspensions, steering system, etc..) will not be divulged.

Chapter 6

Validation of the control model: experimental testing

I have had the opportunity to use the control model and validate it inside a WorldSim scenario, whose usage requires a license, and test it on an high fidelity driving simulator which is commonly used by industries.

In this chapter, I will go through each step of the experimental testing, starting from the creation of the simulation scenario on the software, the communication setting with the simulator and finally the tuning procedure of the control parameters.

6.1 Simulation setting on a Real-Time Concurrent machine

6.1.1 Creation of the simulation scenario with WorldSim Studio

The general idea is to create a scenario in which the ego vehicle is following another car which has a velocity changing in time. In this way both the lane keeping and the adaptive cruise control can be checked and tested. An extra-urban scenario has been chosen.

The user needs to add two vehicles, the ego car (the highlighted vehicle in figure 6.1) and the leading one (the highlighted vehicle in figure 6.2):

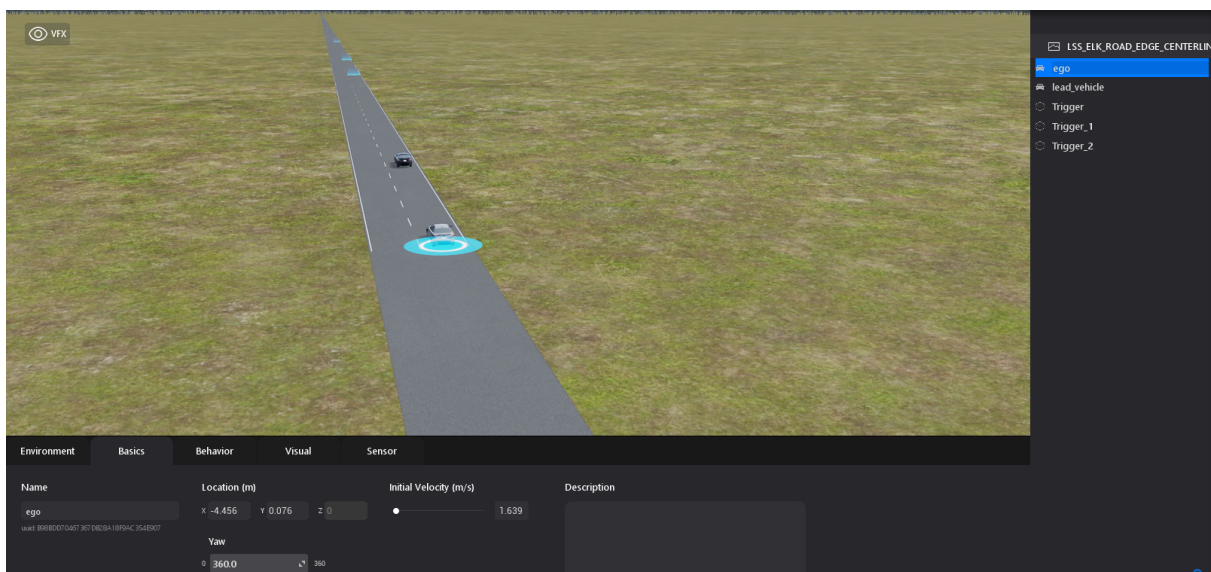


Figure 6.1: Ego vehicle configuration

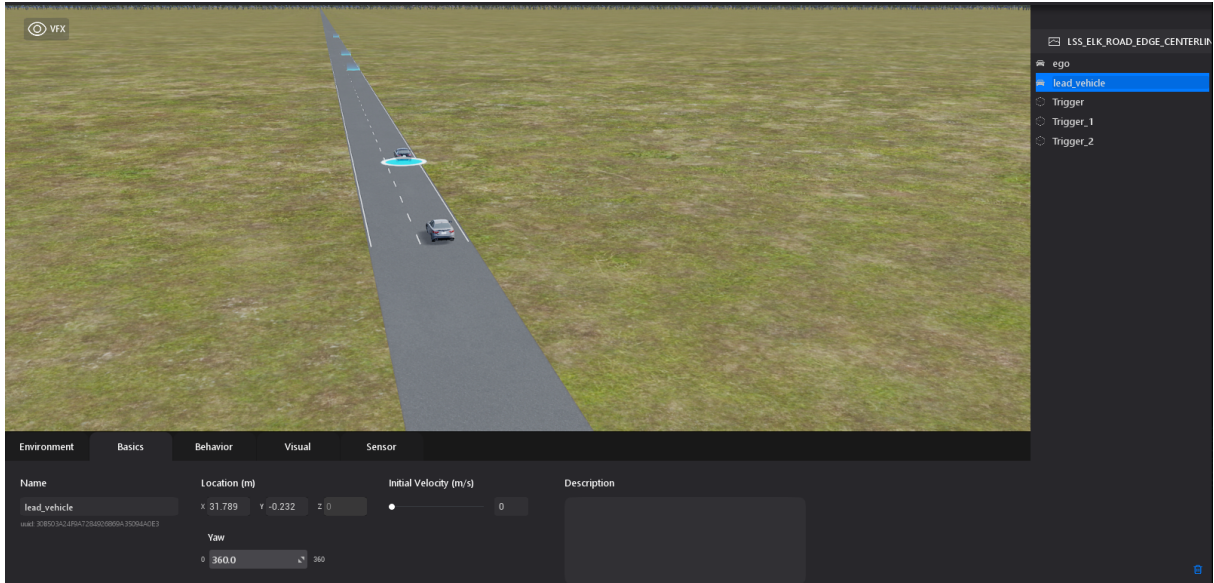
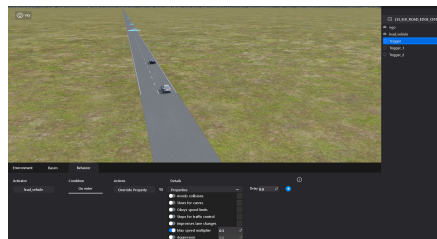


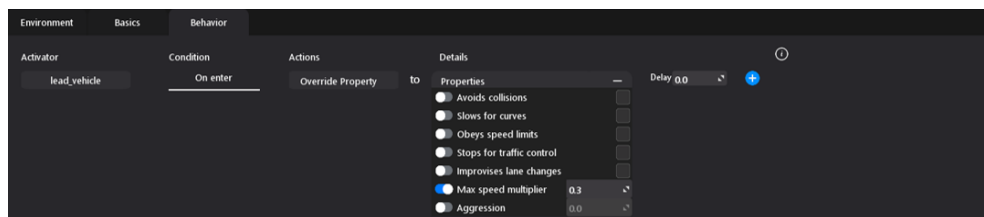
Figure 6.2: Lead vehicle configuration

As you can see, the ego vehicle has an initial velocity, while the lead one starts from zero speed. This has been done in order to see how the ego vehicle has to decelerate in the first meters because of the presence of a slower vehicle.

After that, some **triggers** have been added; their aim is to apply proper changes to the speed of the leading vehicle, in order to see if the Adaptive Cruise Control works properly (adapting the ego speed to the one of the lead vehicle). In the following picture, the first trigger is represented.



(a) The highlighted element is a trigger



(b) Behavior of the trigger

The *activator* is the leading vehicle and the *condition* to activate the trigger is "On enter", thus meaning the vehicle has to pass the line representing the trigger. The *action* the trigger involves is a deceleration of the speed of the lead vehicle, acting on the Property "Max speed multiplier" which basically limits the maximum speed of the vehicle.

Other two additional triggers have been used, to increase and decrease again the max speed multiplier. In this way a variation of the leading vehicle speed is obtained, thus being able to test the proper functioning of the ACC.

6.1.2 SimWorkbench and DriveSim

Once the scenario is created, the user needs to set properly the simulation choosing several parameters. In order to do so, the following software packages are used:

Simulation Workbench (SimWB)

It provides a framework that allows the execution of simulation models in real-time. Inputs and outputs can be connected with external hardware and it supports multiprocessors architectures (let us think at the many processes that run within a single simulation in a WorldSim test).

Within SimWB it is possible to create the simulation *test*, which is made up by different parts. In the following picture it is reported the test that has been created and used to perform my simulations.

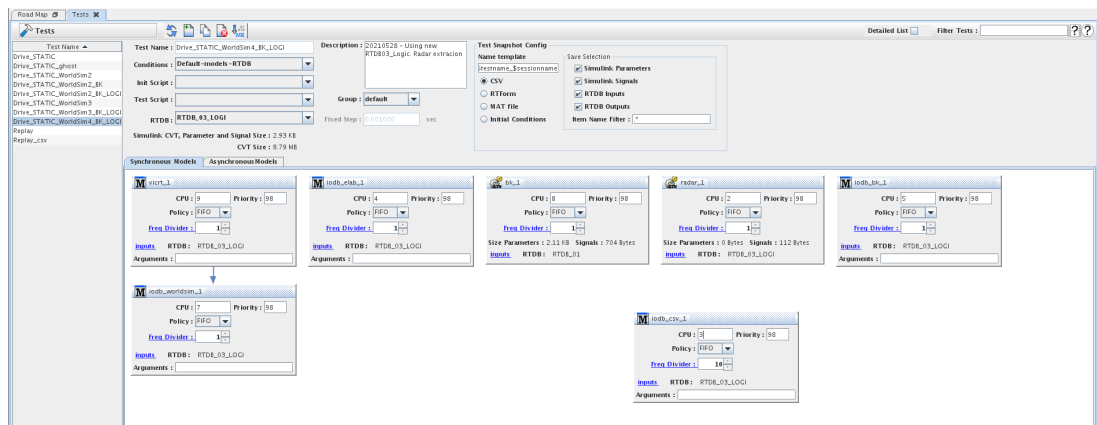


Figure 6.3: SimWB Test

The main important blocks are:

- bk: it is basically a simulink model assessing for the use of SimSound (the process in charge for simulation audio).
- vicrt: it is the VI-CarRealTime solver for Linux. It allows the user to exploit a VI-CRT vehicle model.
- iodb_csv: it is necessary to save the simulation data or to stream them to a third party telemetry software.
- iodb_worldsim: it allows the running of the simulation inside a WorldSim scenario.
- radar: an additional simulink model which has been created and compiled to solve a problem (it will be analyzed in a while).

Once the test is properly created, a session has to be added to it and then the simulation is ready to start.

DriveSim

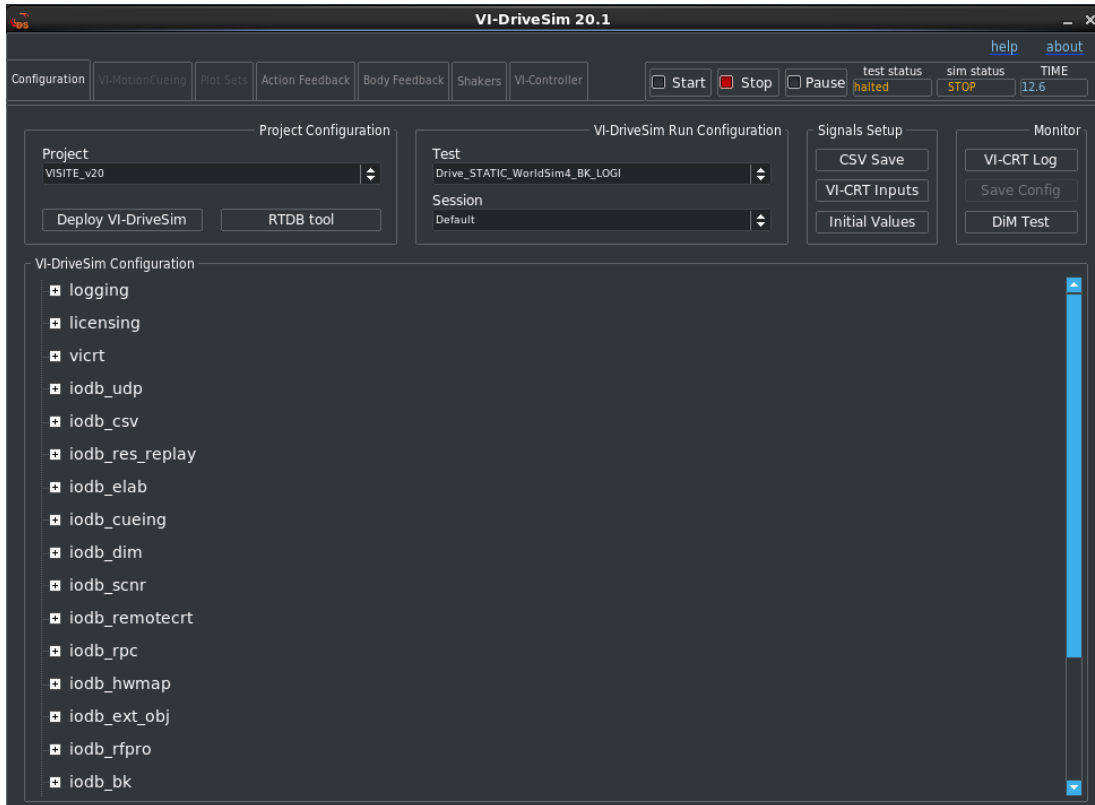


Figure 6.4: VI-DriveSim GUI

Here the user can choose the Project in which the test has been created, the test and the session. Once this is all set, some parameters have to be chosen inside the different tabs under the DriveSim Configuration section.

Before actually starting the simulation, *VI-Controller* has to be started.

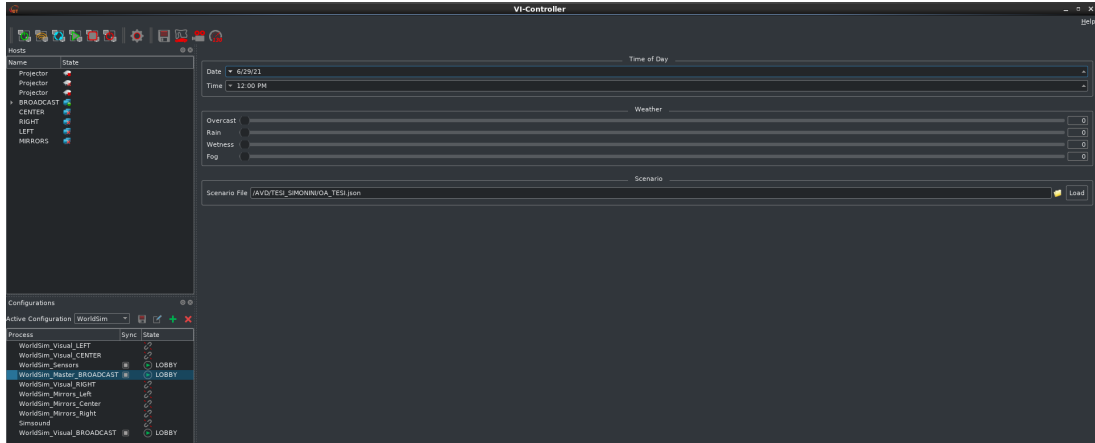


Figure 6.5: VI-Controller GUI

Here, the user can turn on all the machines which belong to the architecture of the simulator and the different projectors. As all the machines are on, the user can start the processes which are associated to them.

As a last step, the *Scenario File* has to be chosen and loaded (here the json file of the scenario which has been created at the previous step has to be taken).

It is worth to say that once the simulation is started and is running, from the controller is possible to add widgets (telemetry widget, and so on) and change environmental or weather conditions like the time of the day (thus affecting the inclination of sun rays), the presence of rain, fog and the wetness of the road. All these features contribute to increase the level of immersion of the driver within the simulation.

6.2 The static simulator



Figure 6.6: Static simulator.

As already discussed in the first chapter, when we talk about driving simulators, lots of different options are available. Each architecture differ from the others depending on the degree of fidelity, which leads to different possibilities from the engineering testing point of view and,

of course, to different prices.

The simulator I had the chance to use is defined as **static simulator**. The reason behind the name is quite intuitive, it does not have any degree of freedom, thus resulting in no motion of the body.

6.2.1 Immersion of the driver

Even if there is no motion, the driver is still very immerse in the virtual reality for many reasons.

For a start, the simulation's scenario is projected on a 7 meter fixed curved screen (defined as *dome*). This means the driver can see what happens in the virtual world, with a field of view of 180 degrees. In figure 6.7 it is possible to see the projection of a simulation scenario on the curved screen.



Figure 6.7: Example of scenario on the curved screen.

In addition to that, there are small screens which are substituting the mirrors (the left and right, and the central one), thus resulting in the possibility for the driver to see what happens even behind him, in the simulation's scenario.



Figure 6.8: Flat screen assessing for the car mirror.

Finally, thanks to the presence of hardware assessing for the reproduction of the rod's forces inside the steering system, a force feedback on the steering wheel is perceived, thus increasing again the level of fidelity of the simulator and immersion in the scenario. This hardware part plays a very important role in the so called *Hardware-In-the-Loop* (HIL) activities, in which mechanical parts (in this case parts belonging to the steering system) undergo stresses thanks to the action of software and electronic bodies.

Simulator Adaption Syndrome (SAS) Simulators and virtual reality play a relevant role in nowadays testing and therefore developing of vehicles, but at the same time they could lead to a particular syndrome for the driver: the so called *Simulator Adaption Syndrome*.

As the years pass, the technology improves and with it the delay in simulation. In particular, I am referring to the delay between the instant in which the driver commands some inputs for the simulator and the instant in which these inputs are actually received by the machine and the driver can perceive the change. This delay, even if it could seem irrelevant at first sight, it can cause some perception problem.

As the driver comes back driving in real world, its brain has adapted at the virtual scenario and it expects everything will occur with that delay, even if, of course, there is not any.

This is the problem with SAS and it is present in all simulators, not only the driving ones.

In order to reduce the effect of the syndrome, the delay should be reduced as much as possible, but still it has to comply with technology. The only possible workaround is to drive in the virtual world for small amounts of time, in order to avoid the brain's adaptation, or wait a while for driving in real environment (for example, airplanes' pilots must wait for almost a week after a session in the simulator, in order to let the brain to come back to "no delay situation").

6.2.2 General architecture

The static simulator is made up by a lot of machines, but the most important one is the **Concurrent** one. This Real-Time Operating System (RTOS) is able to sustain any real-time application, thus allowing not only the streaming of information to build and run the simulation's scenario, but also the opposite flow of data, leading to the possibility of retrieving data from the simulation (the so called *data logging*, but also the real-time data analysis).

In my case the concurrent machine is a Linux one and, without entering too much into the details, it communicates with other additional windows machines, assessing for operations like graphical projection of the scenario (on the different screens; in particular the dome, the mirrors and an additional flat screen which can be used to perform real-time data analysis), scenario's sound reproduction (which again increases the immersion of the driver), scenario's editing and so on.

All the simulation is run and edited by means of the concurrent only, inside the so called *control room* where engineers and technicians can see in real time the simulation and perform a first post-processing of data.

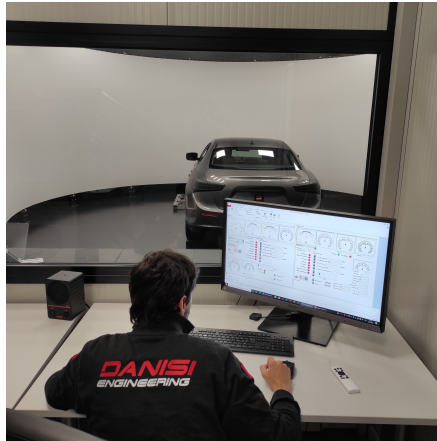


Figure 6.9: Control room.

To make it more clear and sum up the key components which make up the simulator, a schematic representation is proposed below:

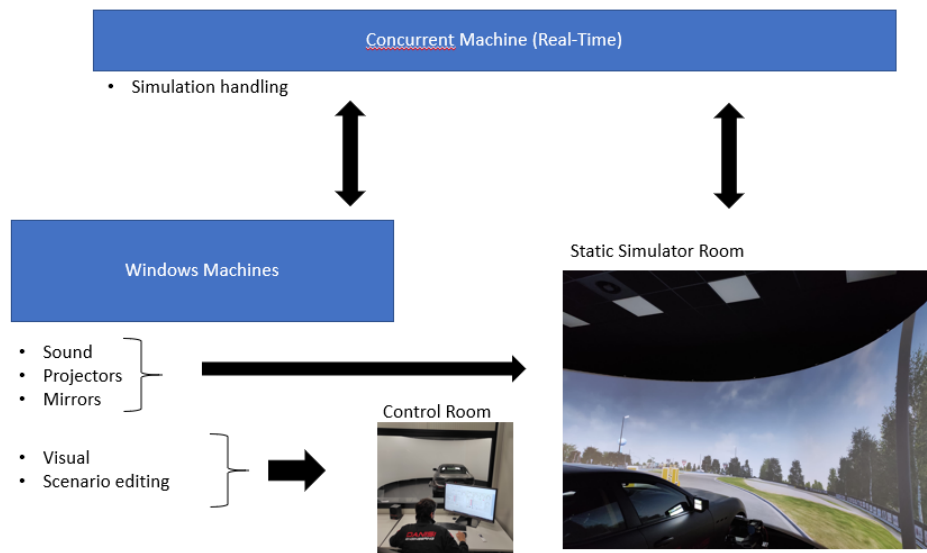


Figure 6.10: Static simulator architecture.

6.2.3 Communication SimWorkbench - MATLAB/Simulink

SimWorkbench allows the user to exploit, within the simulation (performing a **co-simulation** actually), a Simulink model. Actually there are two possible ways to do so.

1) Compiled model This solution is the preferred one usually. Let us see which are the steps to be followed in order to set this solution up:

- Create the simulink model. Very likely you will need to refer to some data outgoing the concurrent (outputs of the simulation) and send data back to it (inputs for the simulation); furthermore, this can happen with data that are already present on the RTDB or you may need to create new channels and add them at the RTDB. In order to do so, SimWorkbench offers a small Simulink Library in which you find 4 different blocks:

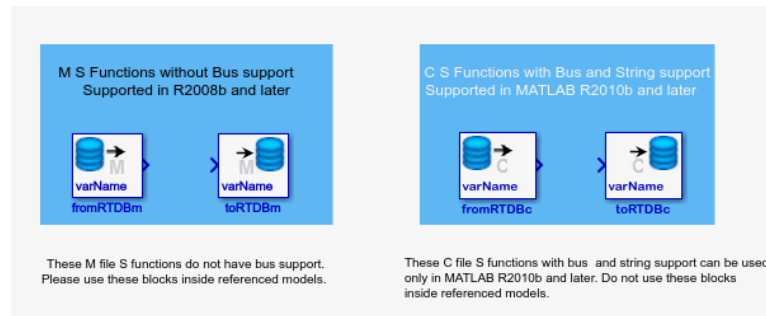


Figure 6.11: RTDB library inside Simulink.

Two blocks, the ones with the M, refer to matlab variable, while the ones with the C refer to C language variable; for each kind then we have the block assessing for the reception from the concurrent and the one for the outgoing from the Simulink model.

These blocks are characterized by some parameters to be imposed, like the name of the variable (which must be the same as the block's name), the dimension of the signal, the sample time and others.

- Take the Simulink model you want to use and *build* it. In order to do so you need to set some parameters like the language (C, C++ etc ..) and then click on the corresponding tab (the one inside the circle in the following picture):



Figure 6.12: Build tab.

This procedure actually generates the code of the simulink model (in the language you have decided). In this way you no longer need Simulink itself to exploit what is inside the model.

- SimWorkbench has a Simulink toolkit which can be installed and has a Graphical User Interface with 4 different options, but two of them are the relevant ones:
 - Simulator Access which allows the user to connect to the Concurrent machine via ethernet by specifying all the required informations (Host name and so on). It is important to remember to unload the RTDB on SimWorkbench (if there was one loaded).

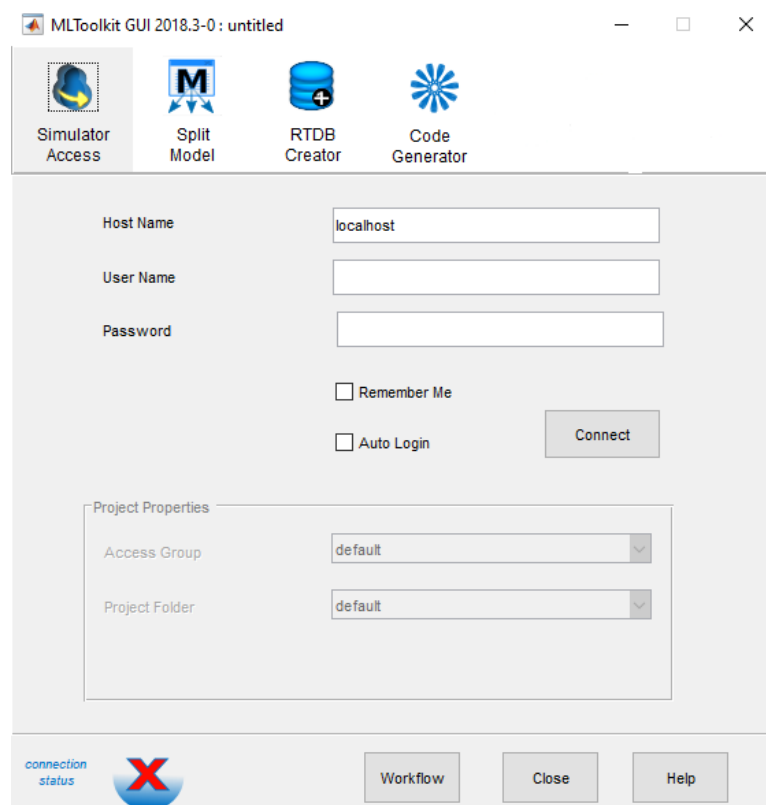


Figure 6.13: Simulator connection by means of SimWB Toolkit.

- Code Generator which actually generates the code and export it on the concurrent machine. In this way you will be able to visualize the model from the GUI of SimWorkbench.

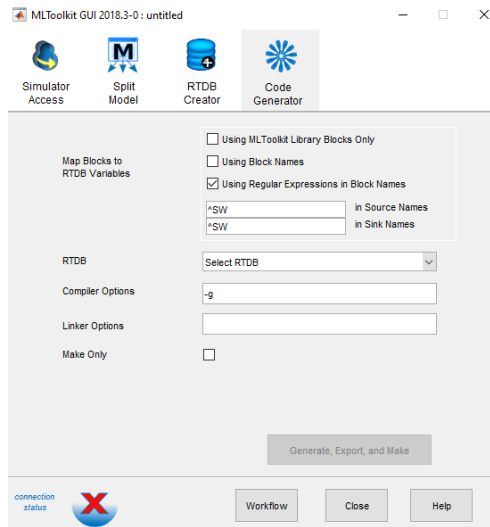


Figure 6.14: Code generation by means of SimWB Toolkit.

Once the code has been generated and sent to the concurrent, you will be able to visualize it from the GUI of SimWorkbench.

- Open the Test which is going to be used and add the corresponding model to it. Let us recall that the model will have the same name as you have saved it on Simulink, and if you will apply changes to it and perform again the steps for the compilation it will be automatically modified on SimWorkbench too.

2) UDP communication The other way of exploiting a Simulink model in co-simulation is to run the simulink model on an additional machine and then streaming the data *from/to* the concurrent via UDP (User Datagram Protocol); thanks to its characteristics, it is able to support a real-time simulation. Let us see which are the steps to be followed on the tow sides (Concurrent and Simulink):

- Concurrent side: as you can see in the following picture, click on the tab on the top left assessing for Inputs/Outputs devices.

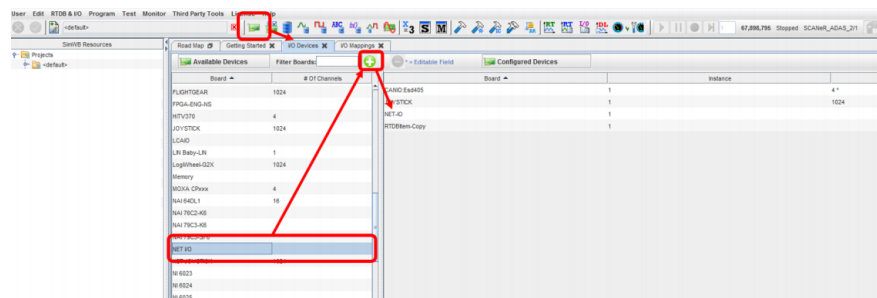


Figure 6.15: Net I/O on SimWB.

Look for *NET I/O* in the list and press the "+" button, then save with the top left green button. In this way, we have made sure that we can receive and send information over the network via UDP.

Now we need to load the RTDB (let us recall that each RTDB has its specific way of retrieving and sending data) and then, as in the following picture, press the icon on the top to open the *I/O Mappings* tab. In this page, we need to link the inputs and outputs, that will be streamed via udp, to the specific RTDB channel.

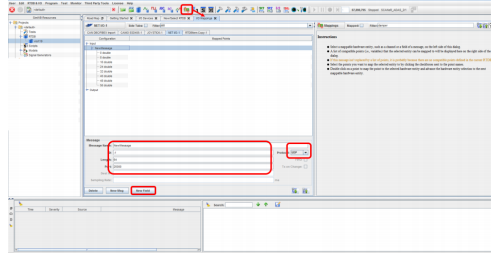


Figure 6.16: Add new messages on I/O Mappings

Messages have to be created and for each of them we need to specify some parameters and informations like the kind of protocol to be used (even TCP is available, but it does not support real time application), the message length in terms of bytes and so on. Once we have created the message, we need to add to it all the fields: they actually represents the variable we are taking from the simulink model running on the external machine and sending to the corresponding RTDB channel.

- Simulink side: here the creation of the model is performed by means of other blocks which are present in the Simulink Library belonging to the SimWorkbench toolkit:

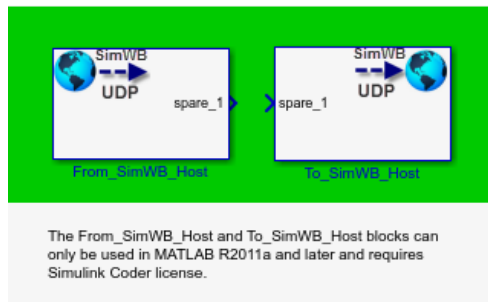


Figure 6.17: UDP communication Library in Simulink.

As you can see, two kinds of blocks are available: the ones assessing for the reception of data from the concurrent (therefore outputs of the simulation) and the ones for the sending of data to the concurrent (inputs for the simulation). As always, the user needs to set some parameters up (the communication port, the I.P. addresses of the two machines and so on) and then the simulink model is complete.

6.3 Modified architecture for the experimental process

As you may have realized, the overall structure of the simulator requires a huge amount of electric power. Since the validation of a control model usually requires many attempts and as a consequence a discrete amount of time, I decided to slightly modify the architecture for performing the simulations, in order to reduce the power requirement.

The general idea is to cut out as much part as possible from the simulator, in order to decrease energy consumption. In particular, the components which makes up my architecture are:

- **Concurrent Linux machine:** this is mandatory in order to allow the real time communication between the simulation's scenario, the car and the data streaming with the other machines.
- **Broadcast Windows machine:** this is one of the many additional windows machine that are used within the static simulator and it is also the mandatory one. Different functions are associated with it:
 - *Visual*: it show the simulation. Let us recall that when the simulator is working at its "best", it is connected to a 7 meter curved screen; here the simulation is displayed in the broadcast only, thus reducing a lot the power consumption.
 - *Sensor*: the broadcast is associated with the correct functioning of the sensors on the ego vehicle inside WorldSim environment.
- **Additional Windows machine:** this is the machine which is responsible for the compilation of the Simulink control model.
- **Steering Wheel:** this tool is used to make the experimental tests easier to be performed and controlled. With this additional steering wheel the user can handle the simulation from the control room without the need of another person on the simulator.

In order to sum up and make the architecture more clear, a schematic representation is shown below:

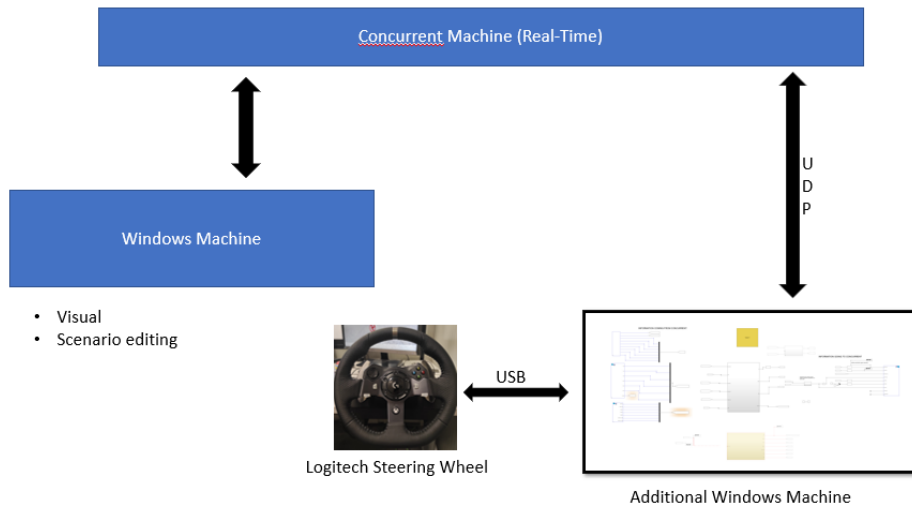


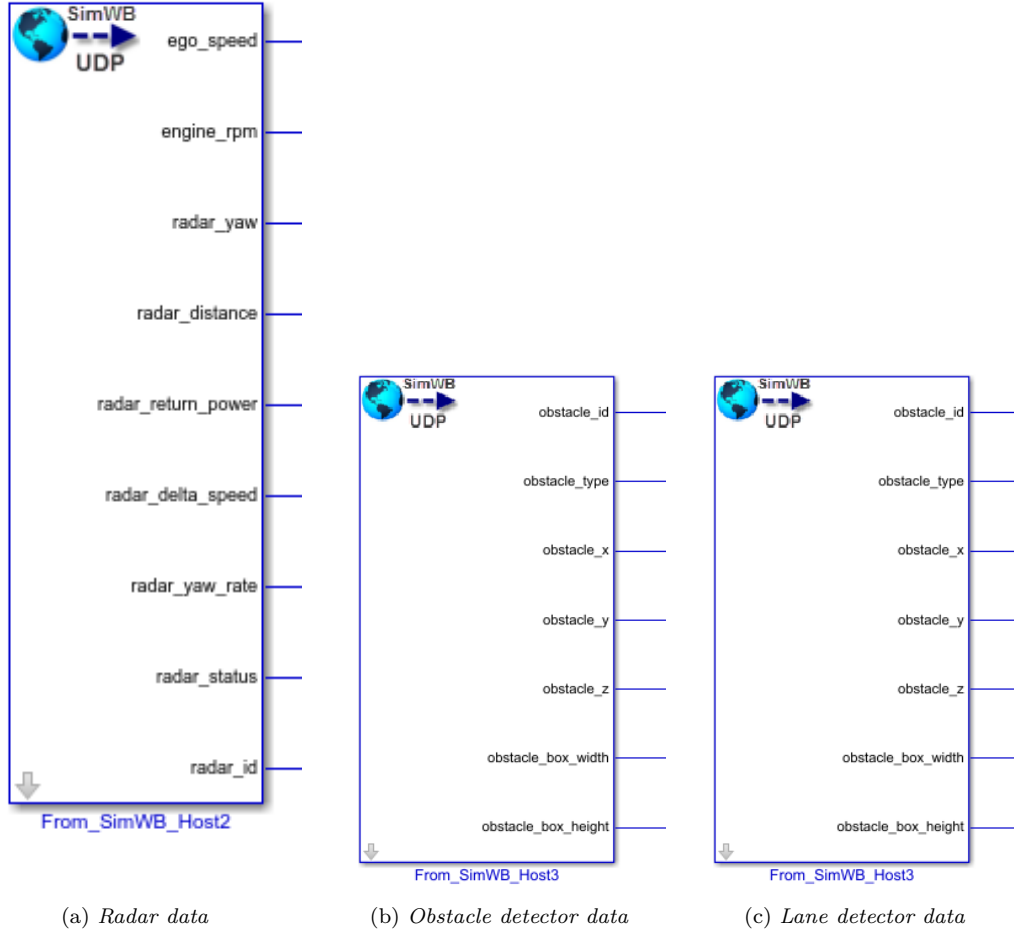
Figure 6.18: Modified architecture of the simulator for experimental tests.

6.3.1 UDP Communication

As already mentioned before, when dealing with a UDP communication with the Concurrent (with SimWorkbench), you need to properly set up two sides: the Simulink model, in which the *From SimWorkbench Host* and *To SimWorkbench Host* blocks are used to, respectively, receive data from the simulation and send them back.

Simulink side

The simulink side dedicated to this operation is shown below:



At first, the model was born to receive all the signals from the concurrent on a unique port. After some attempts though, I have realized that the overall number of signals was too high to be sent in this way, thus resulting in problems of stability and connection. For this reason, I have decided to split the data on three ports, reducing the amount of signals streamed on a single channel, as you can see in the figure.

On the contrary, the signals which are streamed out of my PC towards the concurrent are lower in number, thus needing one port only:

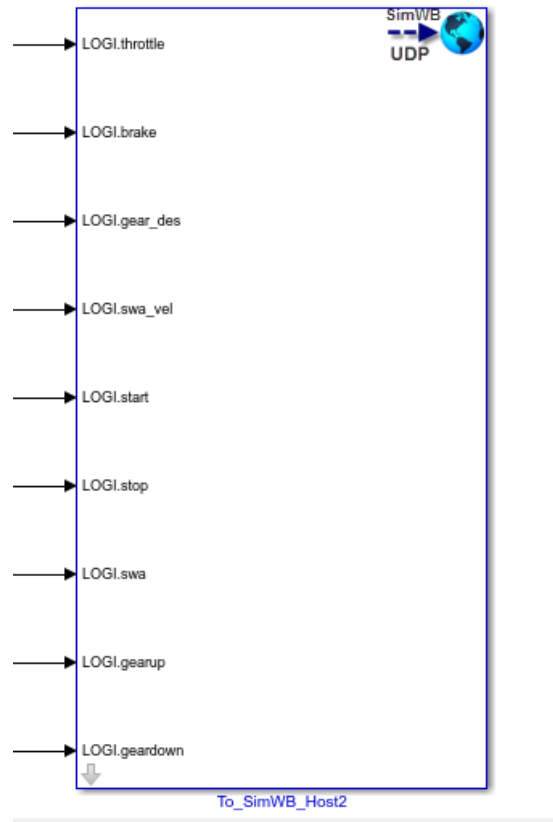


Figure 6.19: Data streamed towards the concurrent

From figure 6.19 it is possible to see what are the commands and the inputs for the simulation. In particular:

- LOGI.throttle stands for the throttle command, replacing the human pedal request.
- LOGI.brake stands for the brake command, replacing the human pedal request.
- LOGI.gear_des stands for the gear which is required. In this case the vehicle is an automatic one and the gear shifting has been simplified with a *look-up* table (figure 6.20) with values of gear depending on the speed of the ego vehicle (keeping at the minimum the environmental impact):

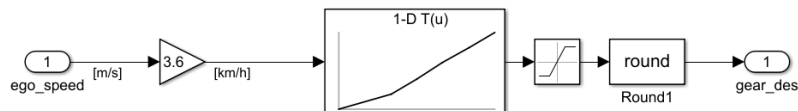


Figure 6.20: Gear shifting LUT

- LOGI.swa_vel stands for the steering wheel velocity command, replacing the human request.
- LOGI.start is a boolean which is used to start the simulation directly from the simulink model.

- LOGI.stop is a boolean which is used to stop the simulation directly from the simulink model.
- LOGI.swa stands for the steering angle command, replacing the human request.
- LOGI.gearup stands for the gear up command, replacing the human request (not used).
- LOGI.geardown stands for the throttle command, replacing the human request (not used).

Concurrent side

Once the simulink side is ready, we need to act on the Concurrent. All the channels need to be properly set up, in order to have compliance between what is received by the simulink model, therefore the outputs of the simulation, and what is sent by it, therefore the inputs for the simulation.

This operation is performed on the tab dedicated to the I/O Mappings, whose procedure has already been described previously.

6.3.2 Steering Wheel

Since I had the chance, I decided to start my experimental and validation procedure exploiting a **gaming steering wheel**: in this way a huge power consumption was avoided.

The world of gaming has been influenced by the technological development of the last years. This can be observed not only from the graphical improvement, but also from the equipments point of view: if we consider the the driving simulator's games, the steering wheels are more and more accurate and they degree of fidelity is higher and higher. This particular steering wheel is equipped with a motor which is able to reproduce the feedback torque, thus reducing the gap between the game (simulation) and the reality.



Figure 6.21: Steering wheel

Compatibility problems

Unfortunately, the steering wheel was not compatible with the version of Concurrent I was using. This led me towards another direction.

At first, the idea was to follow the first option that we have discussed in the previous section: compiling the simulink model and sending it to the Concurrent is actually the best solution since it does not involve any additional communication (the base of the second option, the streaming of data via UDP), which could potentially lead to delays in the co-simulation.

The fact I was not able anymore to connect directly the steering wheel to the Concurrent led to to choose the second option, thus creating the communication via UDP between my Windows machine, which runs the control model, and the Concurrent.

Reception of inputs from the Steering Wheel on Simulink

In order to see and read the signals from the steering wheel, the **Joystick Input** block has been used:



Figure 6.22: Joystick Input block

It allows different configurations (depending on the type of joystick that is used) which can be set by clicking on the block and set the corresponding parameter (in my case is to be set to 1).

As Input it can receive the *Feedback Force* and it generates two Outputs:

- *Axes*: with a Demux block it is possible to divide between Steering Wheel Angle, Throttle, Brake and Clutch (the last three arrives from the pedals (the Steering Wheel comes with a pedal set). In the following picture you can see I had to use some gains in order to get compliance between the outputs coming from the Steering Wheel and the inputs for the DriveSim software.

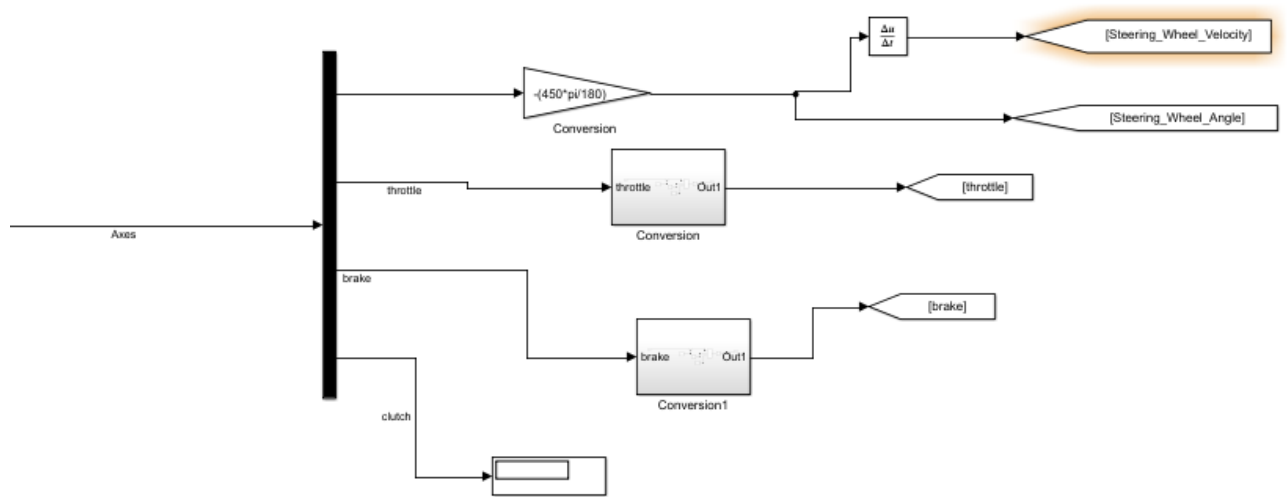


Figure 6.23: Demux of Axes from the Controller

- *Buttons*: up to 18 buttons can be configured. The only thing the user has to take into account is that each joystick will have different buttons associated with the outputs of the simulink block, thus it is required an operation of finding the proper correlation. As you can see in the following picture, 4 buttons has been used in my case (the two paddles for the commands "Gear Up" and "Gear down", and two other button for starting and stopping the simulation), but the block allows the user to configure up to 18 different signals:

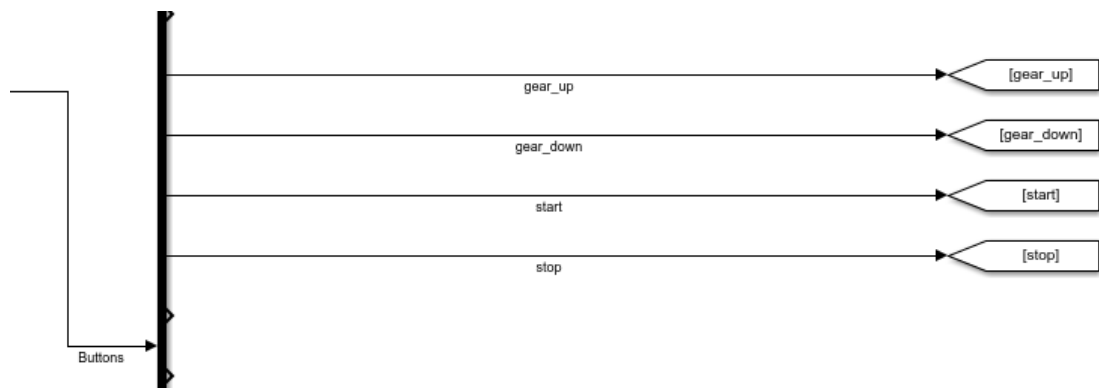


Figure 6.24: Demux of Buttons from the Controller

Validation of the model, PID tuning and final changes

Once the communication between the Concurrent and my PC has been set up, the *compliance between the inputs and outputs of the control model and the ones of the simulator* had to be found.

In order to do so, some tests has been performed and one at a time small changes to the model had been made: for example some measure units were different (e.g. speed $[m/s]$ vs $[kh/h]$) or the reference frame was so that the values of the distances from the left lane had a negative sign. These were only a couple of examples to explain the procedure of getting compliance of the different systems.

As this first step has been ultimated, now it was time for the first actual test of the model. When we have a controller, one of the mai thing to be carried out is the *tuning* of the parameters that characterize it.

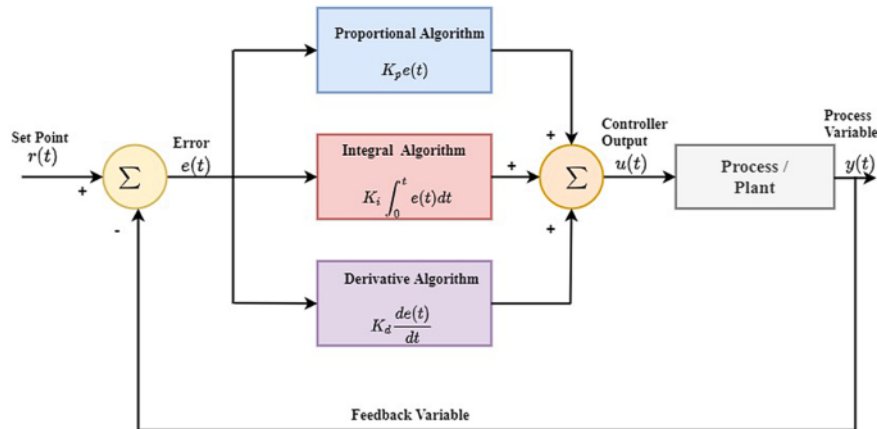


Figure 6.25: PID scheme

Let us recall that, as it is depicted in figure 6.25, the PID controller is based on three contributions: the proportional part, the derivative part and the integral part. Each of these parts if characterize by a coefficient and depending on the three total coefficients, the response of the system to a variation from the set point changes, in terms of stability, overshooting and time to reach the set point.

As a matter of fact, the procedure of tuning is the search of the optimum coefficients which lead to the best compromise between oscillation of the response and time to reach the set point.

I decided to go for an experimental way to find the best coefficients (let us recall that this holds for all the PID controller which are present in the model). Starting from nil values of the three parameters, I started to increase their values and perform tests at each changes (changing one parameter at a time), until quite satisfying results had been obtained (In figure 6.26 an example of the Excel table to keep the results is shown).

Let us underline that these tests for the tuning of the PID parameters have been performed without the model. By imposing at first constant, then ramps and finally sine waves, it is possible to test the control offline.

Case study	P [-]	I [-]	D [-]	Time to reach target [s]	N° of crossings [-]	Notes
case 1	0.38	0	0	5	4	
case 2	0.8	0	0	inf	inf	
case 3	0.5	0	0	inf	inf	
case 4	0.38	0.01	0	5	4	not reaching perfectly
case 5	0.38	0.5	0	inf	inf	many small osciollations (unstable)
case 6	0.38	0.25	0	10	10,11,depends	not stable
case 7	0.38	0.12	0	6.5	5	
case 8	0.38	0.08	0	7.2	5	
case 9	0.38	0	0.1			
case 10	0.38	0	0.05			
case 11	0.4		0.03	3.5	2	
case 12	0.4		0.02	3.7		
case 13	0.4		0.025	uguale		
case 13	0.35		0.03	3	1	piccola instabilità a 1.5s
case 14	0.36		0.03	2.8	1	
case 15	0.36		0.035	2.9		meno stabile
case 16	0.365		0.03	2.85		Rampa non seguita, taglia a gradini
case 17	0.365	0.1	0.03	7.5	1	
case 18	0.365	0.05	0.003			

Figure 6.26: PID search for optimum tuning

Once the tuning procedure has been ultimated, it was time to test the model. As the simulation started though I started to notice problems in the PID for the torque computation of the lateral control. As the request of a specified steering wheel angle arrived, the control started to oscillate up to reach *divergence*. After some days of further tuning, the results were still the same. Some possible reasons behind this phenomena are:

- The steering wheel sensitivity was not hight enough to apply very small changes in the input torque. This was shown by some tests in which very small changes in torque was sent to the wheel, but no movement was perceived. Being the road very straight, the required steering wheel angle and therefore the feedback input force were very low. For this reason the steering wheel did not “see” some inputs, thus resulting in the accumulation of error; as the error was high enough to be perceived by the steering wheel, the control was not able anymore to reach stability.
- The overall architecture and several passages of signals throughout several machines have led to some latency and therefore an incorrect control.

Regardless the actual reason behind the improper functioning of the model with the steering wheel, the model has been slightly changed in terms of lateral control. Now the output of the PID is already the steering wheel angle to be sent as input of the simulation.

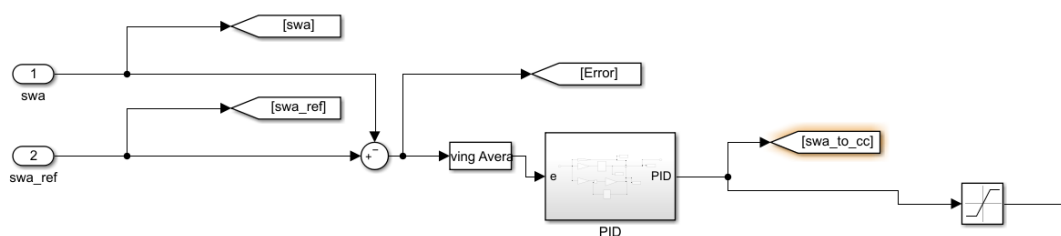


Figure 6.27: New PID output

By doing so, the control model works correctly and the vehicle is controlled properly by the model.

Chapter 7

Results and conclusions

7.1 Results from the control model and the simulation

7.1.1 Adaptive Cruise Control

Let us recap the aim of the ACC: it is used to control the speed of the ego vehicle on the basis of the one of the car in the front of it, in order to automatically brake or throttle if the speed changes. For this reason, several triggers (three in particular) have been used to modify the speed of the vehicle in front of the ego car. In this way we had the chance of testing how the control model reacts to changes in reference speeds and adapts to these changes.

In the following, a matlab plot is used to show how the speed of the ego vehicle is more or less almost the same as the leading car (despite of some high frequency oscillations and a final numerical spike).

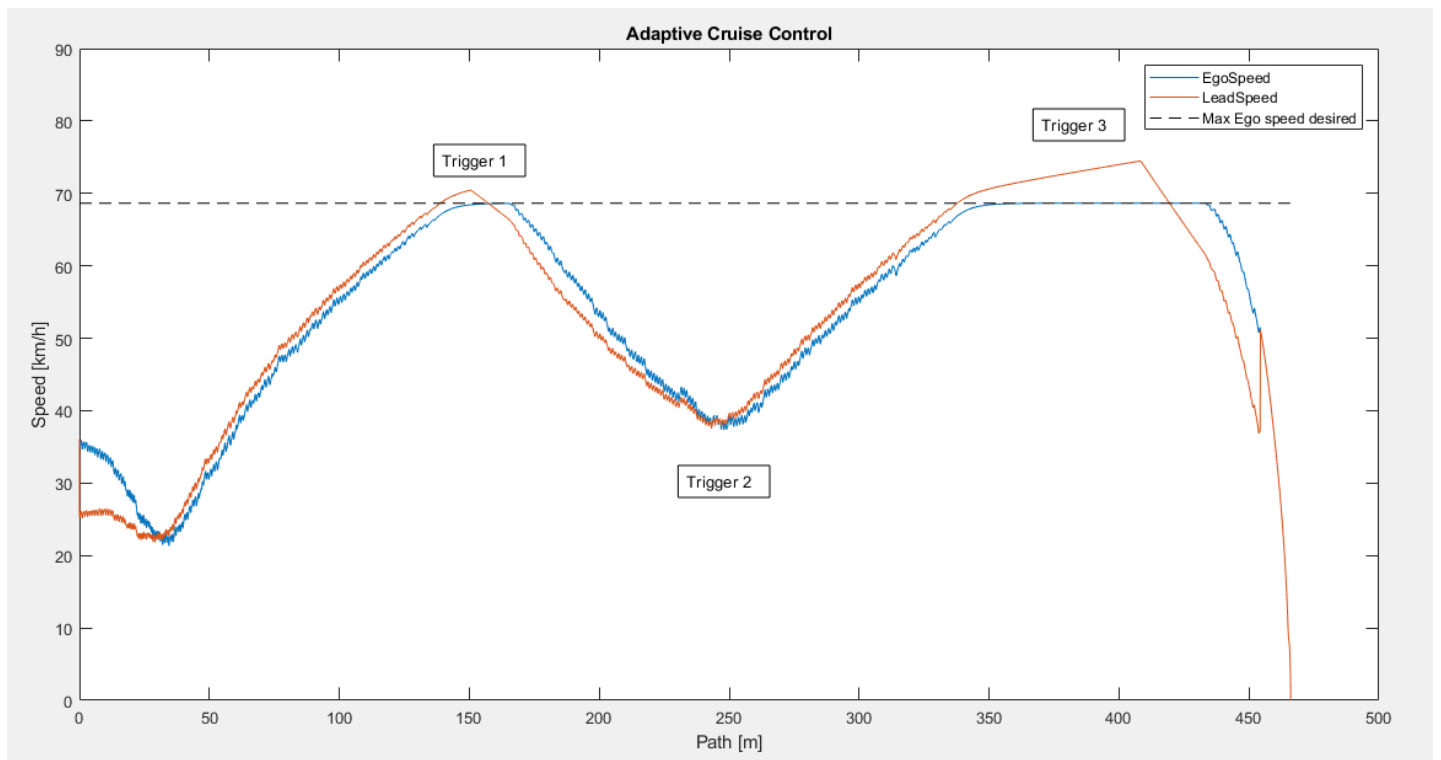


Figure 7.1: Adaptive Cruise Control effect

As you can see from figure 7.1, the effect of the three triggers is perceived in terms of leading

vehicle speed:

- Trigger 1: reduce the maximum (according to traffic limitations) speed of the vehicle with a factor of 0.7.
- Trigger 2: increase the maximum speed with a factor of 1.4.
- Trigger 3: deactivate the leading vehicle, thus meaning its required velocity is set to 0.

In this way it is reproduced the situation in which a vehicle instantly brakes down to zero speed and the human driver, for accident, is not able to brake him/herself and is helped by the control system.

7.1.2 Lane Keeping Assist

The aim of this part of the control is to keep the vehicle inside the lane, in order to avoid possible crashes due to lack of attention of the driver leading the car out of the correct trajectory.

In the figure below we can see the effect of the control; a (small) reference steering wheel angle, computed on the basis of the lane markings, is followed:

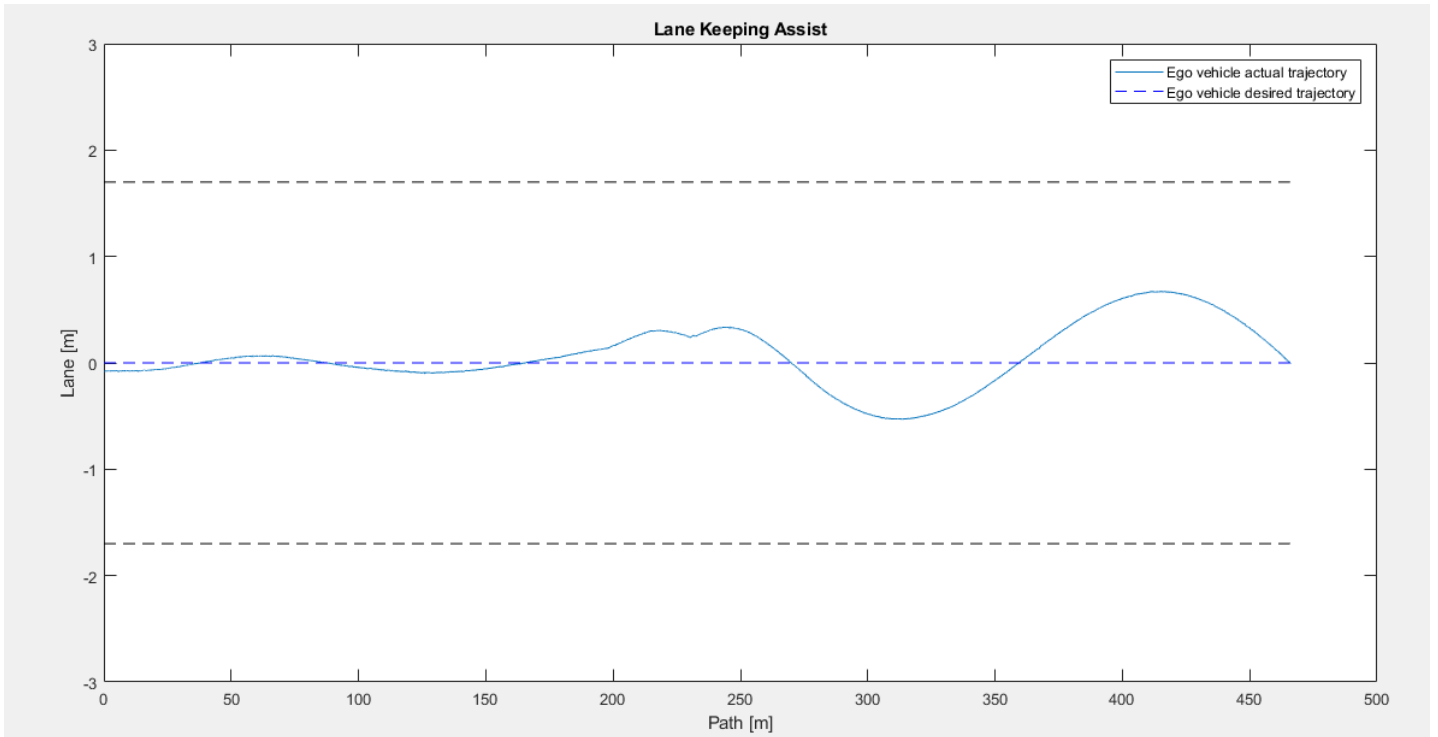


Figure 7.2: Lane Keeping Assist effect

It is clear that the lateral control is not perfectly tuned as the oscillations reveal. As we will see in the last section, this actually represents one of the possible next step for this project; a good tuning may lead to find out if the architecture with the steering wheel is actually the problem in the incorrect control of the vehicle.

7.2 Conclusions

During my thesis project, I had the chance of realizing the importance of driving simulators inside the automotive field; this is perceived both in terms of cost reduction and new technology development.

The first step is realizing, on the basis of the application, the required level of accuracy of the simulation software. Several packages are available and they are characterized by different features and properties, mainly depending on the cost (if present) of the license. The higher the cost, the wider the range of applications as well as the accuracy of the simulation. At the same time, again on the basis of the application, the user (the company) is also able to decide the required level of fidelity of the simulator, choosing the best fit through the wide range of available options.

It is clear that the higher the fidelity of the simulator and the editable potential of the software are, the higher the return in terms of income and development of the company will be. If we think at processes like Hardware In the Loop, Driver In the Loop, Model In the Loop and others, an advanced architecture leads to more applications.

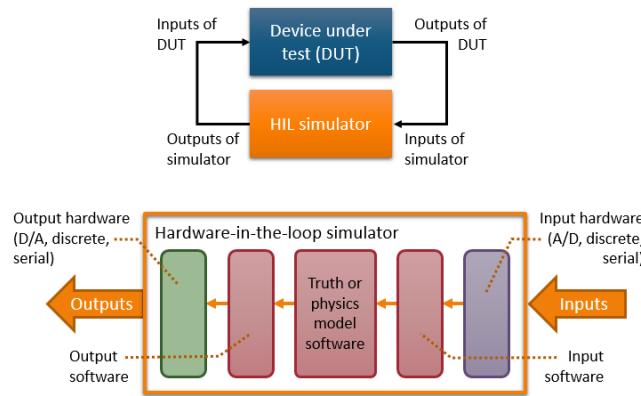


Figure 7.3: HIL process

In parallel with the economic and company development side, there is also another aspect to be considered when dealing with driving simulators: the possibility of testing ADASs and autonomous driving vehicles. Nowadays the automotive field is more and more focused on the development of these fields for many reasons: reduction in accidents and injuries due to human distractions, increase in the efficiency of the drive, reduction in terms of pollution and so on.



Figure 7.4: ADAS

The drawbacks in the development of these systems is that they can not be tested, at the first stages, with real drivers and on normal streets. For this reason, the driving simulators are very effective in this field; one of the most important feature to be checked here is the possibility of adding cameras and sensors to the ego vehicle, but even a good data logging as well as real time data analysis is necessary.

7.3 Next possible steps

Let us see what could be some possible steps forward.

More accurate PID tuning The results from my experiments shows that a better solution in terms of parameters tuning for the PID controllers is possible. Lower oscillations, smaller time to reach stability and ability to follow, in particular in terms of LKA, even more “difficult” reference values.

Test of the controller on other scenarios The extra-urban scenario is quite typical when dealing with ADAS, in particular in terms of NCAP evaluation tests. At the same time though, there are others scenarios that could be tested, like roundabouts, pedestrian crossings and in general urban scenario situations, as well as highway scenarios and so on. The longitudinal control and the lateral control should work in almost all situations (with maybe some changes to the model).

Obstacle avoidance The coupling between lane keeping assist and adaptive cruise control can also be used for obstacle avoidance situation. If we think at the scenario that has been generated and used , there could be the need of overtaking the obstacle (in this case a braking vehicle) and it could be of great interest the development of a control which is able to do this almost automatically.

Implement a better architecture on the static simulator The reduced amount of time to carry out the thesis project led me to concentrate myself on some aspects and to not investigate others.

The problem with the steering wheel is not clear at this time and there is the chance of solving it, with further tests and analysis.

Integrating within a HIL process Another possible application could be integrating the control model within a Hardware In the Loop process to see if it is applicable in that case too.

Validation of the model Finally, a validation procedure could be carried out. A particular test could be firstly on the road then (keeping in mind to have the same vehicle model, same traffic condition and so on) in a virtual scenario. By comparing the data coming from the real and virtual test, a validation of the model can be done.

Appendix A

Simulink Model

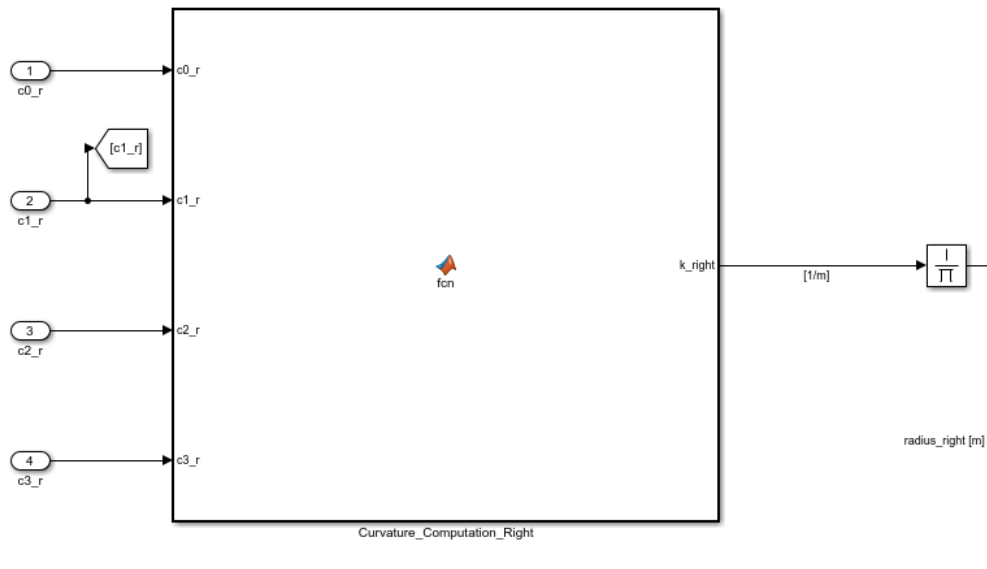


Figure A.1: Curvature computation block.

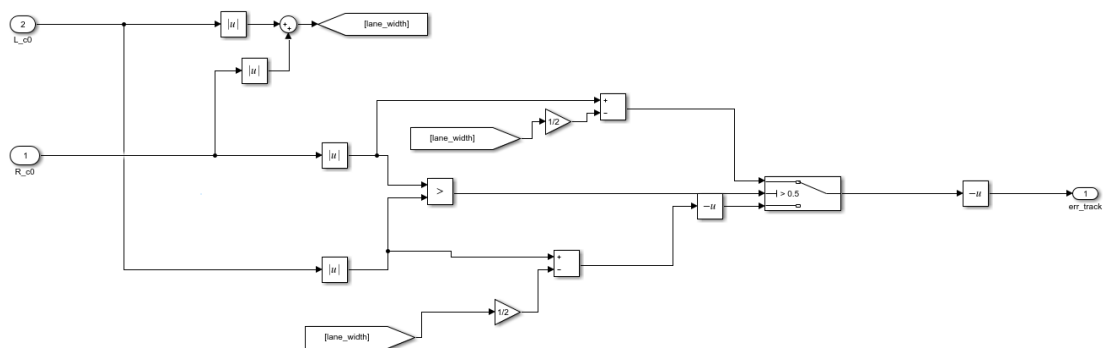


Figure A.2: Cross track error computation.

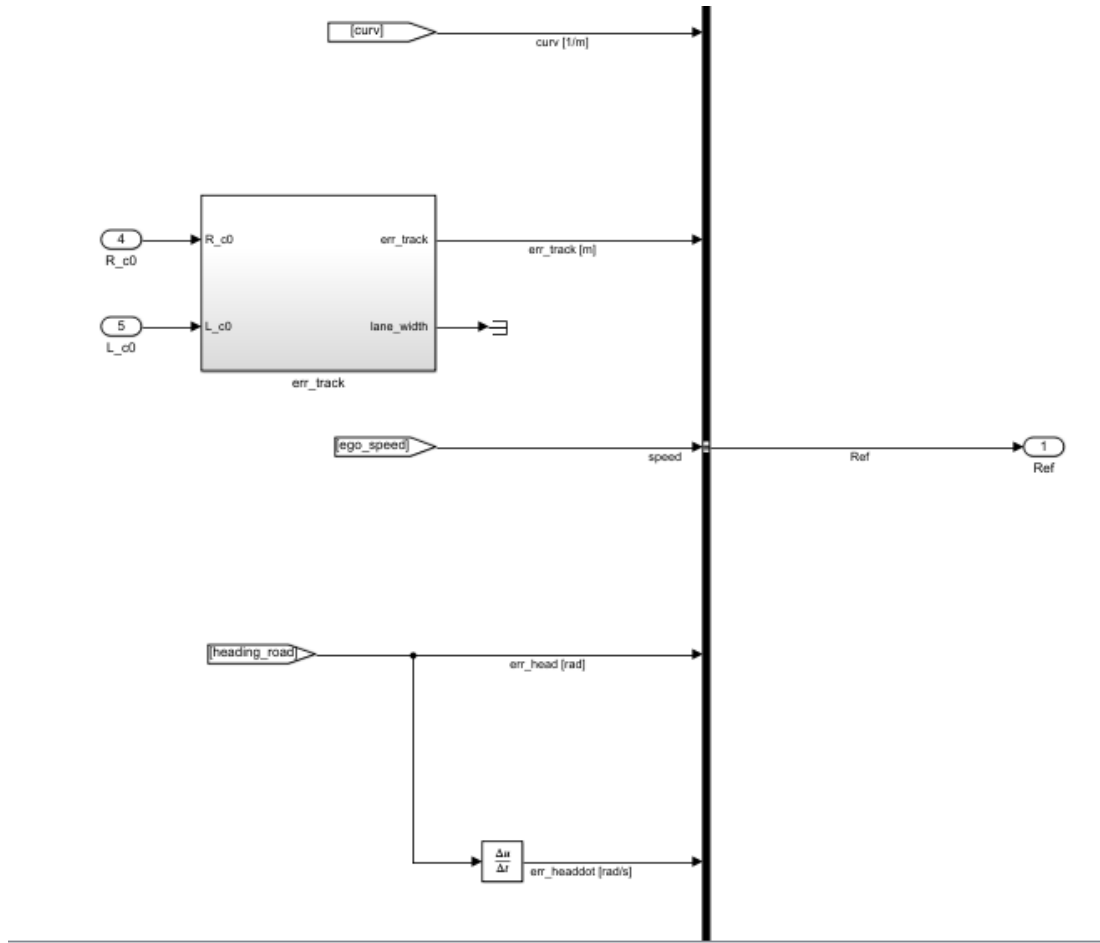


Figure A.3: Reference generator (LKA).

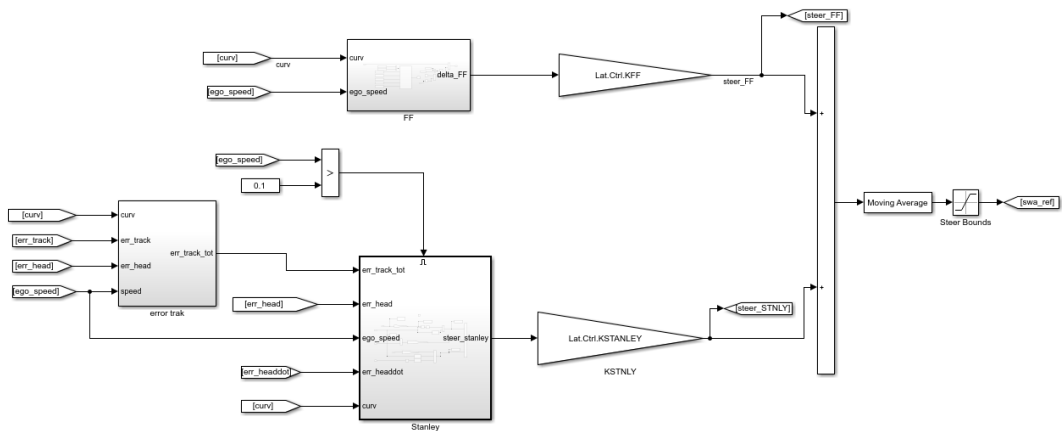


Figure A.4: Reference steering wheel angle generator.

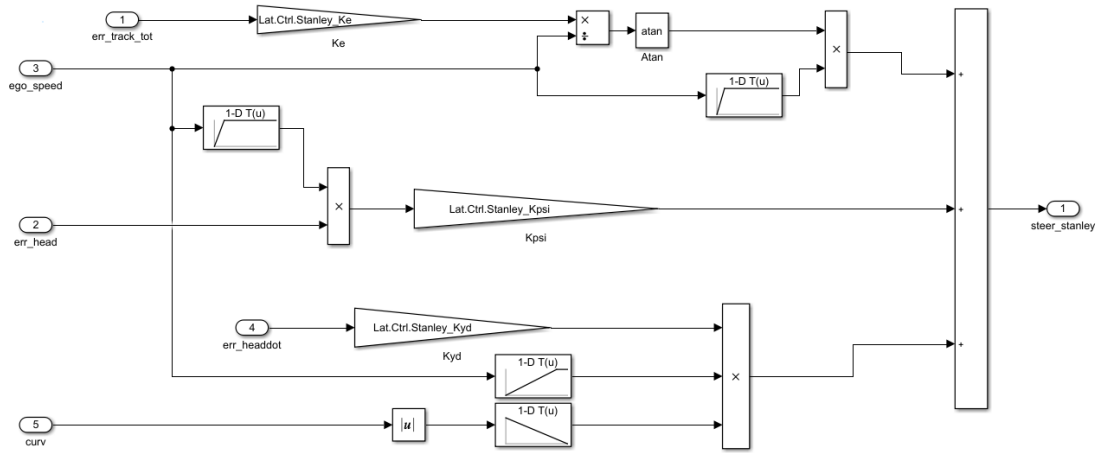


Figure A.5: Stanley block.

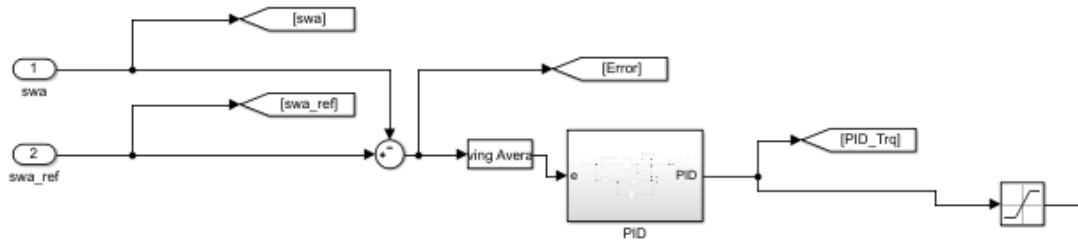


Figure A.6: Control torque computation.

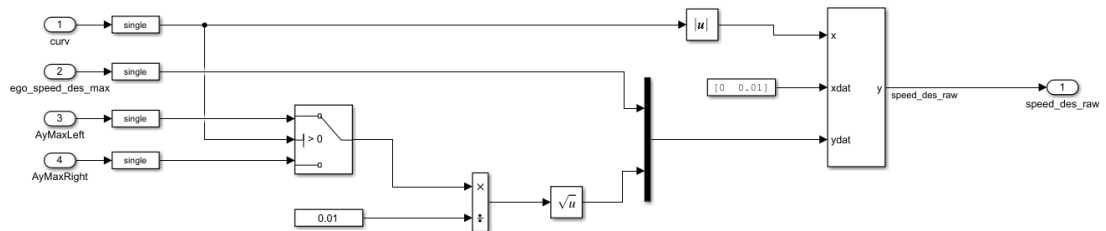


Figure A.7: Ego raw speed desired computation.

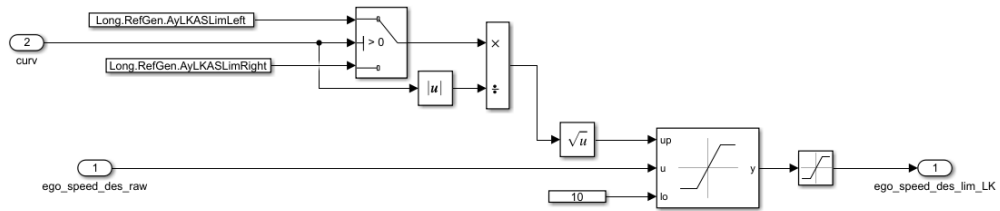


Figure A.8: Ego speed desired computation.

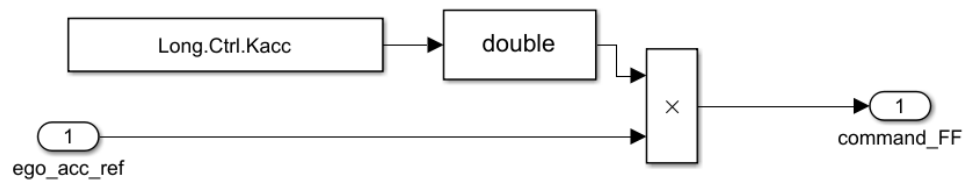


Figure A.9: Feedforward part for commands computation (ACC).

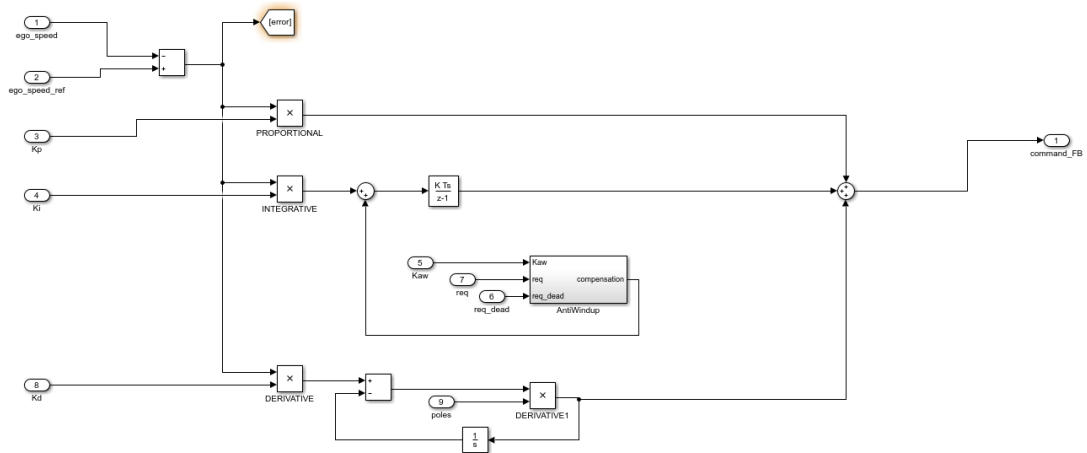


Figure A.10: Feedback loop for commands computation (ACC).

Bibliography and Sitography

- [1] Genta G. *Vibration Dynamics and Control*. 1993.
- [2] Hoffmann M., Tomlin J., Montemerlo M., Thrun S. *Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing*. 2007.
- [3] Jacobson B. *Vehicle Dynamics*. VIII Edition, 2017.
- [4] Waslander S. *Introduction to self driving cars, Lesson: “Geometric Lateral Control - Stanley”*.
- [5] https://en.wikipedia.org/wiki/Driving_simulator.
- [6] <https://en.wikipedia.org/wiki/Simulation>.
- [7] https://en.wikipedia.org/wiki/Advanced_driver-assistance_systems.
- [8] https://en.wikipedia.org/wiki/Euro_NCAP.
- [9] <https://www.pathpartnertech.com/understanding-radar-for-automotive-adas-solutions/>.
- [10] https://en.wikipedia.org/wiki/Driving_simulator.
- [11] <https://it.wikipedia.org/wiki/Radar>.
- [12] <https://www.fierceelectronics.com/components/lidar-vs-radar>.
- [13] https://en.wikipedia.org/wiki/Hardware-in-the-loop_simulation.
- [14] <https://injuryfacts.nsc.org/motor-vehicle/occupant-protection/advanced-driver-assistance/data-details/>.
- [15] <https://www.synopsys.com/automotive/autonomous-driving-levels.html>.
- [16] <https://www.euroncap.com/it/sicurezza-dei-veicoli/la-valutazione-in-dettaglio/safety-assist/sistema-aeb-car-to-car/>.
- [17] https://it.wikipedia.org/wiki/Robot_Operating_System.