POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Master of Science Degree in Mechatronic Engineering

Master's Degree Thesis

Design and Simulation of Autonomous Driving Algorithms

Relatore
Prof. Massimo Violante

Candidato
Marco Cattaruzza

July 2019

Alla mia famiglia

Summary

In the last years many research groups and companies worked on the development of self-driving cars. Among the first automation layers there is the lane keeping functionality. This thesis introduces two lane keeping control algorithms and a computer vision algorithm for lane identification through a camera mounted on the vehicle. In the first part of the thesis, algorithms for tracking the center of the lane have been developed through PID control and optimal control techniques such as non-linear MPC. The vehicle models exploited for controller design are derived from the well known bicycle model and have been widely used in literature. Starting from a simple lateral dynamics linear model for PID control, a more complicated model has been adopted for its possibility to take into account longitudinal vehicle dynamics and vision dynamics. It has been adopted to design a nonlinear MPC. The algorithm should autonomously steer, accelerate and brake to keep the vehicle on track while minimizing the jerk, the steer rate and the difference between the expected speed and the actual one. In this phase Matlab & Simulink have been widely used. Subsequently, computer vision algorithms have been studied, developed in Python with the OpenCV package and tested with the simulator CARLA. These algorithms are needed to extract useful information from camera images, as the lateral offset of the car from the center of the lane, the yaw error and the lane lines curvature. These data are then used by the MPC controller to compute the optimal control inputs. Finally, the use of Convolutional Neural Networks for lane keeping in urban environments have been addressed. An algorithm to collect training and validation data (images) and labelling them has been coded but no training nor testing has been done for lack of computational power.

Keywords: nonlinear model predictive control, lane keeping, autonomous vehicles, automotive control, perception, computer vision, convolutional neural networks.

Sommario

Negli ultimi anni si è vista una crescita di interesse verso lo sviluppo di algoritmi e tecnologie rivolte ai veicoli a guida autonoma. Tra i primi livelli di automazione, una funzionalità rilevante è costituita dal mantenimento automatico delle traiettorie. La tesi seguente introduce due algoritmi per tale obiettivo ed un algoritmo di visione artificiale per l'identificazione delle linee di corsia attraverso le immagini provenienti da una videocamera montata sul veicolo autonomo. Nella prima parte della tesi sono stati sviluppati algoritmi per il mantenimento di centro corsia attraverso un controllore PID e tecniche di controllo ottimo quale l'MPC non lineare. I modelli dinamici del veicolo utilizzati per la progettazione dei controllori sono stati derivati dal modello della bicicletta, ben noto e molto utilizzato nella letteratura scientifica sull'argomento. Partendo da un modello lineare approssimante la dinamica laterale, utilizzato per il design di un controllore PID, un modello più complesso è stato adottato per la sua capacità di tenere in considerazione anche la dinamica longitudinale e quella data dal sistema di visione. Questo modello è stato utilizzato per lo sviluppo di un *model predictive controller* non lineare. L'obiettivo dell'algoritmo è quello di sterzare, accelerare e frenare autonomamente il veicolo per mantenerlo in centro carreggiata, minimizzando la derivata dell'accelerazione (*jerk*), la velocità di sterzata e la differenza tra velocità reale e desiderata. In questa fase Matlab e Simulink sono stati ampiamente utilizzati. Successivamente, algoritmi di visione artificiale sono stati studiati, sviluppati in Python con il pacchetto OpenCV e testati con il simulatore CARLA. Questi algoritmi sono necessari per estrarre informazioni utili dalle immagini provenienti dalla telecamera, come gli errori laterale e angolare tra automobile e carreggiata e la curvatura delle sue linee. Questi dati vengono poi utilizzati dal controllore MPC per calcolare gli input di controllo ottimali. Infine sono stati considerati, per l'utilizzo in ambienti urbani, modelli di *reti neurali convoluzionali end-to-end* quale il modello NVIDIA. Un algoritmo per la raccolta dei dati in CARLA (le immagini) necessari al modello e per etichettarli, è stato scritto in Python. Tuttavia, per mancanza di capacità di calcolo, non è stato possibile fare il training della rete neurale e quindi testarla.

Parole chiave: model predictive control non lineare, mantenimento della corsia, veicoli a guida autonoma, sistemi di controllo automobilistici, percezione, visione artificiale, reti neurali convoluzionali.

Contents

List of Figures

[1]**http://cyberlaw.stanford.edu/loda**.

[2]**https://law.resource.org/pub/us/cfr/ibr/005/sae.j1733.1994.html**.

List of Tables

Chapter 1

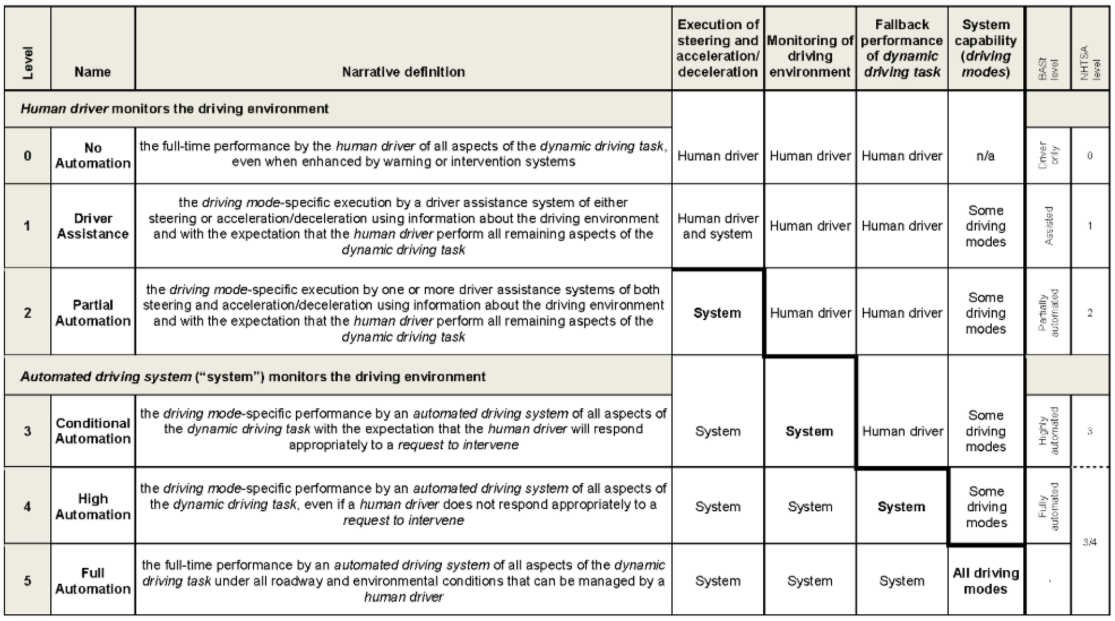
Introduction

Autonomous driving cars can be defined as vehicles that are able to drive in many driving scenarios by substituting the human intervention at different levels. The environment on which the cars have to drive must be considered partially unstructured. If it's evident that streets, signals, semaphores and lanes follow standards to be immediately recognizable by humans, these scenarios are characterized by a certain variability due to different legislations, changing of weather conditions and traffic. Up to now, these are the main challenges that researchers on autonomous vehicles have to deal with.

When talking about self driving cars, people usually have some doubts about the possibility that in a near future these systems will be reality. This happens specifically when fully autonomous vehicles are addressed. A recent American survey [1] reported that more than a half people of the poll are uncomfortable with the idea of riding in self-driving cars, specially non-millennial people. For sure technology is already transforming the way people live and travel but with autonomous vehicles we will have a shift in the way we look at transport systems. From cruise control and lane keeping systems to fully autonomous driving cars, commercial vehicles see an increasing degree of automation for safety and comfort purposes. In addition, the rapid increasing of microprocessors performances, in parallel to a constant drop of their costs, makes possible to monitor and control many active components of a car concurrently with higher precision. It's now feasible to implement complex vision algorithms and trained neural networks on an automotive embedded system.

For sure there will be economic benefits because many people spend a lot of time driving and, specially in big cities, this is largely inefficient. Moreover, there will be a social benefit due to the absence of the human error and accessibility for people that for some reason can't drive. Obviously the transition to autonomous driving brings some challenges too. This challenges are mainly related with decision making at the tactical level, that is related to vehicle manoeuvre selection in a dynamic environment, characterized by a very high variability. The actual algorithms in fact are not able to operate at this level without a complete knowledge of the problem [2].

The Society of Automotive Engineers (SAE) proposed a classification of self driving cars based on 5 levels of automation of the various tasks that are primary in the action of driving (see Figure 1.1). This classification is useful to understand later on the level of automation of the system is presented in this work and the possible evolutions and future goals.



Level	Name	Narrative definition	Execution of steering and acceleration/ deceleration	Monitoring of driving environment	Fallback performance of *dynamic driving task*	System capability (*driving modes*)	BASt level	NHTSA level
Human driver monitors the driving environment								
0	**No Automation**	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a	Driver only	0
1	**Driver Assistance**	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes	Assisted	1
2	**Partial Automation**	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes	Partially automated	2
Automated driving system ("system") monitors the driving environment								
3	**Conditional Automation**	the *driving mode*-specific performance by an *automated driving system* of all aspects of the *dynamic driving task* with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes	Highly automated	3
4	**High Automation**	the *driving mode*-specific performance by an *automated driving system* of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes	Fully automated	3/4
5	**Full Automation**	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**	-	

Figure 1.1: SAE levels of automation in Autonomous Driving[1]

1.1 Thesis motivation

In order to design and realize a complex system like that in a car, it is needed to acquire meaningful information from the environment with many different kinds of sensors, elaborate them and take decisions about the trajectory of the vehicle (trajectory planning) and the actions to undertake to follow this trajectory in the best way. The data coming from the sensors are the "eyes" of the intelligent vehicle and the information have to be extracted with efficient and real time algorithms. For example, to sense the lane computer vision and artificial intelligence algorithms are used. But cameras aren't the only sensors available. Data coming from GPS, LiDAR, radars and others are collected together and exploited. This important step is named *sensor fusion*.

As it's evident, the complexity of these systems is huge. The levels reported in Table 1.1 are related to each other but the problems they face require different tools and skills to be addressed properly. For this reason, this thesis will focus mainly on the operational level, the one that the undersigned judges the most tailored to his skills and knowledge. The reader interested in other levels will find some suggested readings in the references.

In particular, this work addresses the problem of lane keeping on high-curvature roads with only one camera mounted on the vehicle. This is a challenging problem because the speed in this type of roads varies consistently and non-linearities have to be considered in the dynamic model. High steering rates contribute to this.

A problem, that should be addressed in further works, is the efficiency of the control algorithms. Here, some choices has been done to reach this goal, but real-time testing

Level	Characteristic	Example	Existing model
Strategic	Static; abstract	Planning a route; Estimating time for trip	Planning programs (Artificial Intelligence)
Tactical	Dynamic; physical	Determining Right of Way; Passing another car	Human driving models
Operational	Feedback control	Tracking a lane; Following a car	Robot control systems

Table 1.1: The three levels in the driving task [2]

has not been carried on. All the simulation are model-in-the-loop on a Notebook with a i7-7500U 2.90GHz CPU and 16 GB of RAM. The simulators used are Simulink with the Automated Driving Toolbox, described in Chapter 2, and CARLA Simulator, described in Chapter 3.

1.2 Introduction to lane keeping systems

With the increasing speed of modern microprocessors it has become even more common for computer vision algorithms to find applications in real-time control tasks. This, for example, makes possible to address the problem of steer autonomous vehicles along highways with lane keeping support functions using the output coming from one or more video cameras mounted on it. This functionality is very useful to prevent accidents caused by unintended lane departures. It automatically steers the vehicle to keep the center of the lane detected with a vision system.

In the market are present different levels of lane keeping support functions, like [3]: warning functions, intervention functions and control functions. This work address the third category, that is the *Automatic Lane Keeping Control*. The system has the complete control on the steer and, possibly, the throttle or brake of the vehicle and substitutes the human intervention in these tasks. In this case, by looking at Figure 1.1, the SAE level of these kind of system is 2 because the system can autonomously control the steer and the acceleration/deceleration but all the other aspects of the dynamic driving task are left to the human driver.

In this work all the three main building blocks of Figure 1.2 have been analysed, that generally speaking belong to: *Computer Vision* (Chapter 4), *Vehicle Dynamics* (Chapter 5) and *Control Algorithms* (Chapter 6).

The lane detection step converts the input image, coming from the camera, in a more meaningful one, that contains only the pixels associated to lane lines, seen from the top view. This image is then exploited to extract the lane parameters through model fitting techniques. In this work a least-squares approach is used to identify the parameters of a parabolic model and the Hough transform is applied to identify the parameters of straight lines.

Another possibility is to use a Kalman filter for lane estimation, that has the advantage to filter noise and to provide time integration. This way the past measurements are taken

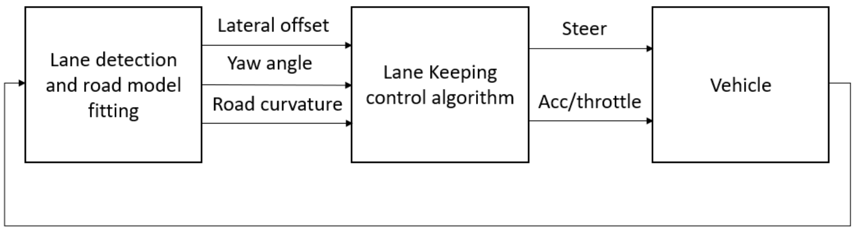

Lane detection and road model fitting
Lateral offset
Yaw angle
Road curvature
Lane Keeping control algorithm
Steer
Acc/throttle
Vehicle

Figure 1.2: Lane Keeping Algorithm building block

into account during the current one and errors in one frame of the camera are handled well [4]. With estimation techniques, the lateral and yaw error of the lane center position and orientation with respect to the ego car are computed and passed to the controller.

In this work, PID and nonlinear MPC has been used, the first for its simplicity, the second for its accuracy to predict the future states of the car. Finally the steering and, in the case of the nonlinear MPC, acceleration commands are applied to the ego vehicle. An example of lane keeping performance metrics is shown in Figure 1.3.

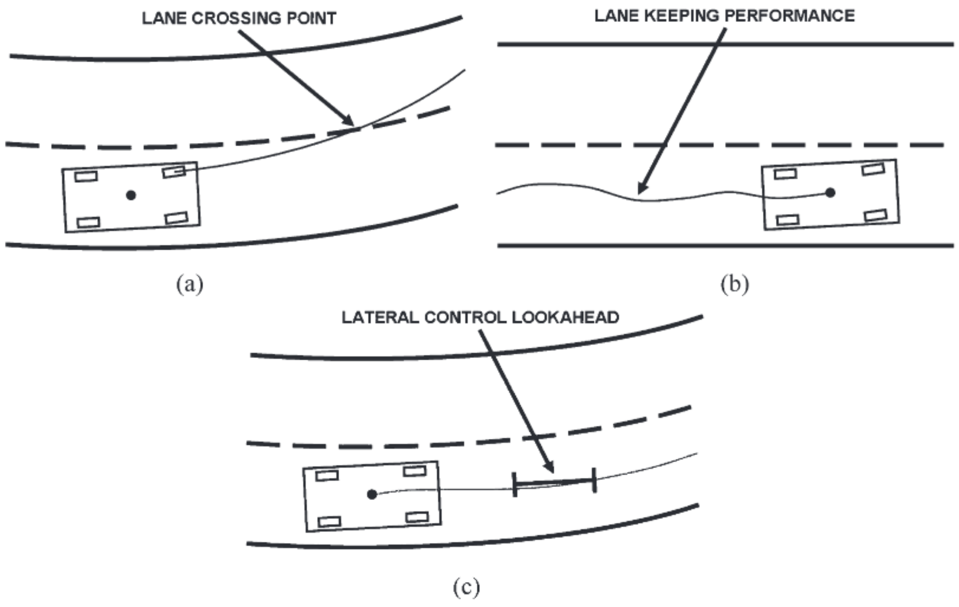


Figure 1.3: Lane Keeping Algorithm: performance metrics. (a) Lane-departure warning, (b) driver attention monitoring, (c) vehicle control [5]

Chapter 2

Automated Driving Toolbox

The Matlab's *Automated Driving Toolbox* provides a set of algorithms and tools for designing, simulating and testing ADAS and autonomous driving systems [6]. The main features of this toolbox exploited in this work are the Driving Scenario Designer App (section 2.2), the Bird's-Eye View plot (section 2.3), the Vision Detection Generator for generating synthetic detections (section 2.4) and the Scenario Reader (section 2.5).

2.1 Coordinate Systems in Automated Driving Toolbox

As reported in [7], the toolbox uses these coordinate systems:

- **World**: A fixed universal coordinate system (X, Y, Z) in which all vehicles and sensors are placed. It's a property of the Driving Scenario and these coordinates can be retrieved during simulation from the Vehicle Block (see Chapter 6). They uniquely define each actor's position in the simulation.

- **Vehicle**: is the vehicle coordinate system (x, y, z) that is also explained in Chapter 5 and is attached to it. x points forward from the vehicle, y points to the left as viewed when facing forward and z points up from the ground to maintain the right-handed coordinate system. This is the ISO convention used by the toolbox and it is different from the SAE J760 coordinate system adopted for vehicle modelling in Chapter 5. A SAE J670E to ISO 8855 conversion will be necessary in the control algorithm to correctly interpret the physical quantities coming from the vehicle dynamics block in order to use them in the controller that uses the Automated Driving Toolbox.

- **Sensor**: coordinate system (X_c, Y_c, Z_c) attached to the sensor in the same way the vehicle coordinate system is attached to the vehicle. That is, X_c points in the direction the camera points to, Y_c points to the left as viewed when facing forward and Z_c points up.

- **Spatial**: coordinate system (u, v) that defines the discrete grid of pixels of an image. In some cases (x, y) has been used without ambiguity.

2.2 Driving Scenario Designer App

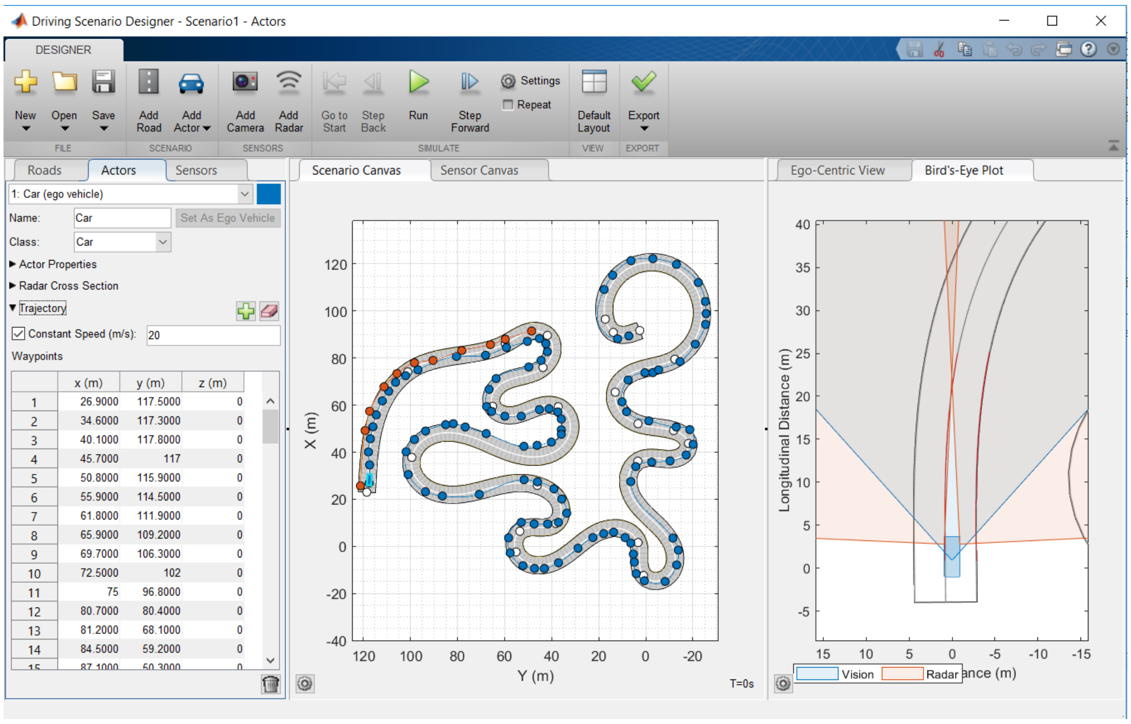
This Matlab application is part of the Automated Driving Toolbox and has been used in this work to design driving scenarios for control algorithms. In Figure 2.1 is visible the user interface with a road and the *ego vehicle* positioned. This name refers to the car that has to be driven autonomously. The blue dots are the waypoints that will be crossed by the ego vehicle during the test of the scenario. During the real simulation their only role is to set the start and the end of the bird's eye view simulation. The other actors instead are dummy and they will go through these points independently.
Building a Scenario from scratch is very simple:

1. Add road(s) and set their parameters like the number of lanes, lane width, lane markings and road center positions.
2. Add actors as cars, trucks, pedestrians and bicycles and set their pose, their physical dimensions and their trajectory. It's important to note that the trajectory of the ego vehicle is not exactly the one will be followed when the autonomous mode will be on. It is only necessary to start and stop conditions in the simulation.
3. Attach sensor(s) to the ego vehicle to get the information we want for our control system. In this work only one camera has been used, but it is possible to add multiple sensors in different positions and orientations relative to the vehicle reference frame. Up to now, only cameras and radars are available. Note that the sensors used in this app won't be retrieved in the workspace and so they won't be used in Simulink but only in this specific app. The only things that will be exploited are the initial pose, the ego speed and the information about the road and other actors.
4. Run the application to see the car moving and following the desired trajectory, detecting lanes with cameras and other actors with radars.
5. Save the model to be used in Matlab scripts.

It is also possible to use prebuilt scenarios.

2.3 Bird's-Eye Scope

The Automated Driving Toolbox offers a nice simulation environment for self-driving cars. On Simulink, after having run the Matlab script to set the controller and all the needed parameters, it's possible to open the *Bird's-Eye Scope* (Figure 2.3), in addition to the traditional *Simulation Data Inspector* and *Logic Analyzer* for logging simulation data (Figure 2.2). This tool offers a top view of the car and the Driving Scenario and offers a visual understanding of what is going on during the simulation and how the car behaves. At each time step the `Scenario Reader` block receives the information on the vector pose and moves the car in the Bird's-Eye Scope accordingly. It highlights also the estimated lanes, from which the `Vision Detection Generator` computes the parameters depending on the camera's characteristics. A picture of this view is presented in Figure 2.4. To start the simulation on the Bird's-Eye Scope, click on **Find Signals** the first time or whenever blocks, ports or signal lines are added or removed and then click on **Run** each time to start



1: Car (ego vehicle)

Name: Car

Class: Car

▸ Actor Properties

▸ Radar Cross Section

▾ Trajectory

☑ Constant Speed (m/s): 20

Waypoints

	x (m)	y (m)	z (m)
1	26.9000	117.5000	0
2	34.6000	117.3000	0
3	40.1000	117.8000	0
4	45.7000	117	0
5	50.8000	115.9000	0
6	55.9000	114.5000	0
7	61.8000	111.9000	0
8	65.9000	109.2000	0
9	69.7000	106.3000	0
10	72.5000	102	0
11	75	96.8000	0
12	80.7000	80.4000	0
13	81.2000	68.1000	0
14	84.5000	59.2000	0
15	87.1000	50.3000	0

Figure 2.1: Driving Scenario Designer App

the simulation. The ego-vehicle will remain in the center of the scope of the driving scenario and the street and other actors will move accordingly to simulate the control actions. It's not necessary to click on **Find Signals** each time a block parameter is modified. An important parameter is the simulation time, that cannot be higher that the time at which the car reaches its final position. This position is computed automatically a priori from the information about waypoints and ego car's velocity in the Driving Scenario Designer App. Otherwise, an error will occur.

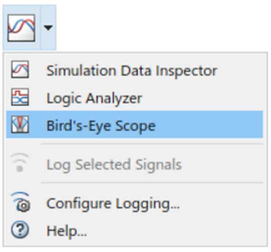


Figure 2.2: Various ways to visualize simulation data in Simulink

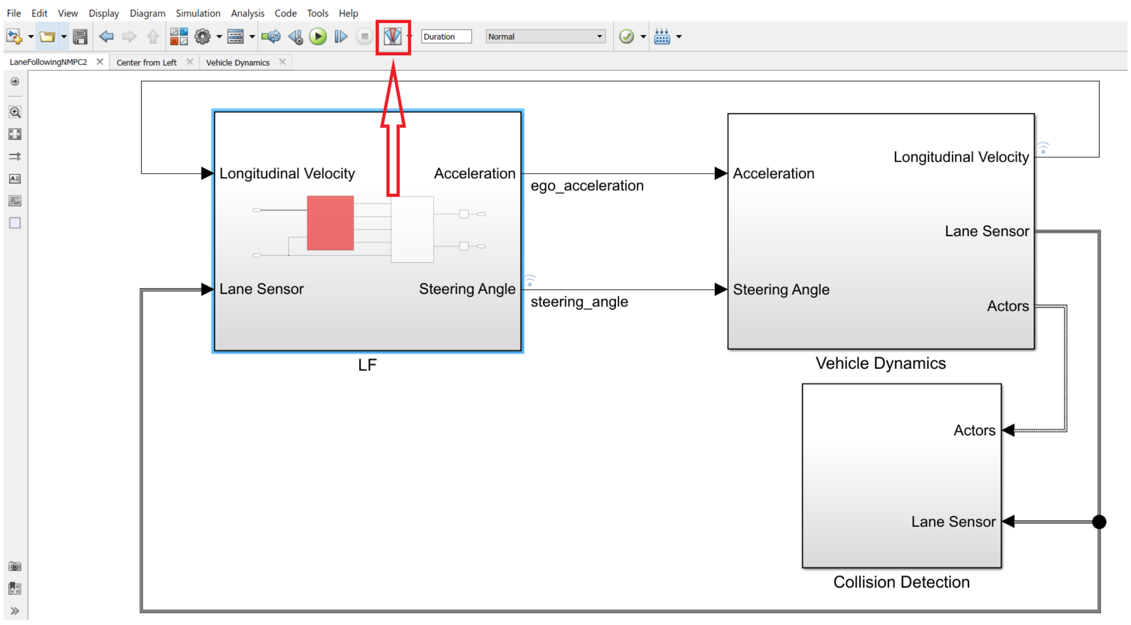


Figure 2.3: Open Bird's-Eye view

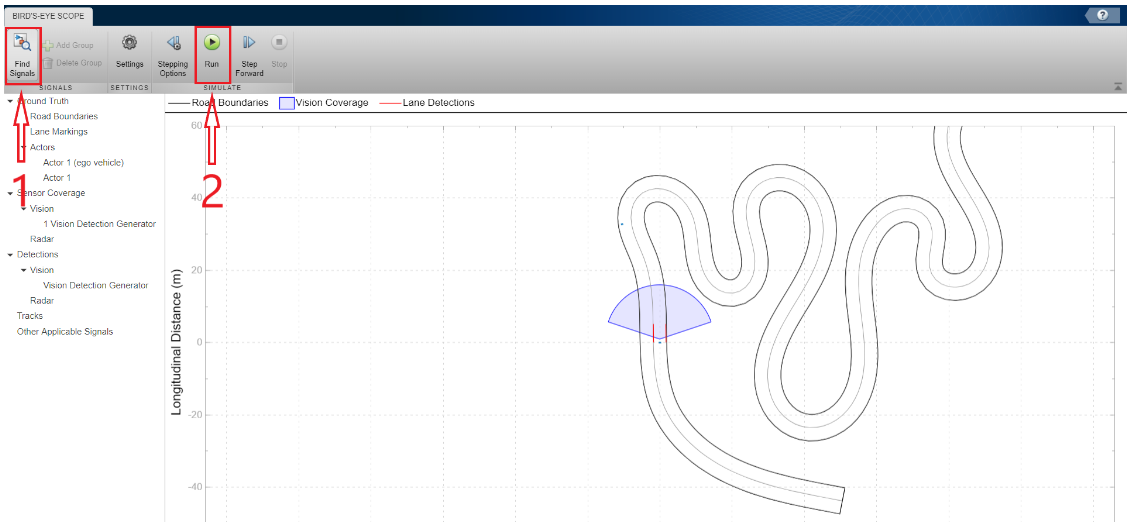


Figure 2.4: Run Bird's-Eye view

2.4 Generate Synthetic Detections

Another important building block of this toolbox is the `visionDetectionGenerator` System object, shown in Figure 2.5. It generates synthetic detections from a camera mounted on the ego vehicle. Its detections are referenced to the ego-vehicle coordinate system and they can interest lane lines or objects. It uses a statistical mode which simulates detections in real world with random noise components and false alarms. It uniquely identifies the sensor and its intrinsic and extrinsic parameters. Also properties about the noise and the accuracy of lane lines and objects detection can be tuned, as well as the maximum detection range. In the case of lane lines detection, the output bus of this system contains the two buses (left and right line) containing the following physical quantities: curvature, curvature derivative, heading angle, lateral offset and strength. An useful block found on the Matlab help has been exploited to convert some angular quantities from degrees to radians, see Figure 2.6.

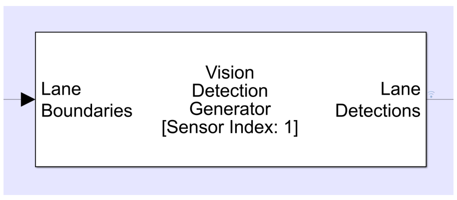


Figure 2.5: Vision Detection Generator block

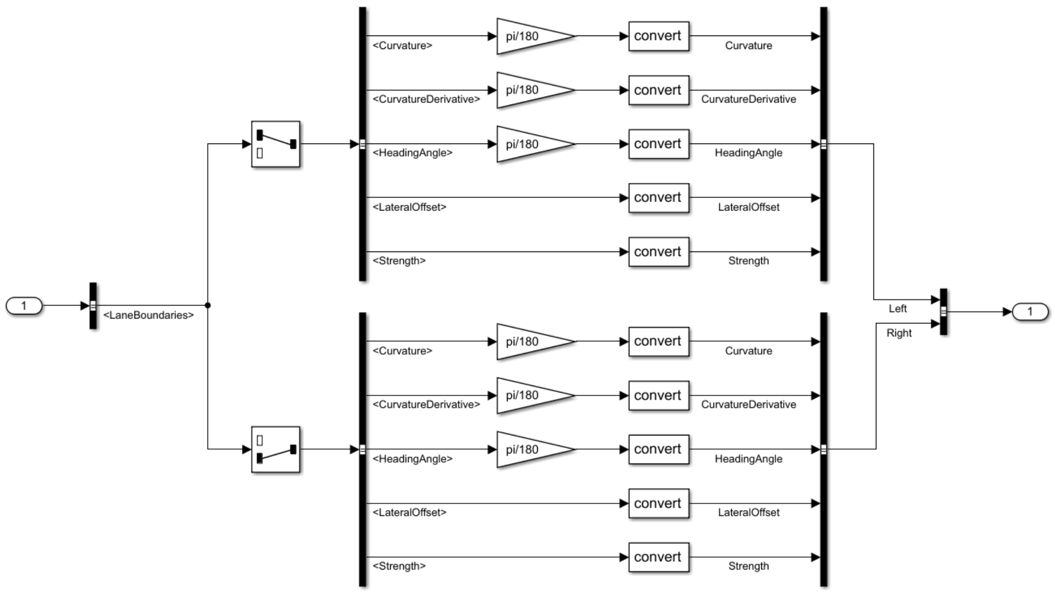


Figure 2.6: Conversion of the Bus containing lane lines information

2.5 Scenario Reader

The `helperScenarioReader` reads actor poses and road data from a recorded driving scenario [7]. The following description is so clear that has been reported as presented in the Matlab help:

> If the actor poses data contains the ego vehicle pose, you can specify which index is used for the ego data. In that case, the block will convert the poses of all the other actors to the ego vehicle coordinates. This allows the actor poses to be used by the visionDetectionGenerator and radarDetectionGenerator objects.
>
> If ego vehicle index is not provided, all the actor poses will be provided in scenario coordinates. You will then have to convert them to ego vehicle coordinates using the driving.scenario.targetsToEgo function before generating detections using the visionDetectionGenerator and radarDetectionGenerator objects.
>
> In addition to reading actor poses data, if you specify an ego actor, you can use this block to read road boundaries data in ego coordinates. Road boundaries in scenario coordinates are obtained using the roadBoundaries method of drivingScenario. They must be saved to the same file as the actor poses using the name RoadBoundaries.

Chapter 3

CARLA simulator

CARLA is an open source photorealistic simulator [8] developed to train, validate and test autonomous driving algorithms. It is written in C++ and its driving scenarios are based on Unreal Engine. It provides digital assets and complete control on actors on the map, environmental conditions control, a sensor suite, maps generation, a flexible API and a server-client based communication. In addition to CARLA Simulator, that embeds all the control logic, the rendering, the physics and all the actor properties, CARLA offers a Python API module. So the server is the Simulator and the client-server communication is controlled through the Python API (see Figure 3.1).

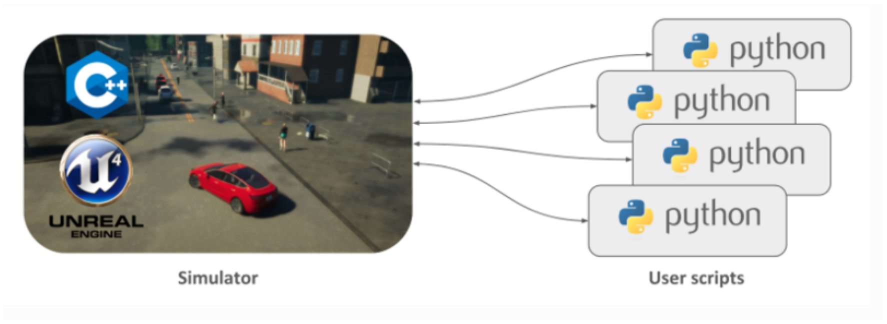


Figure 3.1: CARLA client-server communication [9]

Most of the aspects of the simulation are accessible from the Python API. With Python scripts it is possible to retrieve raw data coming from CARLA sensors attached to the ego vehicle, process them, compute all the parameters needed by the controller and send to CARLA Simulator the controls of throttle, brake and steering.
Each Python script from the client side can be logically divided into two different parts:

- *Configuration of the simulation and of the actors*: before starting the control algorithm, the connection with the server and all the settings of the simulation and the actors has to be done. Here information on the world can be retrieved, actors can be spawned at arbitrary locations and orientations, sensors can be attached to the ego vehicle and other actors can be created, spawned or destroyed.

- *Client-Server synchronized communication*: the control algorithm is contained in this part, where the proper simulation is started. The sensor(s) attached to the vehicle produce raw data and the client subscribe to the sensor stream by providing a callback function that is called each time a new data is generated by the sensor. This callback function is the one that stores the images in a queue accessed by the control algorithm to return the controls to the vehicle. At the end of the simulation the client disconnects.

3.1 Configuration of the simulation and of the actors

All the steps required to configure the simulation and the driving scenario are presented in Figure 3.2. The `Client` class is used to connect to the client and returns an object that can be used to access all the information of the running simulation. By default, there are seven worlds available in CARLA 0.9.5. In this thesis, the 7^{th} has been used because it is a rural environment with different types of streets and lane lines, with different curvatures. This is the best setting to test the algorithm.

Images of the map are visible in Figures 3.3 and 3.4. The goal is to keep the center of the lane with autonomous steering, throttle and brake operations. From the client object the world has been retrieved with `get_world()` and the synchronous mode has been activated. This simulation modality is used to synchronize client-simulator communication. When activated, the simulation is paused at each step until a *tick* message is received. This signal can be sent to the simulator with `world.tick()`. This is very useful when dealing with GPU-based sensors (cameras), that are usually generated with a delay of a couple of frames with respect to data coming from CPU-based sensors (see CARLA documentation [9]).

The newly acquired images produced by the camera can be put in a `Queue` instance, that has been called `image_queue`, by using the `listen` method of the `Sensor` class, that provides a callback function that is called each time a new image is generated. The `get()` method of the image queue is used to retrieve the images at each tick with a first-in first-out logic.

To use the synchronous mode in a consistent and predictable way the Simulator has to run with fixed time-step. Here the step size is 0.1 seconds and is the same value of the sample time of the Nonlinear Model Predictive Controller and between camera captures. The correct way to run the simulation is shown in section 3.3.

Another important part of the configuration interests the actors of the simulation. Actors can be cars, pedestrians, traffic signs and lights, the spectator of the simulation etc. The spectator characteristics can be accessed by calling `world.get_spectator()`. From this instance, it is possible to change the spectator position and orientation, for example to have the desired view of the ego car behaviour.

The actors information objects are instances of the `ActorBlueprint` class. It contains all the tags and attributes of each available actor in CARLA, not only the ones actually alive in the simulation. To access the set of all the blueprints available use the `get_blueprint_library()` method. From this the desired blueprints of the actors can be selected by calling `blueprint_library.find(tag)`, where the tag uniquely identify a blueprint with three words separated by a dot. If we want to select all the actors alive in the simulation that have in common a feature, for example being a vehicle, the command `world.get_actors().filter('vehicle.*.*')` returns a list of actors of type `ActorList` that are all vehicles. This is an iterable object. Note that here a `carla.Actor` is something alive in the simulation, while `carla.ActorBlueprint` contains information about actor that are spawnable but not necessarily alive.

Said that, the blueprints can be modified by calling `set_attribute(key, value)`, for example to change the aspect, and then an actor can be spawned with the `spawn_actor` method. It requires as arguments the blueprint and the `carla.Transform` objects, that contains the information about the desired 3D pose. This procedure can be used for all the actors, also for sensors that has to be attached to the vehicle. It's important to send a tick message each time an actor is spawned or modified in order to avoid errors in code execution. Modifications of the world in the Simulator happens only after the tick.

Then it is suggested to instantiate in this part the object that will be used for the control algorithm execution. Some of its methods, related to the computation of control inputs, will be called inside the main loop, at a frequency determined by the fixed time step size. In this case the class used for the lane keeping control algorithm has been called `LaneKeepingAlgorithm` and will be described later.

CARLA offers the possibility to tune the physical parameters of the wheels and the whole car. The `carla.WheelPhysicsControl` class is used to set wheels parameters and the class `carla.VehiclePhysicsControl` for car parameters, including wheels that are passed as arguments. To apply these parameters to the ego vehicle in simulation use the `apply_physics_control(vehicle_physics_control)` method.

Finally, a queue to store sensors information, in this case images, can be instantiated and `image_queue.put` called each time a new image is available. This thanks to the `listen(image_queue.put)` method of the `carla.Sensor` class, a callback function. At each measurement, the function is called and a camera frame is appended to the queue.

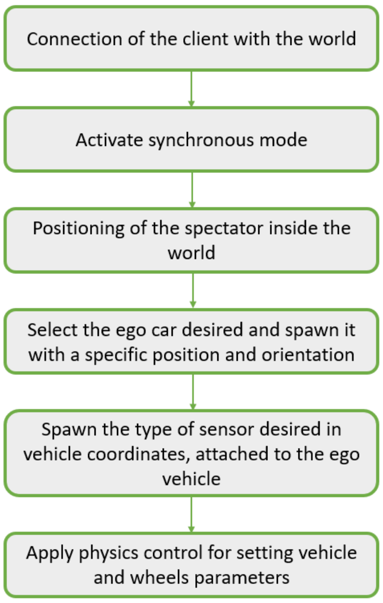

Connection of the client with the world
Activate synchronous mode
Positioning of the spectator inside the world
Select the ego car desired and spawn it with a specific position and orientation
Spawn the type of sensor desired in vehicle coordinates, attached to the ego vehicle
Apply physics control for setting vehicle and wheels parameters

Figure 3.2: Configuration of the simulation and of the actors

Figure 3.3: CARLA Simulator: town 7

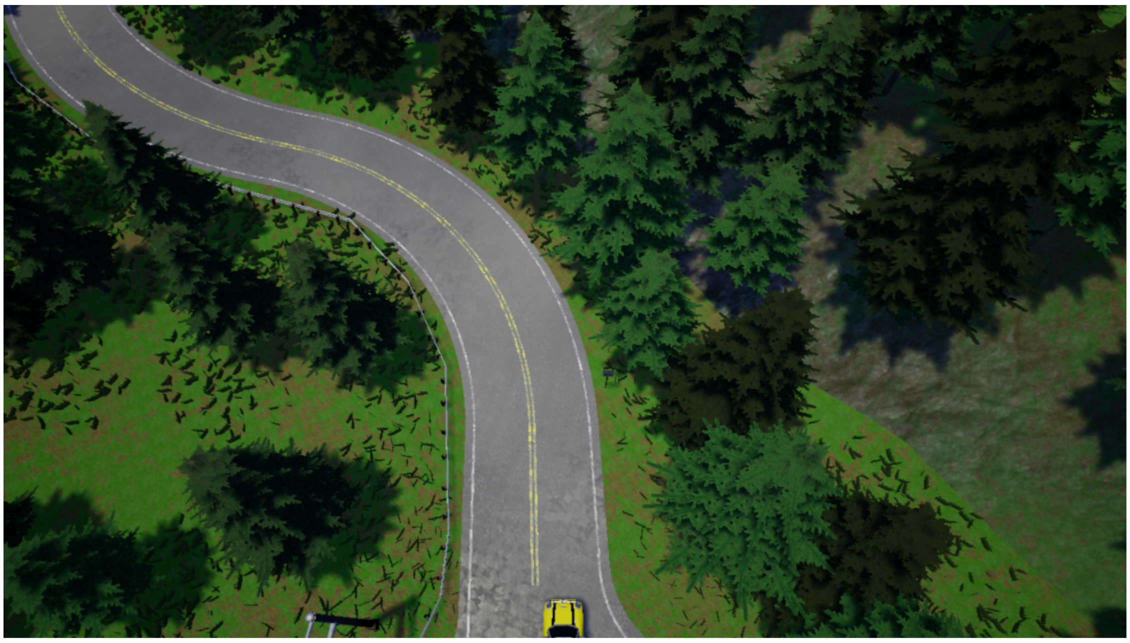


Figure 3.4: Vehicle spawned at start of the track

In the next page is presented the part of the code that configures the simulation and the actors inside it.

```
def main(args):

    actor_list = []
    # Connect to the client
    client = carla.Client('localhost', 2000)
    client.set_timeout(10.0)
    # Load the world for the simulation (default is world 7, if different change
    # also the spawn point of the ego vehicle and its initial pose)
    client.load_world(args.world)
    print(args.world + " loaded")
    # Retrieve the world and the settings to activate synchronous mode
    world = client.get_world()
    settings = world.get_settings()
    settings.synchronous_mode = args.synchronous_mode
    world.apply_settings(settings)
    if settings.synchronous_mode:
        print("Activating synchronous mode")

    # Get the spectator information to set its 3D pose to observe the ego car
    # behaviour
    spectator = world.get_spectator()

    try:
        mymap = world.get_map()
        # Get the world scenario and all the object's blueprints in a list
        blueprint_library = world.get_blueprint_library()

        # Destroy all the vehicles present before starting the simulation
        actors = world.get_actors().filter('vehicle.*.*')
        for actor in actors:
            actor.destroy()

        world.tick()
        world.wait_for_tick()
        # Creating the ego vehicle and positioning it
        ego_blueprint = blueprint_library.find('vehicle.mini.cooperst')
        ego_blueprint.set_attribute('color','255,0,0')
        transform = carla.Transform(carla.Location(x=70, y=-4, z=0.3),
                                    carla.Rotation(yaw=-60))
        ego_vehicle = world.spawn_actor(ego_blueprint, transform)
        world.tick()
        world.wait_for_tick()
        # Set spectator pose
        transform.location.z += 30
        transform.rotation.pitch = -60
        spectator.set_transform(transform)
        world.tick()
        world.wait_for_tick()
        print(ego_vehicle.get_location())
        print ("%s %s %s created and positioned" %(ego_blueprint.tags[0],
                                                   ego_blueprint.tags[1],
                                                   ego_blueprint.tags[2]))
        actor_list.append(ego_vehicle)
        print("Ego vehicle center of mass: ", ego_vehicle.bounding_box.location)
        print("Ego vehicle extention: ", ego_vehicle.bounding_box.extent)
        # Select the type of camera and its features, then spawn it in a specific
        # pose in ego vehicle coordinates and attached to it
        cam_blueprint = blueprint_library.find('sensor.camera.semantic_segmentation')
        cam_blueprint.set_attribute('image_size_x', '800')
        cam_blueprint.set_attribute('image_size_y', '600')
        cam_blueprint.set_attribute('fov', '110')
        cam_blueprint.set_attribute('sensor_tick', '0.1')
        transform = carla.Transform(carla.Location(x=1.1, z=1.4),
                                    carla.Rotation(pitch=-5.0))
        semantic_cam = world.spawn_actor(cam_blueprint, transform,
                                        attach_to=ego_vehicle)
        world.tick()
        world.wait_for_tick()
        print(semantic_cam.get_location())
        actor_list.append(semantic_cam)

        # Istantiate a LaneKeepingAlgorithm object
        mpc_algorithm = LaneKeepingAlgorithm(ego_vehicle, cam_blueprint,
                 semantic_cam, 2, args.save_to_disk)

        # Create front and back Wheels Physics Control
        front_wheel = carla.WheelPhysicsControl(
            tire_friction=4.5,
            damping_rate=1.0,
            steer_angle=degrees(mpc_algorithm.max_steer)
            # Now position cannot be modified in Python but CARLA team
            # is working on give the possibility to change distances
            # of rear and front axles from center of gravity
            # and change the wheelbase and axles length
        )

        back_wheel = carla.WheelPhysicsControl(
            tire_friction=4.5,
            damping_rate=1.0,
            disable_steering=True
            #position=carla.Vector3D(-mpc_algorithm.b, 0.0, 0.0)
        )

        wheels = [front_wheel, front_wheel, back_wheel, back_wheel]

        # Set the physical parameters of the ego vehicle
        ego_vehicle.apply_physics_control(carla.VehiclePhysicsControl(
            use_gear_autobox=True,
            mass=mpc_algorithm.m,
            drag_coefficient=mpc_algorithm.Dc,
            center_of_mass=carla.Vector3D(0.0, 0.0, 0.0),
            wheels=wheels))

        # Subscribe to the sensor stream by providing a callback function,
        # this function is called each time a new image is generated by the
        # sensor. The image queue is useful for storing images in it, waiting
        # to be processed by the mpc_algorithm object
        # by the mpc_algorithm object
        image_queue = queue.Queue()
        semantic_cam.listen(image_queue.put)
        world.tick()
        world.wait_for_tick()
        frame = None

        # HERE THE MAIN LOOP

    except KeyboardInterrupt:
        print('\nExit by user.')

    finally:
        if args.synchronous_mode:
            print('Disabling synchronous mode.')
            settings = world.get_settings()
            settings.synchronous_mode = False
            world.apply_settings(settings)


        print('destroying actors.')
        for actor in actor_list:
            actor.destroy()

        print('done.')
```

3.2 Server-Client communication

After the configuration, the control algorithm can be activated to put in motion and control the ego vehicle in the street shown in Figure 3.4. Fundamental in this part is the correct synchronization between the image frames coming from the camera and the timestamp's frame produced by the tick function. Some control loops suggested by CARLA team on GitHub has been implemented to detect frame skip or wrong image timestamp and wait the correct matching. In this part the main cycle, that repeats itself at each tick received by the simulator, does what is shown in Figure 3.5. The image from the queue is get by using the `get()` method.

There are two possible ways to compute the parameters needed by the controller. One is based on the top view image and computes the lateral error by computing the pixels between the center of the car and the center of the lane in the image. Then it converts this number to meters by knowing the lane width. The yaw error is computed from the slope of the near view section lines w.r.t. the vertical, that is the direction of the camera and of the car. This measure is then converted in radians.

Another way is based on `carla.Waypoint`, that is a class that offers the possibility to access meaningful points of the map. They can be exploited for trajectory planning and to retrieve the coordinates and directions of points in the center of the lane. By comparing this information with the position and orientation of the car inside the lane, it is possible to compute lateral error and yaw error for the controller.

After having computed the control inputs, they are sent to the ego vehicle by instantiating a `carla.VehicleControl` object that contains as arguments steer, throttle and brake. This object is then passed to the `apply_control` method of the Vehicle object.

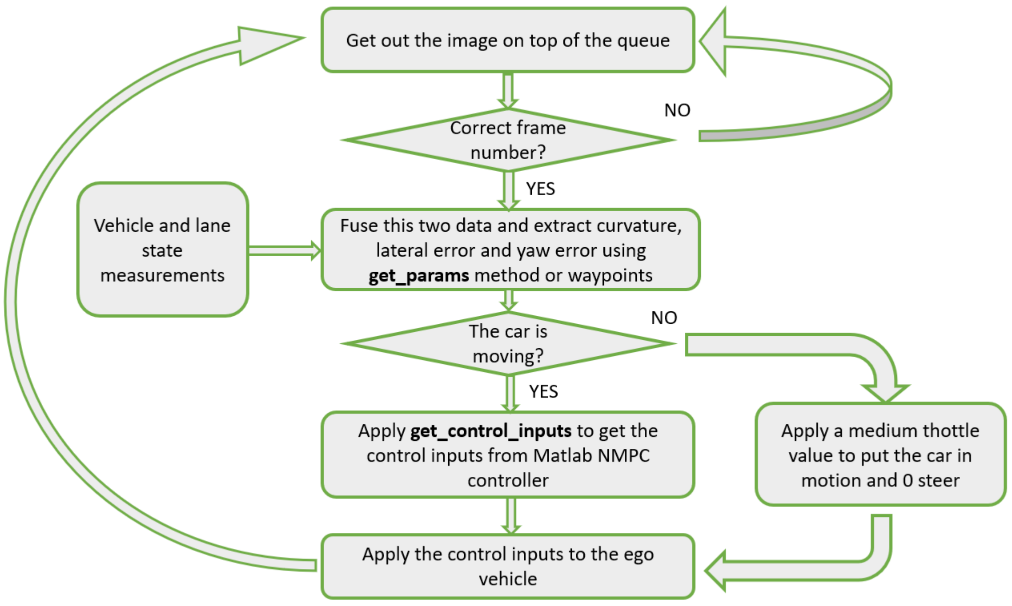

Get out the image on top of the queue
Correct frame number?
NO
YES
Vehicle and lane state measurements
Fuse this two data and extract curvature, lateral error and yaw error using **get_params** method or waypoints
The car is moving?
NO
YES
Apply **get_control_inputs** to get the control inputs from Matlab NMPC controller
Apply a medium thottle value to put the car in motion and 0 steer
Apply the control inputs to the ego vehicle

Figure 3.5: Main loop and server-client communication

Follows in the next page the code of the main loop.

```python
while True:
    # Get vehicle location and its nearest waypoint
    ego_location = ego_vehicle.get_location()
    vechicle_waypoint = mymap.get_waypoint(ego_location)

    # As before, to synchronize the Server-Client communication a tick is sent
    # to the Server (the world) and the Client waits the response before letting
    # the python interpreter going on.
    # To have perfect synchronization, the frame number of the image has to be
    # equal to the frame count of the tick
    world.tick()
    ts = world.wait_for_tick()

    if frame is not None:
        if ts.frame_count != frame + 1:
            logging.warning('frame skip!')

    frame = ts.frame_count

    # It checks if the image frame number from the queue is equal to the
    # tick frame number end exits from the loop only in this case
    while True:
        image = image_queue.get()
        if image.frame_number == ts.frame_count:
            break
        logging.warning(
            'wrong image time-stampstamp: frame=%d, image.frame=%d',
            ts.frame_count,
            image.frame_number)

    lane_width = vechicle_waypoint.lane_width

    right_lane_waypoint = vechicle_waypoint.get_right_lane()
    if right_lane_waypoint:
        # It computes the vehicle lane level localization and lane curvature needed
        # by the controller with the get_params method of the LaneKeepingAlgorithm
        curvature, lateral_error, yaw_error = mpc_algorithm.get_params(image,
                                                lane_width)

        # Other way to compute lateral and yaw error, by using carla.Waypoint class
        e1 = sqrt((right_lane_waypoint.transform.location.x -
            ego_location.x)**2 +
            (right_lane_waypoint.transform.location.y - ego_location.y)**2) -
            lane_width/2
        e2 = radians(right_lane_waypoint.transform.rotation.yaw -
            ego_vehicle.get_transform().rotation.yaw)
        print("e1= %.2f, e2= %.2f" %(e1, e2))

        v = ego_vehicle.get_velocity().x
        # Computes the control inputs for the ego car by using the
        # get_control_inputs method, that connects to Matlab to
        # exploit the designed NMPC controller
        steer, throttle, brake = mpc_algorithm.get_control_inputs(curvature,
                                                                  e1, e2, v)

    else:
        steer = 0.0
        throttle = 0.4
        brake = 0.0

    print("INPUTS: steer={0:.3f} throttle={1:.3f} brake={2:.3f}".format(
        steer, throttle, brake))

    # Apply control values to the ego vehicle
    ego_vehicle.apply_control(carla.VehicleControl(
        steer=steer,
        throttle=throttle,
        brake=brake))
```

3.3 Running the simulation

To run this simulation are needed:

- Matlab & Simulink with Control, Model Predictive Control, Vehicle Dynamics Blockset and Automated Driving toolboxes.
- Windows or Linux x64 operating systems with Python 3.7 and the following python packages: OpenCV, Numpy, Matplotlib, PIL, Imageio, matlab and matlab.engine (see "Calling MATLAB from Python" in the Matlab help).
- CARLA 0.9.5 binary or compiled version. The binary version is ready to use and contains the Unreal Engine's components needed for maps, blueprints, actors etc. It occupies less space in hard disk but obviously doesn't allow modifications of the Python API neither of the C++ simulator's code. Moreover, the precompiled version doesn't allow to build new maps but only to use the pre-existing ones. Build a map with RoadRunner is very easy and they can be imported in Unreal Engine directly to save them properly. Then it is sufficient to compile from source to have the newly created map available. RoadRunner is the software used by the creators of CARLA for its simplicity compared to Unreal Engine. To use it with a long-term academic license at no cost is sufficient to contact them as a professor of an academic institution.

Follow these steps to run the simulation with the compiled version:

1. Open Matlab and run the script `ckla_carla` to set the workspace for the Python script. The Matlab script contains all the parameters of the controller, the vehicle and the simulation. The tuning of these parameters have to be done directly here. The python script will retrieve these data from workspace when launched. Matlab creates the controller and Kalman filter structures needed.

2. Run the command `matlab.engine.shareEngine` to make the Matlab workspace shareable with a python script.

3. Run `CARLAUE4` executable (.exe for Windows, .sh for Linux) from the terminal with the following string to run the simulator with fixed-time step mode:

```
E:\CARLA_bin>CarlaUE4.exe -benchmark -fps=10
```

4. Run the Python client named `NMPC.py` without passing any argument. The default ones are: –synchronous-mode(-s): True; –save-to-disk(-d): True; –fps(-f): 10; –world(-w): "Town07". In this way the simulation runs at 10 frames-per-second in synchronous mode, on map Town07 and saves the input and processed images to disk inside the "`PythonAPI\examples\top_view`" folder:

```
E:\CARLA_bin\PythonAPI\examples>python NMPC.py
```

5. Look at `CARLAUE4` to watch the car moving on the photorealistic world, to "`top_view`" folder for the top-view image of the lanes and the near and far section lines identification, to the DOS terminal of the client for the information about the actor, the world and the input commands.

Chapter 4

Computer vision

In Autonomous Driving, vehicle localization with respect to lane coordinates is fundamental for lane keeping and obstacle avoidance. The vehicle localization can be addressed by exploiting different technologies and fuse their data for getting more precision. GPS is very effective for geolocalization of the car and is heavily used by global motion planning algorithms to choose the best route on a digital street map. In this case the goal is achieved using a number of different sensors: Global Navigation Satellite System, Inertial Measurement Unit and odometry.

The localization of the ego car w.r.t. the other actors in its proximity can be accurately achieved by the LIDAR technology. CARLA offers natively LIDAR sensors. It produces a map of digital points that are produced by a laser emitter that sends pulses at a specified frequency in all the directions. This source of pulses spins to have a 360 degree and 3D "vision" of the neighbourhood of the car, at a sufficiently high angular speed have near enough samples of the surroundings. However, it is an expensive technology for its intrinsic cost and for the high performing hardware that needs in the background to crunch this huge amount of data. In fact, the points of the LIDAR maps aren't meaningful by themselves but require multiple steps for extract useful informations, as: segmentation & outlier elimination, feature extraction and classification. Usually these steps require more or less complex computer vision techniques for the first 2 steps and artificial intelligence for the last (e.g. Convolutional Neural Networks).

Among the various types of sensors used in autonomous driving, the camera is for sure one of the most important. In fact, the real time images captured by the camera can be processed by algorithms in order to extract useful features from the data and use this features in the control strategy. In autonomous vehicles the vision system normally involves road detection and on-road object detection. The first includes two subcategories: lane line marking detection and road surface detection [10]. For lane keeping purposes, in this work most of the attention has been devoted to lane line marking detection with a single camera outside the vehicle, positioned on its longitudinal axis.

4.1 Camera models and characteristics

Images can be considered as 2D discrete functions of intensity values. In this work has been extensively used the RGB representation of an image, that is a three channels representation (red-green-blue) with a matrix of pixels for each channel. The color of each pixel is determined by the value on the respective entry on each channel. We have to keep in mind that an image is a 2D projection of the real world, representable by 3D Cartesian space of spatial coordinates. The medium that does this projection is the camera. It is mounted on the vehicle near the view mirror or on the front, it is positioned on the longitudinal axis of symmetry and it is slightly inclined towards the road surface. This is assumed flat and implies that there is a projective relationship between the reference frame on the image plane and the corresponding reference frame on the ground plane. In general the relationship between vectors of coordinates of the two reference frames is:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = H \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{4.1}$$

where H is a homography that can be recovered through calibration. This equation holds for cameras with a wide field of view (110 in our case) and for areas not too far from the vehicle. By using CARLA virtual cameras we don't need to take into consideration the distortion produced by the camera model because it is absent.

4.2 Lane Line Marking Detection

High precision lane/vehicle localization is of fundamental importance in the Autonomous Driving field. This work addresses lane following and identifies mathematically lane lines is the first step to estimate the pose of the vehicle w.r.t. the lanes and feed the control system with data about lateral offset, yaw error and previewed curvature. Here a purely vision system is proposed. It exploits Hough transform and least-squares for second order polynomial fitting. A Kalman Filter should be used to improve continuity of the solution. However, it will be shown that this algorithm is very sensitive to noise and road uncertainties. To work properly it needs both lane lines to be present. Otherwise it is not able to steer correctly the car. The control algorithm proposed bases its predictions and control solutions entirely on lane lines identification. Other researchers have addressed this problem by using stereovision and particle filters [11], integrating lane and vehicle detections [12]. Most computer vision algorithm for lane detection can be subdivided in the following parts [11] (see Figure 4.1):

- *Lane line feature extraction*: lane sensing is required to identify the pixels that belong to lane line markings and eliminate non-lane marking pixels. Image descriptors present in the literature include adaptive and global threshold [13, 14] (e.g. adaptive Canny), steerable filters [5], edge detection [15] and top-hat filters [14]. An algorithm to implement this part is presented in section 4.2.1. It adopts a linear model by using Hough transform. CARLA Simulator contains a useful feature to get a meaningful representation of the objects present in the virtual world. By using a *semantic* *segmentation camera* the images received by it contain classified objects that are displayed in a different color according to the object class. Each BGRA image has the tag encoded in the red channel. So this feature will be used in the algorithm written in Python to select pixels that represent lines and put them to 255 value and put all the others to 0.

- *Top-view image transformation*: this perspective transformation is useful to eliminate the perspective distortion from the image. The car direction and orientation can be estimated better because applying model fitting to the top-view image is easier, although disturbances on the camera position and the flatness of the road can have a significant influence on the transformation.
- *Model fitting*: It is the process to extract a mathematical representation of the lane from the previous step. Here linear and parabolic models has been used. Obviously these models are good approximations of the real lines, that have the advantage of being computationally and memory efficient. Real lines are better described by clothoid models and splines. Rural and urban roads may contain various discontinuities, which can require more sophisticated road modelling.
- *Time integration*: is usually applied to make use of the previous information to guide the search in the current image. Most of the approaches in scientific papers are stochastic. Mostly, Kalman and particle filters are used. The vehicle dynamic model can be taken into account to improve estimation. The vehicle pose can be derived from the fitted model. It can be also used to guide the lane line detection in the next frame to improve continuity. Kalman filters tend to work well for continuous structured roads, as the one considered here. In this work has not been implemented.

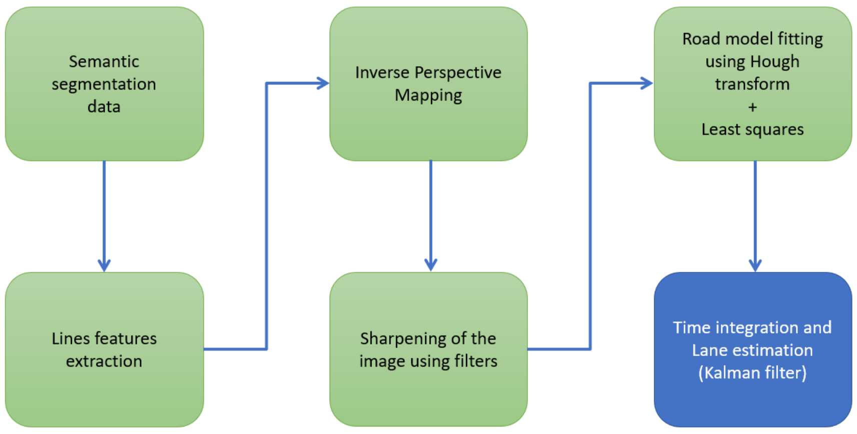

Semantic segmentation data
Lines features extraction
Inverse Perspective Mapping
Sharpening of the image using filters
Road model fitting using Hough transform + Least squares
Time integration and Lane estimation (Kalman filter)

Figure 4.1: Lane line marking detection algorithm pipeline

The steps until the 4^{th} are coded inside the `LaneEstimator` class, that is instantiated by the `LaneKeepingAlgorithm` constructor. Its public method, called `process`, receives as input a semantic segmentation coded image and returns the warped top-view image. This is ready for the road model fitting, done by the RoadModel class. The time integration step has been left out but should be added in future works to give robustness to this algorithm. Below is shown the constructor and the process method, while the specific methods for lane detection and top-view transform are addressed in subsections 4.2.2 and 4.2.3. Some of the object variables in the constructor will be addressed in subsection 4.2.3.

```
class LaneEstimator(object):
    def __init__(self, camera_info, camera_actor, poly_order,
                save_to_disk=False):
        # Camera characteristics
        self._camera_info = camera_info
        pose = camera_actor.get_transform()
        self._H = float(pose.location.z) # Height
        self._pitch = radians(float(pose.rotation.pitch)) # Pitch angle
        # Horizontal field of view
        self._h_fov = radians(float(camera_info.get_attribute('fov')))
        # Horizontal image size
        self._V = int(camera_info.get_attribute('image_size_x'))
        # Vertical image size
        self._U = int(camera_info.get_attribute('image_size_y'))
        # Vertical field of view (derived from other camera info)
        self._v_fov = 2*atan(float(self._U / self._V) * tan(self._h_fov / 2))
        # Camera tilt angle (derived)
        self._tilt = pi / 2 + self._pitch - self._v_fov / 2
        self.__str__() # print camera characteristics
        # rectangle of the image to be considered, upper-left and lower-right points
        self.box = (280, 130, 510, 750)
        # Width and height values choosen to crop the image to improve efficiency
        self._width_crop = self.box[3] - self.box[1]
        self._height_crop = self.box[2] - self.box[0]

        # These source-points have been found to be optimal for the camera
        # pose and parameters we're dealing with. This resulted in a better
        # top-view transform than the set of functions reported in the section
        # named "Top-view transform"
        self._source_points = np.array([[150, 0],
                                        [self._width_crop - 300 - 1, 0],
                                        [500, self._height_crop - 1],
                                        [300, self._height_crop - 1]],
                                        dtype = "float32")

        # Order of the polynomial fitting (the default value is 2)
        self.n = poly_order
        # Stride and skip values are used to identify lane lines pixels on the
        # top-view image in function compute_poly_params
        self.stride = 2
        self.skip = 5
        self.l = np.empty(6)
        # Kernel of the filter adopted to sharpen the image
        self.kernel = np.array([[-1,-1,-1],
                                [-1, 9,-1],
                                [-1,-1,-1]])
        self.save_to_disk = save_to_disk

    def _detect_lanes(self, image):
        # Addressed in "Lane line feature extraction with CARLA" section

    def process(self, image):
        sharp_image = self._detect_lanes(image)
        warped_image = four_point_transform(sharp_image, self._source_points)
        if self.save_to_disk:
            imageio.imwrite('top_view/lanes.png', warped_image)
        return warped_image

    def __str__(self):
        print('*' * 60)
        print ("%s %s %s created and positioned" %(self._camera_info.tags[0],
            self._camera_info.tags[1], self._camera_info.tags[2]))
        print("-----PARAMETERS-----")
        print("Height: %.2f" %self._H)
        print("Pitch angle: %.2f" %degrees(self._pitch))
        print("Tilt angle: %.2f" %degrees(self._tilt))
        print("Image size: %d x %d" %(self._V, self._U))
        print("Horizontal field of view: ", round_up(degrees(self._h_fov)))
        print("Vertical field of view: ", round_up(degrees(self._v_fov)))
        print('*' * 60)
```

4.2.1 Lane line feature extraction with Hough transform

In this subsection, an algorithm for detecting lanes from camera recordings is shown. This algorithm is similar to the one presented in a web-course on self driving cars[1]. Data are preprocessed and then used to supply to the control system with vehicle location and response signals to calculate appropriate steering and acceleration/deceleration. An useful tool for computer vision in general is OpenCV, an open source library extensively used all around the world for this kind of applications. It allows image processing and feature extraction. It has been used to identify lane parameters and the pose of the car with respect to the lanes. Typically, the operations needed to pre-process data and identify lane lines are the following [16]:

1. Acquire an image
2. Digitize image
3. Detect edge (Sobel filter, Canny methods)
4. Thresholding (generate a binary image)

[1] **https://eu.udacity.com/course/self-driving-car-engineer-nanodegree–nd013**.

5. Noise cleaning
6. Hough transform
7. Identify lane-marker candidates
8. Decide the lane markings
9. Apply least squares or Kalman Filter techniques to identify line parameters

Here edge detection is used in order to find regions in an image where there are sharp changes in intensity and in color. In order to recognize the edges in a mathematical way, the gradient is applied to the image, seen as a matrix of pixels, thanks to a digitalization of spatial coordinates (i.e. image sampling). Each pixel mapped in the matrix contains the light intensity at some location in the image, denoted by a numeric value that ranges from 0 to 255 (i.e. 0 means black and 255 means white). This is the so called intensity or grey level quantization. So the gradient is a measure of the change in brightness over adjacent pixels. The function to which the gradient is applied is the pixels intensity function in the two spatial variables x and y:

$$\nabla f(x,y) = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix} \tag{4.2}$$

where G_x and G_y are the so called Sobel operators, a robust implementation of gradient discretization. In the simplified case of 3x3 kernels, they are equal to:

$$\begin{aligned} G_x =& [f(x+1,y-1) + 2f(x+1,y) + f(x+1,y+1)] - \\ & [f(x-1,y-1) + 2f(x-1,y) + f(x-1,y+1)] \\ G_y =& [f(x-1,y+1) + 2f(x,y+1) + f(x+1,y+1)] - \\ & [f(x-1,y-1) + 2f(x,y-1) + f(x+1,y-1)] \end{aligned}$$

If there is a strong change between two adjacent pixels the gradient will be high and the other way around. The algorithm shown here can detect edges as rapid changes in brightness, so edges correspond to pixels where the gradient is higher than a fixed threshold. To do this operation easily, the image has been converted to grayscale. Numpy, a python library for scientific computing, has been used for dealing with data structures and pixels' brightness intensities. Then noise has been filtered out with a Gaussian filter with a 5x5 kernel (a good size for most cases) to avoid false edges that ultimately affect edge detection. The Gaussian filter is a low-pass filter because it removes the high-frequency components of the image to which is convoluted. An equivalent way to explain this is that rapid changes in brightness are filtered out and by consequence the image is blurred. The kernel is an approximation of a 2D Gaussian function in the discrete 2D space of the image pixels. The classical formula for a 2D Gaussian distribution is:

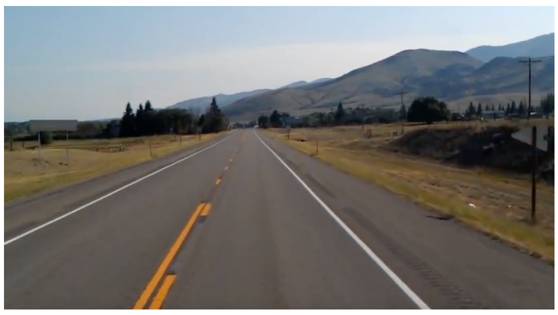
$$h(u,v) = \frac{1}{2\pi\sigma^2} exp\left(-\frac{u^2+v^2}{\sigma^2}\right) \tag{4.3}$$

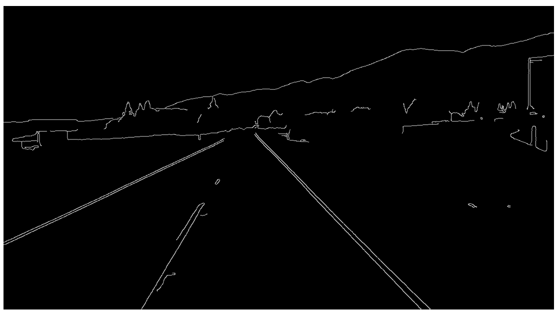
and the equivalent kernel representation in case of dimension 5 is:

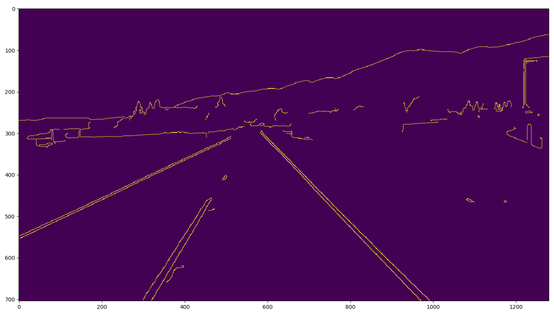
$$H[u,v] = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{4.4}$$

The Gaussian kernel is symmetric, all its values are symmetric and sum to 1. The amount of smoothing is proportional to the mask size.

For edge detection, the Canny method is used, as it is one of the most robust methods while it allows a reduction of the amount of data to be processed. It uses double thresholding to determine potential edges, which lead to the definition of strong and weak edges. If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed. A weak edge is considered only if it is connected to a strong edge. In the code below the low threshold is equal to 50 and the high threshold is equal to 150. The `canny` function does the grayscale conversion, applies the Gaussian filter and the Canny method. In Figure 4.2b is reported the output of this function. The image clearly traces an outline of the edges that correspond to the most sharp changes in pixel's intensity, where gradients exceed the high threshold.


(a) Original


(b) Canny



(c) Vertices

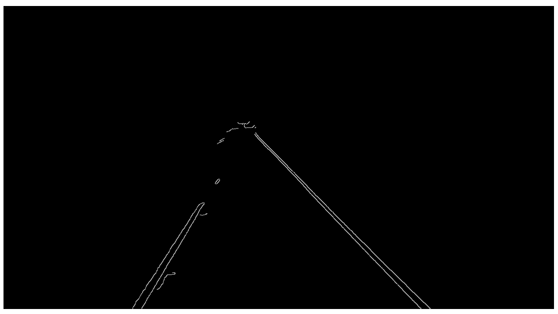

(d) ROI

Figure 4.2: Canny method and ROI isolation

In Figure 4.2b it's now possible to define a region of interest (called *ROI*) which corresponds to the lanes that the car has to track. In order to isolate this region of interest, the Matplotlib library has been used to plot the image with (x, y) coordinates. The ROI has been considered a triangle mask that has, as vertices, points of coordinates (200,700) – (1100,700) – (550,250). The mask has been computed through the `region_of_interest` function. The `zeros_like` function gives as output an image with the same number of pixels of the input image but with all zero values (i.e. all pixels are black). Then the function `fillPoly` fills the black picture with the triangle specified with an array of its vertices and sets the value of the pixels inside the triangle to to 255 (i.e. white). Then, with a `bitwise_and` operation, the result reported in figure 4.5c has been obtained.

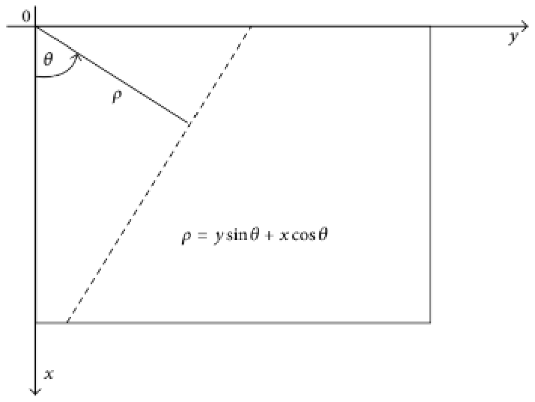


Figure 4.3: Polar coordinates for Hough transform [15]

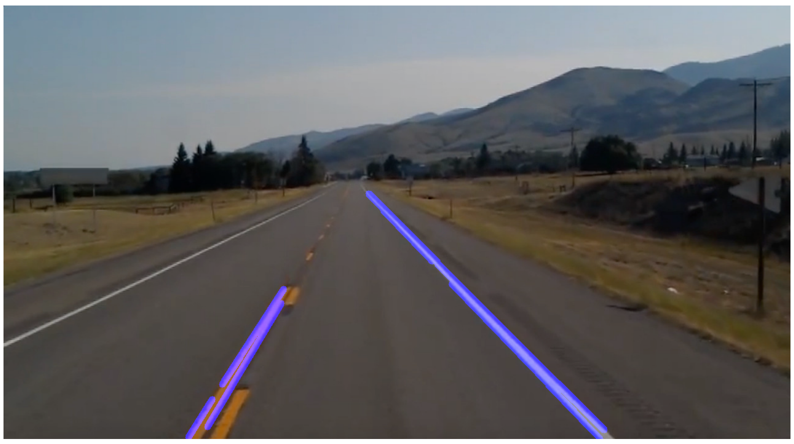


Figure 4.4: Lane identification

After that, the Hough transform is used to detect straight lines. With this tool, a single line in a Cartesian space of equation $y = mx + b$ can be represented as a single point of coordinates (m, b) in the Hough space. Taking the coordinates of a segment's starting point (x_1, y_1) and end point (x_2, y_2) the formulas for the map from Cartesian to Hough space can be easily obtained:

$$b = \frac{(y_2 - y_1)}{(x_2 - x_1)}, \qquad a = y_1 - \frac{(y_2 - y_1)}{(x_2 - x_1)} x_1 \tag{4.5}$$

A single point in the Cartesian space is mapped as a line in the Hough space. So it is possible to get the line that pass through some points in the Cartesian space by looking at the intersection of the corresponding lines in the Hough space. This point has coordinates m and b that uniquely identify a line in the Cartesian space. Usually, points in this space are not perfectly aligned, so a method to decide if some points have to be considered part of a single line or not is needed. To this aim, the Hough space is discretized in *accumulator cells*, defining a matrix with null initial entries. For each cell with unique coordinates (m, b), a vote equal to the number of intersections inside is casted. The cell with the highest score is the one corresponding to the parameters m and b of the best fitting line. A problem of using Cartesian coordinates in Hough transform is that cannot deal with vertical lines, because m tends to infinity. The solution is using polar coordinates (ρ, θ) to represent lines, with the difference that the set of lines in the Cartesian plane corresponds to a sinusoid in the Hough space (see Figure 4.3). In the code, the function `HoughLinesP` has been used, giving as arguments the image, the resolution of accumulator cells in pixels with one degree of precision in radians, the threshold (i.e. minimum number of votes needed to accept a candidate line), a place-holder, the minimum length for a line to be accepted and the maximum distance in pixels between segmented lines which we will allow to be connected. It implements a probabilistic Hough Transform, that is an optimized version of the Hough Transform we saw. It takes into consideration only a random subset of points and that is sufficient for line detection. The function `display_lines` takes an image and the lines detected through the Hough transform and display them. Then, the function `addWeighted` blends in a weighted way the original image with the detected lines. In this way, lane-marker candidates has been identified as the lines which best describe data, visible in Figure 4.4.

As last step, a further optimization on displaying lane-markers is suggested. The function `make_points` computes the end-points of both right and left lane lines at fixed vertical coordinates v, from the bottom of the image to 3/5 of the height of the image. In this way, at each iteration we get the same length of the identifies lines.

```python
# Title : Find lane lines
# Author : Udacity
# URL : https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013
# Description : Simple algorithm used to compute the parameters of lane lines
#               with Hough transform and 1st order fitting from RGB camera images


import cv2
import numpy as np


def make_points(image, line):
    """
    It computes the end points of the lines from the values of slope
    and intercept. The y values are at the bottom and at 3/5 of the
    image, that is a value chosen as the end of the near section of
    the image.
    """
    slope, intercept = line
    y1 = int(image.shape[0])     # bottom of the image
    y2 = int(y1*3/5)             # slightly lower than the middle
    x1 = int((y1 - intercept)/slope)
    x2 = int((y2 - intercept)/slope)
    return [[x1, y1, x2, y2]]

def average_slope_intercept(image, lines):
    """
    1st order fitting of the line's points and separation in left
    line and right line based on the intercept sign. Then it averages
    the slope and intercept values of both the lines and computes the
    start and end points for both.
    """
    left_fit    = []
    right_fit   = []
    if lines is None:
        return None
    for line in lines:
        for x1, y1, x2, y2 in line:
            # Polyfit computes the 1st order fitting of the lane points
            fit = np.polyfit((x1,x2), (y1,y2), 1)
            slope = fit[0]
            intercept = fit[1]
            if slope < 0: # y is reversed in image
                left_fit.append((slope, intercept))
            else:
                right_fit.append((slope, intercept))

    # add more weight to longer lines
    left_fit_average  = np.average(left_fit, axis=0)
    right_fit_average = np.average(right_fit, axis=0)
    left_line  = make_points(image, left_fit_average)
    right_line = make_points(image, right_fit_average)
    averaged_lines = [left_line, right_line]
    return averaged_lines

def canny(img):
    """
    - It does the RGB to GRAY conversion (from 3 channels to 1)
    - It does the convolution of the image with a 5x5 gaussian
      kernel to smooth the image and remove the high frequency components
    - It applies the Canny algorithm with 50 and 150 as thresholds
    """
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    kernel = 5
    # The standard deviation is passed as 0, so it is automatically
    # calculated from the kernel size
    blur = cv2.GaussianBlur(gray, (kernel,kernel), 0)
    canny = cv2.Canny(gray, 50, 150)
    return canny

def display_lines(img,lines):
    """
    It brings to 0 value all the pixels and draws the left and right
    lines from the end points. The thickness is 10 pixels
    """
    line_image = np.zeros_like(img)
    if lines is not None:
        for line in lines:
            for x1, y1, x2, y2 in line:
                cv2.line(line_image, (x1,y1), (x2,y2), (255,0,0), 10)
    return line_image

def region_of_interest(canny):
    """
    It produces a mask of 0 values of the same shape of the canny image
    with a triangle inside that is our region of interest, that is the
    region where we want to find and consider the lane lines. It fills
    the triangle with 255 pixels and it does a bitwise and logic operation
    to reveal the lane lines inside the triangle coordinates.
    Then it returns the masked image.
    """
    height = canny.shape[0]
    width = canny.shape[1]
    mask = np.zeros_like(canny)

    triangle = np.array([[
    (200, height),
    (550, 250),
    (1100, height),]], np.int32)

    cv2.fillPoly(mask, triangle, 255)
    masked_image = cv2.bitwise_and(canny, mask)
    return masked_image


cap = cv2.VideoCapture("test2.mp4")
# It cycles each frame of the video until the end, then it exits
while(cap.isOpened()):
    _, frame = cap.read()
    canny_image = canny(frame)
    cropped_canny = region_of_interest(canny_image)
    lines = cv2.HoughLinesP(cropped_canny, 2, np.pi/180, 100, np.array([]),
        minLineLength=40, maxLineGap=5)
    averaged_lines = average_slope_intercept(frame, lines)
    line_image = display_lines(frame, averaged_lines)
    # It blends the original image with the one that comes from the image
    # processing algorithm with a weight coefficient of 0.8
    combo_image = cv2.addWeighted(frame, 0.8, line_image, 1, 1)
    cv2.imshow("result", combo_image)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Some of the features present in this algorithm will be used in the Python script that communicates with CARLA Simulator, but in that case the fitting will be done once the Inverse Perspective Mapping is applied and the top view transform is subdivided in near view and far view image. The first part will be fitted using a similar procedure with Hough transform while in the second a least-squares 2nd order fitting is applied.

4.2.2 Lane line feature extraction with CARLA

In CARLA Simulator this procedure can be done automatically by using a semantic segmentation camera. It returns a `carla.Image` object that contains sensor data attributes and the raw data, that contain an array of BGRA 32-bit pixels with the tag information encoded in the red channel. The function `save_to_disk_sem` has been written to save converted semantic segmentation images to disk. To highlight the pixels that represent lane lines the function `detect_lanes` reshape the image array representation to get the ARGB equivalent and selects the red channel, that is the channel that encodes the object tag. The pixels corresponding to the tag associated to road lines are put at maximum brightness while the other to 0 value. Then the image is cropped to get the region of interest (ROI) and sharpened (see Figure 4.5). Follows the python script that implements the already described functionality.

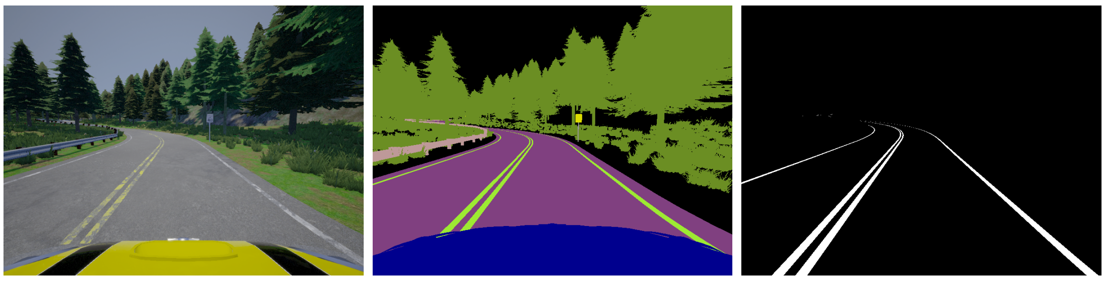

(a) RGB image (b) Semantic segmentation (c) Line's pixels highlighted

Figure 4.5: Camera image processing

```
def _detect_lanes(self, image):
    """
    This function highlights in white the lanes in a semantic
    segmentation image coming from a CARLA sensor and put every other
    pixel to 0 (black). Originally it is an BGRA. Note that the CARLA
    server encodes the tag information in the red channel and the tag
    for lane lines is 6
    """
    array = np.array(image.raw_data, copy=True, dtype='uint8')
    # copy of the sensor data to be manipulated
    array = np.reshape(array, (image.height, image.width, 4)).T
    # reshaping in ARGB
    array = array[2, :, :].T # I get only the second channel where
                             # the tag is, the RED one

    # I put the pixels referred to lanes to 255 (white) and the other to 0
    for pixel in np.nditer(array, op_flags=['readwrite']):
        if pixel[...] == 6:
            pixel[...] = 255
        else:
            pixel[...] = 0
    #pixel coordinates for cropping the image in the area I'm interested in
    array[360:, 398:402] = 255
    cropped_image = array[self.box[0]:self.box[2], self.box[1]:self.box[3]]
    sharpened = cv2.filter2D(cropped_image, -1, self.kernel)
    return sharpened.astype(float)
```

4.2.3 Top-view transform

Top-view image transformation is a very effective tool for lane detection and fitting. It removes the perspective effect and makes easier to compute lane parameters and vehicle's lateral offset and yaw error. This mapping is dependent of the pose of the camera w.r.t. the ego vehicle and of the street characteristics. An important assumption that is usually made is to consider fixed camera pitch angle and height, plus the assumption of flat road. These kinds of disturbances can affect drastically the transform and aren't considered here. With these assumptions, after the top view image transformation the shape of the lane on the image becomes almost the same as the real road lane with a minimal distortion. Here, the inverse perspective transform is used for lane keeping purposes, so it's really important to get a lane representation that presents lane lines as parallel when are straight. The methods, the equations and the images for the transform that are presented here have been taken from [15]. The general idea is to map pixels from the front-view image to the top-view image and is shown in Figures 4.6 and 4.7.

In Figure 4.7 are shown the coordinates needed for the transform, that map pixels of coordinates (U_i, V_i) in the front-view image to pixels of coordinates (X_i, Y_i) in the top-view image. Here θ_v is the vertical field of view, abbreviated as *vfov*, θ_h is the horizontal field of view (*hfov*), H is the height of the camera from the ground, α is the tilt angle, L_0 the horizontal distance between the camera and the first observable pixels from below and L_i the horizontal distance between the camera and pixel P_i we're actually considering. The

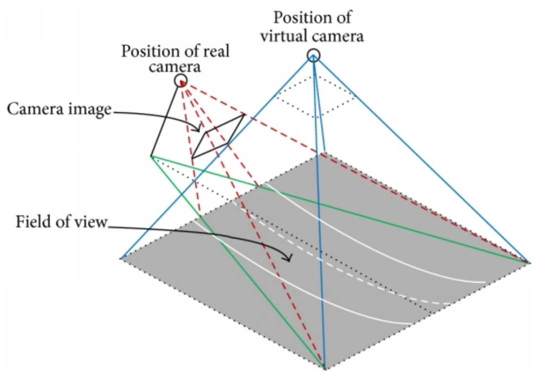


Figure 4.6: Top-view image transformation: general idea [15]

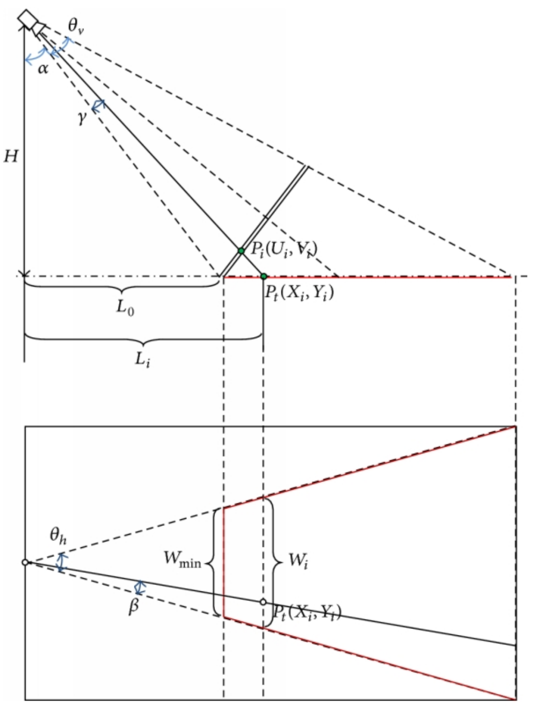


Figure 4.7: Top-view image transformation: linear and angular coordinates [15]

same reasoning can be applied to W_{min} and W_i. The angles γ and β depend on the pixel we are considering. So at each pixel corresponds a set of parameters $\{L_i, W_i, \gamma, \beta\}$. Here are reported the various equations that can be used to implement the following relation:

$$P_i(U_i, V_i) \longrightarrow P_t(X_i, Y_i) \qquad \forall i \in \text{Image} \tag{4.6}$$

That is,

$$L_{min} = H \tan\alpha$$
$$W_{min} = 2L_{min} \tan\left(\frac{\theta_h}{2}\right)$$
$$K = \frac{V}{W_{min}}$$

The height of the camera located in pixel data H_{pixel} is computed by $H_{pixel} = H \cdot K$ and the inverse perspective mapping transformation equations can be computed with the following equations:

$$\gamma = \theta_v \left(\frac{U - U_i}{U}\right)$$
$$L_i = H_{pixel} \tan(\alpha + \gamma)$$
$$L_0 = H_{pixel} \tan\alpha$$
$$x_i = L_i - L_0 = H_{pixel} \tan(\alpha + \gamma) - H_{pixel} \tan\alpha$$
$$\beta = \theta_h \left(\frac{V - V_i}{V}\right)$$
$$y_i = L_i \tan(\theta_h - \beta)$$

This algorithm has been reported and tested with Python (see the `_top_view_transform` function) but the result was a not complete removal of perspective effect. So another simple method based on OpenCV docs has been adopted (see `four_point_transform`).

The `LaneEstimator` class has been created for lane detection, by using `_detect_lanes` function, and the top-view transform called `four_point_transform`. This function orders the source points found to be optimal for the transformation in our particular case and computes the perspective transformation map through `getPerspectiveTransform`, a OpenCV function. Then it is possible to apply this map to the original image through the `warpPerspective` function. Finally, the final image is cropped again to focus only on the area we are interested in, shown in Figure 4.8. Note that this image is converted to `uint8` type in order to allow an easy postprocess later on.

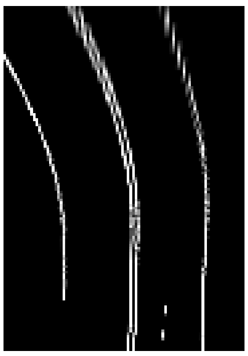


Figure 4.8: Top-view and cropped image

```
def four_point_transform(image, src):
    # obtain a consistent order of the points and unpack them
    # individually
    (tl, tr, br, bl) = src

    width = br[0] - bl[0]
    height = br[1] - tr[1]
    # now that we have the dimensions of the new image, construct
    # the set of destination points to obtain a "birds eye view",
    # (i.e. top-down view) of the image, again specifying points
    # in the top-left, top-right, bottom-right, and bottom-left
    # order
    dst = np.array([
        [0, 0],
        [width - 1, 0],
        [130, height - 1],
        [120, height - 1]], dtype = "float32")

    # compute the perspective transform matrix and then apply it
    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(image, M, (width, height))
    warped = warped[100:, 60:150]
    # return the warped image
    return warped.astype('uint8')

def _top_view_transform(self, cropped_image):
    top_image = np.empty([self.U, self.V], dtype=int)
    L0 = self.H_pixel * tan(self.tilt)
    for Ui in range(self.U):
        gamma = self.v_fov * ((self.U - Ui) / self.U)
        Li = self.H_pixel * tan(self.tilt + gamma)
        x = int(Li - L0)
        for Vi in range(self.V):
            beta = self.h_fov * ((self.V - Vi) / self.V)
            y = int(Li * tan(self.h_fov - beta))
            print(x, y)
            top_image[x][y] = cropped_image.getpixel((Ui, Vi))

    top_image = Image.fromarray(top_image, 'P')
    imageio.imwrite('top_view/_lanes.png', top_image)
```

4.2.4 Model fitting

After the Inverse Perspective Transformation, the obtained image has been used for model fitting, whose aim is to extract meaningful information from the pixels associated to lane lines. In this work, the python class `RoadModel` has been built to implement this step. Its constructor is called each time the method `compute_params`, part of the *LaneKeepingAlgorithm* class, is passed to the interpreter in the main cycle. This class divides the image received as input in two different sections, as suggested in [15]:

1. **Near view image**: for this section, a straight line detection algorithm has been developed by using a probabilistic Hough transform. The basic theory behind this technique has been already explained in subsection 4.2.1.

2. **Far view image**: on this section is applied a least-squares polynomial fitting by identifying the pixels that represent the curved lines and using the Numpy function `polyfit` inside `get_coeff` to compute the polynomial's parameters (c, d, e). By default, the polynomial order has been set equal to two, so a parabolic model is used. Giving the identified points representative of the lines, ordered as $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, the problem is quite simple:

$$\underbrace{\begin{bmatrix} n & \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i^3 \\ \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^4 \end{bmatrix}}_{M} \begin{bmatrix} c \\ d \\ e \end{bmatrix} = \underbrace{\begin{bmatrix} \sum_{i=1}^n y_i \\ \sum_{i=1}^n y_i x_i \\ \sum_{i=1}^n y_i x_i^2 \end{bmatrix}}_{Y} \tag{4.7}$$

   So the well known least-squares solution can be computed:

$$\begin{bmatrix} c \\ d \\ e \end{bmatrix} = M^\dagger Y \tag{4.8}$$

   where $M^\dagger$ is the pseudo-inverse.

An idea of the accuracy of the model fitting algorithm is given in Figure 4.9.

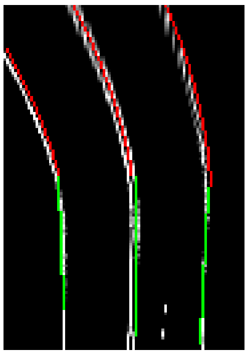


Figure 4.9: Model fitting algorithm: accuracy

Follows the code with detailed comments.

```python
def get_coeff(points, n):
    # Returns the coefficients of the polynomial of order n fitting the points
    # in 2D with Least Squares
    return list(np.polyfit(np.flip(points[:, 0]), np.flip(points[:, 1]), n))

def poly(coeff, y, n, w, point):
    # Uses the poly coefficients to compute x coordinates from y ones
    x2 = 0
    y2 = np.copy(y[:point])
    for i in range(n+1):
        x2 += coeff[i]*(y2**(n-i))

    h = len(x2) - 1
    # Limits the computation of x coordinates to the ones inside the image width
    for i in range(h, -1, -1):
        if x2[i] < 0 or x2[i] >= w:
            x2 = x2[h:i:-1]
            y2 = y2[h:i:-1]
            break
    return x2, y2


class RoadModel(object):
    """
    The constructor receives as input the image for model fitting, the lane
    width currently measured, the polynomial order for the far view section,
    the save-to-disk option and the fraction of the image height that is
    considered as near view section.
    """
    def __init__(self, img, lane_width,
        poly_order=2, save_to_disk=False, fract=0.5):
        self.img = img                            # Input image
        if self.img is None:
            print("Error")
            quit()
        self.h2 = self.img.shape[0]               # Image's height
        self.fract = fract                        # Near view section's fraction
        self.h = int(self.h2 * self.fract)        # Near view section
        self.w = int(self.img.shape[1])           # Image's width
        self.error = False                        # Error variable
        self.stride = 2 # n° of pixels ahead to be considered for the fitting
        self.skip = 5   # n° of pixels to skip when a line's pixel is identified
        y, x = self._get_left_point() # get the pixel at the bottom of left line
        self.img[y:, x:(x+1)] = 255   # extends the left line with a straight line
        self.image_to_display = np.copy(self.img) # image to be saved to disk
        self.real_width = lane_width              # Current lane width
        self.save_to_disk = save_to_disk          # Save-to-disk option
        self.poly_order = poly_order              # Polynomial order for far view
```

```
def _get_left_point(self):
    """
    It gets the pixel at the bottom of the left line
    """
    for i in range(self.h2 - 1, -1, -1):
        for j in range(int(self.w/3)):
            if self.img[i, j] > 200:
                return i, j

def get_dfc(self, center_line, right_line):
    """
    It returns the number of pixels between the center of the lane and the
    position of the front of the car. This measure is needed for converting
    it to meters
    """
    for i in range(self.h2-5, -1, -1):
        for j in range(center_line[0]+5, right_line[0]-5, 1):
            if self.image_to_display[i, j, 0] > 100:
                dfc = j - center_line[0]
                return dfc

def compute_params(self):
    """
-    It computes the Hough transform for the near view image by cropping
    the top view image to extract all the pixels that have the y
    coordinate between the bottom and a fraction of the height determined
    by the variable "fract" (default 0.5).
-    It converts the top view image from grayscale to RGB to be displayed
    after linear and parabolic fitting lines will be drawn on it.
-    It saves the points where the straight lines computed through Hough
    transform end. From this points to the upper end of the image, a
    polynomial least-squares fitting will be applied.
-    It checks if the straight lines have been detected or not.
    The fundamental    ones are the ones that delimit the lane on which the
    car is. The opposite one is useful to mediate the lines parameters,
    keeping the supposition    that the lines have quite identical parameters
    at a certain point.
-    It computes the lane slope from center and right line segments and the
    yaw error from it, converting the measure in radians.
-    It computes the lateral offset, that is the distance from the front of
    the car    and the center of the lane, by using the distance in the image,
    measured in    pixels, and then converting it to meters by exploiting the
    knolwedge of the lane width.
-    It uses the method "compute_poly_params" to fit the far view section and
    compute the curvature of the lane.
-    Finally, it returns lateral offset, yaw error and curvature, the
    parameters needed by the nonlinear MPC.
    """
    crop_img = self.img[self.h:, :]
    lines = cv2.HoughLinesP(crop_img, 4, np.pi/60, 10,
    minLineLength=10, maxLineGap=10)
self.image_to_display = cv2.cvtColor(self.image_to_display,
    cv2.COLOR_GRAY2RGB)
flag = [False, False, False]
[left_line, center_line, right_line] = (None, None, None)

for line in lines:
    for x1, y1, x2, y2 in line:
        y1 += self.h
        y2 += self.h

        if any(flag):
            if abs(x1-point[1])<5:
                continue

        if y1 < y2:
            point = [y1, x1]
        else:
            point = [y2, x2]
        # if all 3 lines are detected, display them and
        if x1 < self.w/3:
            left_points = np.array(point, dtype='int')
            left_line = (x1, y1, x2, y2)
            flag[0] = True
        elif x1 > self.w*2/3:
            right_points = np.array(point, dtype='int')
            right_line = (x1, y1, x2, y2)
            flag[1] = True
        elif x1 > self.w/3 and x1 < self.w*2/3:
            center_points = np.array(point, dtype='int')
            center_line = (x1, y1, x2, y2)
            flag[2] = True

        cv2.line(self.image_to_display, (x1,y1), (x2,y2), (0,255,0), 1)
        plt.plot([y1, y2], [x1, x2])
        if self.save_to_disk:
            cv2.imwrite('top_view/houghlines.png',
                        self.image_to_display)

    if all(flag):
        break

lines = [left_line, center_line, right_line]
left_detected = True
for cnt, line in enumerate(lines):
    if not line:
        if cnt == 0:
            print("Opposite lane line not detected")
            left_detected = False
        if cnt in (1, 2):
                raise Exception("""Attention,
                    line detection not ended correcty""")

        if left_detected:
            points = [left_points, center_points, right_points]
            end_points = (left_points[0], center_points[0], right_points[0])
        else:
            points = [center_points, right_points]
            end_points = (center_points[0], right_points[0])

        lane_width_pixel = right_line[0] - center_line[0]
        #slope of the straight lines wrt the vertical direction of the image
        lane_slope=((right_line[2]-right_line[0])/(right_line[3]-right_line[1])
           + (center_line[2]-center_line[0])/(center_line[3]-center_line[1]))/2
        yaw_error = radians(np.arctan(lane_slope))
        dfc = self.get_dfc(center_line, right_line)
        if dfc is None:
            dfc = lane_width_pixel
        lateral_error = (0.5 - float(dfc/lane_width_pixel))*self.real_width
        print("Lateral error: %.2f" %lateral_error)
        print("Yaw error: %.2f" %yaw_error)
        curvature = self.compute_poly_params(points, end_points, left_detected)
        return curvature, lateral_error, yaw_error

    def compute_poly_params(self, points, end_points, left_detected):
        """
    -    This function does the polynomial fitting of the lane lines in the far
        view section. It identifies the pixels that are representative of the
        lines and it inserts their coordinates in 3 different vectors, one for
        each line.
        In order to have more measurements to meadiate, also the line of the
        opposite lane is fitted (the left line). This isn't always detected
        and in this case the computation of lane parameters is done without
        the information coming from this line. The loop iterates through the
        entire image, when it identifies a pixel with value higher than a
        threshold it stacks the coordinate of the pixel x steps    ahead,
        with x=stride. This to be near the center of the line and not at the
        border.
        The image is vertically subdivided in 3 equal parts in order to
        cathegorize the pixels in left, center and right points. When a pixel
        is cathegorized, the loop advances of a number of pixels equal to "skip",
        to leave the already identified line and search for the others in the row.
    -    The get_coeff function is applied to each vector of points to get the
        coefficients of the identified polynomial and the mean of the curvature
        of the 3 lines is computed.
    -    Another loop uses the information obtained before to display in red the
        identified lines on the top-view image, if save_to_disk=True
    -    Finally, the mediated curvature needed by the contoller is returned
        """
        # Curved line detection
```

```
if left_detected:
    left_points, center_points, right_points = points
else:
    center_points, right_points = points
# Builds the data-structures that are used for fitting of the lane lines
for i in range(self.h2 - 1, -1, -1):
    p = 0
    for j in range(self.w):
        if (j + p) < self.w:
            if self.img[i, j + p] > 100:
                point = np.array([i, j + p + self.stride])
                p += self.skip
                if (j + p) < (self.w / 3 - 1):
                    if left_detected:
                        left_points = np.vstack((left_points, point))

                elif (j + p) < (2 * self.w / 3 - 1):
                    center_points = np.vstack((center_points, point))
                else:
                    right_points = np.vstack((right_points, point))
        else:
            break

center_coeff = get_coeff(center_points, self.poly_order)
right_coeff = get_coeff(right_points, self.poly_order)

if left_detected:
    left_coeff = get_coeff(left_points, self.poly_order)
    curvature = np.mean([left_coeff[0], center_coeff[0], right_coeff[0]])
    coeffs = [left_coeff, center_coeff, right_coeff]
else:
    curvature = np.mean([center_coeff[0], right_coeff[0]])
    coeffs = [center_coeff, right_coeff]

print("Curvature: %.2f" %curvature)

Y = np.linspace(0, self.h2, self.h2, endpoint=False)
for i, coeff in enumerate(coeffs):
    X2, Y2 = poly(coeff, Y, self.poly_order, self.w, end_points[i])
    X2 = np.around(X2)
    d = np.c_[X2, Y2]
    for row in d:
        self.image_to_display[int(row[1])][int(row[0])] = [0, 0, 255]

if self.save_to_disk:
    cv2.imwrite('top_view/final.png', self.image_to_display)

return curvature
```

4.2.5 Time integration

Scene recognition during motion is easier with a static camera if some knowledge about the motion behaviour of the vehicle carrying the camera, the so called "ego vehicle", is given. From Carla, it is possible to get velocity and position of the vehicle, augmenting the data coming from the sensor used for estimation of road parameters. This method uses a differential geometry representation of the road lines combined with an inverse perspective mapping (IPM) model based on a Cartesian space representation of the environment. During the estimation process, temporal continuity constraints derived from the known dynamics of the vehicle and constraints on the control input can be exploited.

In this work time integration is used for the estimation of the state of the vehicle for the NMPC but not for tracking of lane lines or objects. A Kalman filter could be used to build a model of the motion of the lane lines w.r.t. the ego-vehicle. This estimate is updated using the previous states and the current measurements. So this is a recursive approach that aim to keep the state estimate as accurate as possible, while minimizing the mean-square error and giving robustness to the vision algorithm. This step has not been implemented in this work because isn't part of the thesis goals. The result is that lane estimations can vary a lot between two successive time steps, even if we don't trust a lot the new measurements coming from the camera and we have a good a priori knowledge of the lane lines parameters. For example, from CARLA we can derive easily informations about lane width and near waypoints of the lines that could be used to enhance the dynamic model of the lines as a priori informations. In addition, in case of abrupt changes of line parameters when the car cross an intersection, without a Kalman filter the rapid changes in model fitting parameters could be critical.

The parabolic model fitting used here for the far section view and presented in 4.2.4 can be used for design a Kalman filter that is proposed in [12] and is reported here for clarity:

$$
\begin{aligned}
x(k|k-1) &= Ax(k-1) + w(k-1) \\
y(k) &= Cx(k) + v(k) \\
x &= \begin{bmatrix} \phi & \dot{\phi} & b_l & b_r & C & W \end{bmatrix}^T \\
A &= \begin{bmatrix} 1 & v\Delta t & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
C &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -\frac{1}{2} \\ 1 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}
\end{aligned}
$$

Other lane line dynamic models for Kalman filtering can be found in literature, for example in [7, chap. 3] and [5].

4.3 Vision dynamics

Here is presented a way to take into consideration the information coming from the camera to get the position of the car w.r.t the lane in order to perform a correct control action to steer the car[17]. The additional measurements provided by the vision system are (Figure 4.10):

- y_L The lateral offset of the center of mass of the car w.r.t. the center of the lane
- ϵ_L The relative yaw angle of the front of the car w.r.t. the tangent of the lane boundaries at the lookahead

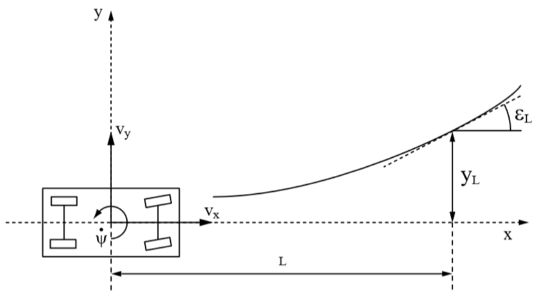


Figure 4.10: The vision system derives the distance y_L between the vehicle speed line at the lookahead L and the lane boundaries and also the yaw error ϵ_L

The dynamic of these two variables is fundamentally the following:

$$\dot{y}_L = v_x \epsilon_L - v_y - \dot{\psi} L \tag{4.9}$$

$$\dot{\epsilon}_L = v_x K_L - \dot{\psi} \tag{4.10}$$

where K_L represents the curvature of the road.

We can now combine the vehicle lateral dynamic and the vision dynamic systems together into a single augmented dynamical system with an exogenous measured disturbance:

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} + E\mathbf{w}$$
$$\mathbf{y} = C\mathbf{x}$$

where $\mathbf{x} = [v_y, \psi, y_L, \epsilon_L]^T$ is the state vector of the augmented system, $\mathbf{y} = [\dot{\psi}, y_L, \epsilon_L]^T$ is the output vector, $\mathbf{u} = \delta_f$ is the control input and $\mathbf{w} = K_L$ is the measured exogenous disturbance. The resulting combined model is:

$$\begin{bmatrix} \dot{v_y} \\ \ddot{\psi} \\ \dot{y_L} \\ \dot{\epsilon_L} \end{bmatrix} = \begin{bmatrix} -\dfrac{C_f + C_r}{mu} & -u - \dfrac{C_f a - C_r b}{mu} & 0 & 0 \\ -\dfrac{C_f a - C_r b}{I_z u} & -\dfrac{C_f a^2 + C_r b^2}{I_z u} & 0 & 0 \\ -1 & -L & 0 & u \\ 0 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_y \\ \dot{\psi} \\ y_L \\ \epsilon_L \end{bmatrix} + \begin{bmatrix} \dfrac{C_f}{m} \\ \dfrac{aC_f}{I_z} \\ 0 \\ 0 \end{bmatrix} \delta_f + \begin{bmatrix} 0 \\ 0 \\ 0 \\ v_x \end{bmatrix} K_L \tag{4.11}$$

$$y = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_y \\ \dot{\psi} \\ y_L \\ \epsilon_L \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \delta_f \tag{4.12}$$

As before, it is possible to derive the transfer function between the steering angle and the lateral offset from the state-space model. The difference in this case is only that this TF considers the offset at the lookahead and not the actual offset as before. This improvement is important mostly at high speeds, where the lookahead plays an important role for the stability of the systems. For the purposes of this thesis, high curvature roads with low speed, this difference is secondary if the sampling time is below 0.1 seconds but the system become unstable for higher sampling times. The analysis of possible control systems using these models is addressed in the next chapter.

Chapter 5

Vehicle dynamics

To effectively design a control system for autonomous vehicles, a good and possibly simple to handle dynamic model of the system under study is needed. In this chapter are presented the dynamic models that will be used for the design of the controllers. Here kinematic and dynamic aspects are explained in details to give a clear overview to the reader. All these models are derived from the bicycle model, whose dynamics is already implemented in the Vehicle Dynamics Blockset™ of Matlab[18]. This toolbox has been extensively used for the development of all the control strategies presented in this work.

5.1 Notation for vehicle dynamics

Vehicle motion is typically described in terms of the velocities: forward, lateral, vertical, roll, pitch and yaw in the vehicle-fixed coordinate system as referenced to an earth-fixed (i.e. inertial) reference frame (respectively u, v, w, p, q and r). A vehicle is decomposed in two main parts: the *unsprung mass* (the wheels and the part of suspension and braking system directly connected) and the *sprung mass* (all the rest). The Society of Automotive Engineers (SAE) has introduced standard coordinates and a notation to describe vehicle dynamics that are widely used [16, p. 54]. Here the earth-fixed reference frame's axes are referred as X, Y and Z and the vehicle reference frame's axes as x, y and z. The angles are roll (θ), pitch (ϕ) and yaw (ψ).

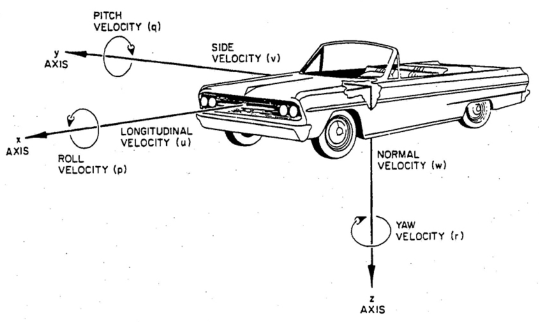


Figure 5.1: Vehicle dynamics coordinate system-SAE J760[1]

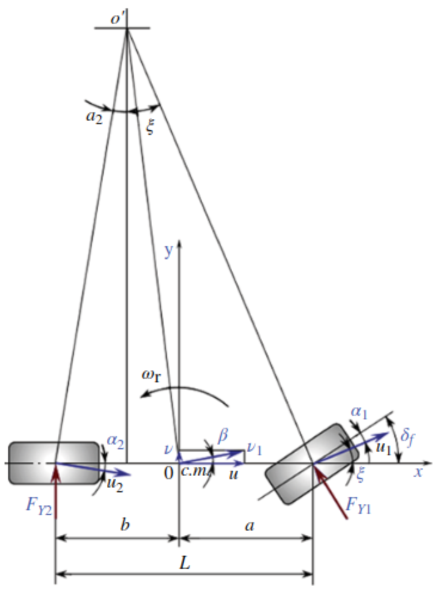


Figure 5.2: Bycicle model [19]

Referring to Figure 5.2:

- ψ is the yaw angle
- β is the sideslip angle
- u is the forward velocity
- ν is the lateral velocity
- $\rho = 1/R$ is the turn radius, where $R = \overline{OO'}$

The forces (F_x, F_y, F_z) and moments (M_x, M_y, M_z) are associated with both the tires and the subscripts f and r stay for front and rear tire respectively. In this simple case, to derive the equations of longitudinal and lateral motion we can use Newton's equations for translational motion in x and y and Euler's equations for angular motion in z. The external forces that act on the vehicle are tyre contact forces and aerodynamic forces. Before addressing the problem of modelling a simple dynamical system, a series of simplifications has been done:

- A "bicycle" dynamical model is considered, having longitudinal, lateral and yaw motion. That is, planar motion is parallel to the road's surface and no vertical, pitch or roll motion is present (left and right tyre side-slip angles are equal).
- The mass of the vehicle is lumped in the center of mass, lying in the segment that connects the two wheels.
- The tractive force comes only from the front wheel.

- The rolling resistance for both wheels is directly proportional to the normal force on the tire.
- Clearance and deformation of the steering system is neglected.
- The rotation angle of the front wheel δ_f is taken as an input directly.
- The change of the tyre characteristics and the role of the aligning torque on the left wheel caused by load changes are disregarded.
- The aerodynamic force is ignored (but will be taken into consideration by the Simulink model).

So, in the case of the linear model, the control system will have as control signal the angle of the front wheel δ_f. Thus the centrifugal force is generated at the center of mass, which causes the lateral reaction forces F_{y1} and F_{y2} and the corresponding cornering angles α_1 and α_2. The forward-velocity vector is $\vec{V} = u\mathbf{i} + v\mathbf{j}$. Despite these simplifications, the compliance of the pneumatic tires has been taken into account because it has a significant effect on vehicle dynamics. Recalling that $\frac{d\mathbf{i}}{dt} = r\mathbf{j}$ and $\frac{d\mathbf{j}}{dt} = -r\mathbf{i}$ it is possible to write:

$$\mathbf{a}_G = \frac{d\mathbf{V}_G}{dt} = \dot{u}\mathbf{i} + ur\mathbf{j} + \mathbf{j} - vr\mathbf{i} = (\dot{u} - vr)\,\mathbf{i} + (\dot{v} + ur)\,\mathbf{j} = a_x\mathbf{i} + a_y\mathbf{j} \tag{5.1}$$

$$a_x = \dot{u} - vr = \dot{u} - u^2\rho\beta \tag{5.2}$$

$$a_y = \dot{v} + ur = u\dot{\beta} + \dot{u}\beta + u^2\rho \tag{5.3}$$

5.2 Longitudinal Vehicle Motion

Application of Newton's equation in the x direction gives[16, p. 58]:

$$ma_x = F_{xr} - R_{xr} - R_{xf} - D_A - mg\sin\alpha \tag{5.4}$$

Where the aerodynamical force D_A is given by:

$$D_A = 0.5\rho C_d A(u + u_w)^2 \tag{5.5}$$

and F is the tractive force, R is the rolling resistance, C_d the drag coefficient, ρ the air density, A is the maximum vehicle cross-sectional area, u the vehicle velocity and u_w the wind velocity. As supposed before, a direct relationship between the rolling resistance and the normal force on the tire is present [16]:

$$R_x = R_{xr} + R_{xf} = f(F_{zf} + F_{zr})$$

where f is the rolling-resistance coefficient and F_{zf}, F_{zr} are the normal forces that act on the tire due to the vehicle weight.

Obviously, at the equilibrium the weight is balanced by the normal forces at the wheels. If the vehicle is still, the forces distribution is symmetric w.r.t. the longitudinal direction, but the more the vehicle is accelerating the more the forces distribution will act ahead from the center of the wheel and the rolling friction increases. In fact, the rolling coefficient is:

$$f = B_0 + B_2V^2 \tag{5.6}$$

where B_0 depends mainly on the road surface and B_2 on the tire.

5.3 Lateral Vehicle Motion

5.3.1 Model derivation

Now the dynamic equations for the single track (bicycle) model are derived. The velocities we have to consider are:

- u: forward speed
- v: lateral speed
- $r = \dot{\psi}$: yaw rate

Referencing the center of mass we have:

$$\mathbf{V_G} = u\mathbf{i} + v\mathbf{j} \tag{5.7}$$
$$\dot{\psi} = r\dot{\mathbf{k}} \tag{5.8}$$

Being $\mathbf{V}$ the speed of the centroid and R the distance between O and O', we have $V = R\dot{\psi}$ ($\omega_r = \dot{\psi}$) and the following relationship hold:

$$u = V \cos\beta \tag{5.9}$$

As the sideslip angle at the GC β is very small, we can assume $u = V = R\dot{\psi}$ and:

$$\beta = \arctan\frac{v}{u} \approx \frac{v}{u} \tag{5.10}$$

Consider now the lateral dynamics of the vehicle in the y direction due to steering. The following equations are derived applying Newton's second law to both lateral and circular motion:

$$F_{y1} + F_{y2} = ma_y = m(\dot{v} + u\dot{\psi}) \tag{5.11}$$
$$aF_{y1} - bF_{y2} = I_z\ddot{\psi} \tag{5.12}$$

where F_{y1}, F_{y2} are the lateral reaction forces on the front and rear wheels, m is the vehicle mass, a, b are the horizontal distance between the front and rear axles and the vehicle centroid, and I_z is the moment of inertia of the vehicle body with respect to the z-axis. The lateral forces are related to the slip angles α_1 and α_2. This relationship is nonlinear but for small slip angles we can use the approximation:

$$F_y = -C_\alpha \alpha$$

For the sake of simplicity this linear relationship will be used for the development of the theory. However, the bicycle model of Matlab divides the normal forces by the nominal normal load to vary the effective friction parameters during weight and load transfer:

$$F_{y1} = -C_{y1}\alpha_1\mu_f\frac{F_{z1}}{F_{znorm}} \qquad F_{y2} = -C_{y2}\alpha_2\mu_r\frac{F_{z2}}{F_{znorm}}$$

The block uses these equations to maintain pitch and roll equilibrium:

$$F_{z1} = \frac{bmg - (\dot{u} - v\dot{\psi})mh}{a+b} \qquad F_{z2} = \frac{amg - (\dot{u} - v\dot{\psi})mh}{a+b}$$

The front and rear tire slip angles can be obtained from the following geometrical relationships:

$$\alpha_1 = \arctan\left(\frac{v + a\dot{\psi}}{u}\right) - \delta_f \approx \frac{a\dot{\psi}}{u} + \beta - \delta_f \tag{5.13}$$

$$\alpha_2 = \arctan\left(\frac{v - b\dot{\psi}}{u}\right) \approx \beta - \frac{b\dot{\psi}}{u} \tag{5.14}$$

Therefore,

$$\alpha_1 - \alpha_2 = (a+b)\frac{\dot{\psi}}{u} - \delta_f = \frac{L}{R} - \delta_f \tag{5.15}$$

or

$$\delta_f = \frac{L}{R} - (\alpha_1 - \alpha_2) \tag{5.16}$$

where the static part is equal to $\frac{L}{R}$ (known as the *Ackermann angle*) and the dynamic part is equal to the difference between the two tire slip angles. Assuming that the vehicle is in a steady turn can give us a deep insight of the importance of the slippage of the tires in vehicle dynamics. In this condition $\dot{\psi} = 0$ and equations (2.5) and (2.6) becomes:

$$mu^2/R = F_{y1} + F_{y2} \tag{5.17}$$

$$0 = F_{y1}a - F_{y2}b \tag{5.18}$$

Thus the lateral forces become:

$$F_{y2} = m\frac{au^2}{LR} \qquad F_{y2} = m\frac{bu^2}{LR}$$

Using the relation between lateral forces and slip angles gives:

$$\alpha_1 = \frac{mu^2b}{RLC_f} \qquad \alpha_2 = \frac{mu^2b}{RLC_r}$$

where h is the vertical distance from center of mass to axle plane. Now, substituting these relationships in (2.9) and knowing that, in steady-state, $a_y = u^2/R$ gives:

$$\delta_f = \frac{L}{R} - \left(\frac{mb}{LC_f} - \frac{ma}{LC_r}\right)\frac{u^2}{R} \equiv \frac{L}{R} - Ka_y \tag{5.19}$$

where K is the stability factor [19]. When $K > 0$ the vehicle is termed *understeer*; when $K < 0$, the vehicle is termed *oversteer*; and when $K = 0$, the vehicle is termed *neutral steer*. For example, an understeer vehicle (as all commercial vehicles), the steer angle must increase with speed to maintain a constant-radius turn.

Having said that, transient 2-*dof* linear models are now considered. Rewriting equations (2.5) and (2.6) and still assuming that the front and rear tires are linear [16, p. 70], we have:

$$m(\beta + u\dot{\psi}) = C_f\delta_f - (C_f + C_r)\beta - \Big(\frac{C_f a - C_r b}{u}\Big)\dot{\psi} \tag{5.20}$$

$$I_z\ddot{\psi} = aC_f\delta_f - \Big(C_f a + C_r b\Big)\beta - \Big(\frac{C_f a^2 - C_r b^2}{u}\Big)\dot{\psi} \tag{5.21}$$

where u is the velocity component in the longitudinal direction. Rearranging this equations yields the state space system described as below:

$$\dot{\boldsymbol{X}} = \boldsymbol{Ax} + \boldsymbol{Bu} \tag{5.22}$$

and, remembering that $\beta = v/u$, it is possible to expand it obtaining:

$$\frac{d}{dx}\begin{bmatrix}\beta \\ \dot{\psi}\end{bmatrix} = \begin{bmatrix} -\left(\dfrac{C_f + C_r}{m}\right) & -\left(\dfrac{C_f a - C_r b}{mu}\right) - u \\ -\left(\dfrac{C_f a - C_r b}{I_z}\right) & -\left(\dfrac{C_f a^2 + C_r b^2}{I_z u}\right) \end{bmatrix}\begin{bmatrix}\beta \\ \dot{\psi}\end{bmatrix} + \begin{bmatrix}\dfrac{C_f}{m} \\ \dfrac{aC_f}{I_z}\end{bmatrix}\delta_f \tag{5.23}$$

5.3.2 Stability analysis of the linear model

The system above is internally stable if all the eigenvalues of the state matrix A have negative real part. In this case, the characteristic polynomial must have two roots with negative real part:

$$s^2 - tr(A)s + det(A) = 0 \tag{5.24}$$

$$tr(A) = -\left[\left(\frac{C_f + C_r}{m}\right) + \left(\frac{C_f a^2 + C_r b^2}{I_z u}\right)\right] < 0 \tag{5.25}$$

$$det(A) = \frac{C_f C_r\left(a^2 + b^2\right) - u^2 m\left(C_f a - C_r b\right)}{mI_z u} \tag{5.26}$$

Analysing equation 5.26, it is possible to compute the velocity that makes the state matrix singular, hence an eigenvalue becomes null and the system looses asymptotic stability. Imposing $det(A) = 0$ this speed, called *critical*, is:

$$V_{cr} = \sqrt{\frac{C_f C_r L^2}{m\left(C_f a - C_r b\right)}} \tag{5.27}$$

5.3.3 Augmented model

An augmented state space model of this problem is present in the literature [20] and will be used as the final model for the linear control strategy. For clarity y_r is defined as the lateral deviation of the mass center and ψ_d is the yaw angle of the trajectory to be followed, so the quantity $\psi - \psi_d$ has to be minimized too. So the final linear model of the front-wheel-steering vehicle is:

$$\frac{d}{dx}\begin{bmatrix} y_r \\ \dot{y_r} \\ \psi-\psi_d \\ \dot{\psi}-\dot{\psi}_d \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{A_1}{u} & -A_1 & \frac{A_2}{u} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{A_3}{u} & -A_3 & \frac{A_4}{u} \end{bmatrix} \begin{bmatrix} y_r \\ \dot{y_r} \\ \psi-\psi_d \\ \dot{\psi}-\dot{\psi}_d \end{bmatrix} + \begin{bmatrix} 0 \\ B_1 \\ 0 \\ B_2 \end{bmatrix}\delta_f + \begin{bmatrix} 0 \\ A_2-u^2 \\ 0 \\ A_4 \end{bmatrix}\rho \tag{5.28}$$

where:

$$A_1 = -\frac{(C_f+C_r)}{m} \qquad A_2 = -\frac{(C_f a - C_r b)}{m} \qquad A_3 = -\frac{(C_f a - C_r b)}{I_z}$$
$$A_4 = -\frac{(C_f a^2 + C_r b^2)}{I_z} \qquad B_1 = \frac{F_x + C_f}{m} \qquad B_2 = \frac{a(F_x + C_f)}{I_z}$$

where F_x is the tractive force in the front wheel. The output equation is:

$$y = y_r + d_s(\psi - \psi_d) = [1 \quad 0 \quad d_s \quad 0]\mathbf{x} \tag{5.29}$$

that represent the lateral deviation obtained when the distance between the center of the vehicle and the camera is d_s. The transfer function from the front wheel steer angle δ_f and the output y is[16, p. 354]:

$$\frac{y(s)}{\delta_f(s)} = \frac{1}{\Delta(s)}\left[(d_s B_2 + B_1)s^2 + \frac{d_s(B_1A_3 - B_2A_1) + B_2A_2 - B_2A_4}{u}s + B_1A_3 - B_2A_1\right] \tag{5.30}$$

where

$$\Delta(s) = s^2\left[s^2 - \frac{A_1 + A_4}{u}s + \frac{A_1A_4 - A_2A_3}{u^2} + A_3\right]$$

This state space equation with its transfer function will be used in PID control presented in Chapter 6, in parallel with the bicycle model block present in the Vehicle Dynamics Blockset. Also the nonlinear MPC will use a model similar to 5.28, but simple longitudinal dynamics is added and the longitudinal speed in this case is a state variable.

Chapter 6

Control

In this chapter are shown three different control strategies found in literature and adapted to this case. The vehicle control can be decoupled into lateral and longitudinal control and this way is used here for the design of the PID controller, focusing on lateral control. Other control techniques include Nonlinear Model Predictive Control, which is based on a nonlinear vehicle model, and end-to-end convolutional neural networks.

First of all, a linear feedback control strategy with PID control has been adopted. It is based on the vehicle linear model presented in Chapter 5.

Another hugely adopted control solution for trajectory tracking is the MPC, that allows to optimize a cost function over a *prediction horizon* in order to take present control actions that consider the estimated future behaviour of the system. Being an optimization problem, it naturally takes into account constraints on the various signals and saturation (i. e. physical limitations of the actuators). In particular, a Nonlinear MPC has been used [21]. These results have been obtained using Matlab and Simulink.

Finally, a ready-to-run code for end-to-end convolutional neural-networks has been written specifically to get informations from CARLA cameras and train the model. The training and the test wasn't possible for lack of computational power but a lot of research and test has been done these years for introducing models like this in steering control because they are able to extract all the relevant information from the road scenario without the need to label manually all the different actors.

6.1 PID controller

The main advantage of PID control is that is simple to design, to understand in action and to implement. The disadvantages are that doesn't provide any kind of optimization and adaptiveness to uncertainty in the plant and in the disturbances. In the algorithm that is proposed here, this control strategy works well for lane keeping but if the road and plant characteristics change consistently, the control parameters needs to be adjusted manually. This obviously isn't good for autonomous vehicles, that's why in the literature many kinds of adaptive PID controllers are proposed to update control parameters online. A possible and simple solution is based on the generalized minimum variance method [22]. Another possible strategy is to tune the PID parameters at different velocities and use the PID coefficients that have been tuned for speeds near to the actual one. Obviously in this way the adaptive capability of the controller considers only the speed and not the kind of road the car is riding on. The Simulink model of this control strategy is reported in figure 6.1.

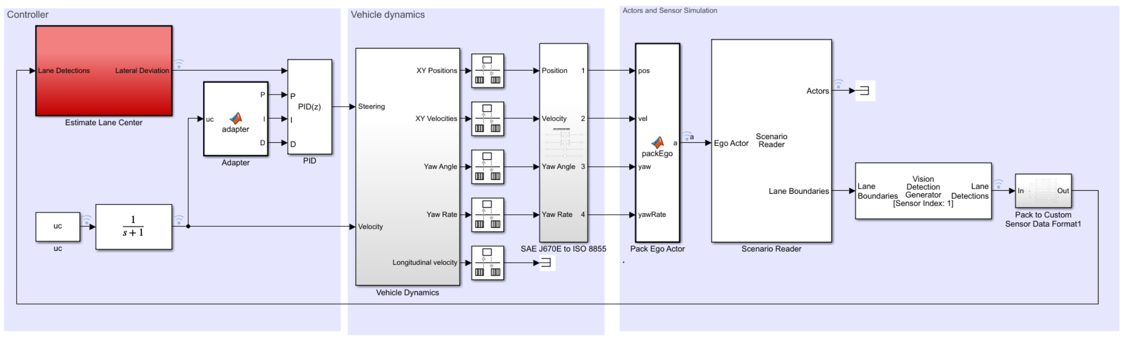


Figure 6.1: The Simulink model used for PID control

6.1.1 Plant

The plant used for controller design is linear and derived from the "bicycle" model state-space. The corresponding transfer function has been derived in Chapter 5, equation 5.30. The vehicle parameters are the following:

Parameter	Symbol	Value
Mass	m	2350 kg
Longitudinal distance from center of mass to front axle	a	1.4 m
Longitudinal distance from center of mass to rear axle	b	1.6 m
Vertical distance from center of mass to axle plane	h	0.5 m
Front tire corner stiffness	C_f	80000 N/rad
Rear tire axle corner stiffness	C_r	90000 N/rad
Yaw polar inertia	I_z	4132 kgm^2
Constant forward velocity	u	5-10-15-20 m/s
Distance between the center of the vehicle and the camera	d_s	0.9 m

Table 6.1: Ego-vehicle parameters

The plant continuous time transfer functions for different longitudinal velocities have been computed with Matlab an they are equal to:

- u = 5 m/s
$$G = \frac{438(s+14.11)(s+2.698)}{s^2(s+17.62)(s+15.59)}$$
- u = 10 m/s
$$G = \frac{58.438(s^2+8.404s+38.07)}{s^2(s^2+16.6s+74.48)}$$

- u = 15 m/s

$$G = \frac{58.438(s^2 + 5.603s + 38.07)}{s^2(s^2 + 11.07s + 37.4)}$$

- u = 20 m/s

$$G = \frac{58.438(s^2 + 4.202s + 38.07)}{s^2(s^2 + 8.302s + 24.43)}$$

Looking at these transfer functions, it's evident that increasing the speed moves zeros and poles towards the imaginary axis on the complex plane. They are all systems of type 2, so the systems are "structurally" unstable. This is obvious because a system with zero steering angle as input will never bring to zero the lateral error (or lateral offset as previously called) neither keep it constant.

Consider instead the transfer function without the 2 poles at the origin (Gs^2), that simply represents the system that has the steering angle as input and the acceleration as output. This is stable for velocities lower that the critical speed, as evident by looking to equation 5.27. The zeros and poles for each velocity can be visualized in Figure 6.2.

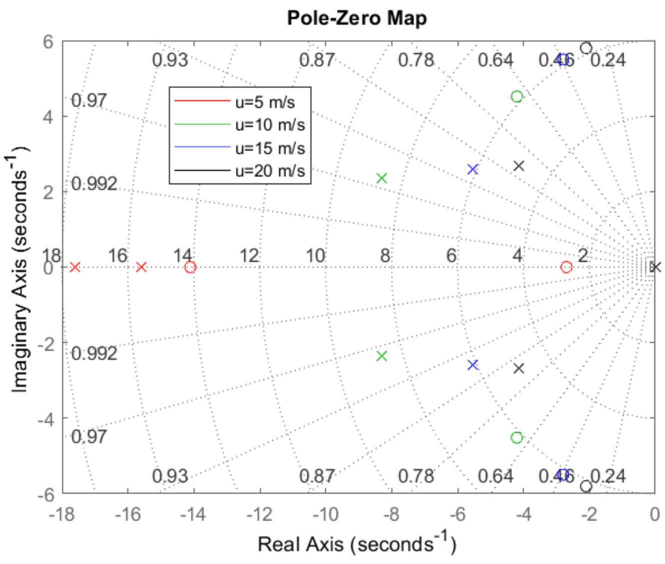


Figure 6.2: Pole-zero map at different velocities: as speed increases the zeros and poles move towards the right-half plane making the system less stable

With all these velocities set as reference ones, during simulations the vehicle has been able to successfully complete the track with a proper PID controller. The dynamical model of the vehicle used in simulations is encoded inside the the *Vehicle Body 3DOF block*, Figure 6.3. It implements a rigid two-axle vehicle body model to compute longitudinal, lateral, and yaw motion. The block accounts for body mass, aerodynamic drag and weight distribution between the axles due to acceleration and steering. It has been chosen because of its simplicity and the absence of pitch and roll motion dynamics that in this case can be neglected in first approximation.

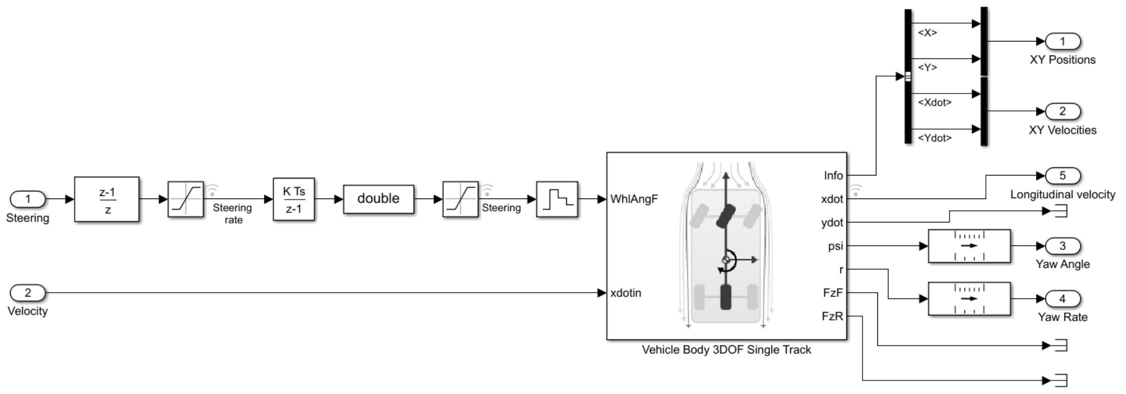


Figure 6.3: Simulink model of the plant for the PID control strategy

It has been configured with the parameters in Table 6.2 and with options `Single` and `External longitudinal velocity`. Its input are the velocity, that is a constant, and the steering angle signal coming from the controller. This signal and its derivative (the steering rate) are clamped with saturation constraints to take into account the physical constraints of a steering system. Also, being the control signal digital, a zero-order-hold (ZOH) block has been added before feeding it into the plant, that is simulated as a continuous dynamical system.

The outputs of the plant are:

- position and velocity components of the center of mass in inertial cartesian coordinates X, $\dot{X}$, Y, $\dot{Y}$
- velocity components in ego coordinates $\dot{x}, \dot{y}$
- yaw angle and its derivative ψ, r

Follows additional blocks called `Rate transition` that make the signal discrete before the scenario reader and the vision system. These blocks must receive discrete signals to update the vehicle state in the driving scenario and update the sensor measurements. The rate of transition is equal to the controller sample time. Then another block converts the physical quantities coming from the 3DOF block, expressed in SAE J670E convention (see Figure 5.1, Chapter 5), to ISO8855 convention (the lateral and yaw vectors have opposite directions, see Figure 6.4).

6.1.2 Scenario Reader and sensor simulation

The physical quantities of the ego vehicle converted before are then packed in a Matlab structure through the function `packEgo`, that is exploited by the `Scenario Reader` block. This Simulink block reads the scenario information and updates the pose and the speed of the ego car accordingly to data coming from the vehicle dynamics block. A valid recorded scenario has to be given. This data structure is obtained from the Matlab file coming from the Driving Scenario Designer App by invoking two functions. The

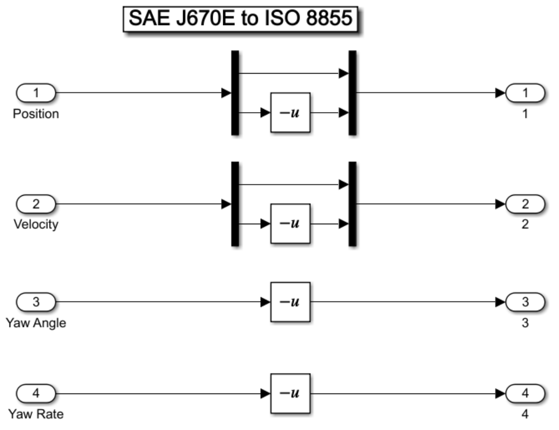


Figure 6.4: From SAE J670E to ISO8855

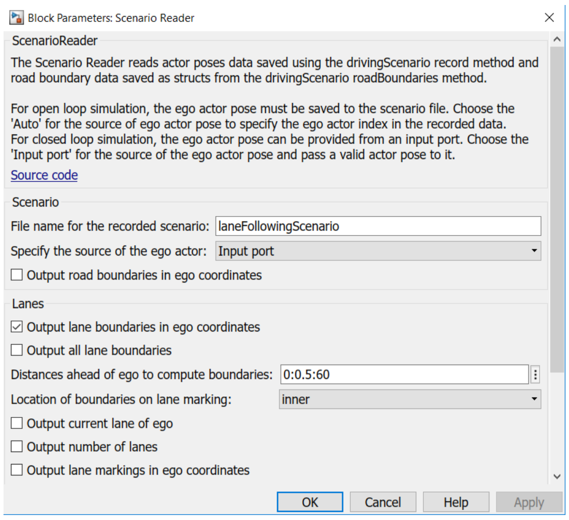
`helperSessionToScenario` extracts the scenario information plus the one of the ego car and other actors and put it in Matlab structures into the workspace.

The `helperSaveScenarioToFile` saves only the scenario structure in another file, the one used by `Scenario Reader`. This block outputs actors detections and the lane boundaries in ego coordinates at different equispaced lookaheads (see Figure 6.5). In this case, actor detections are discarded and lane boundaries detections are fed into the `Vision Detection Generator` block, that simulates a lane detection algorithm from camera measurements (see Figure 6.6). Here are defined the pose of the sensor, the output bus type, measurements and camera intrinsics settings. Also parameters related to precision and noise of lane detection can be set in order to put ourselves in a more realistic situation.

Finally, few words about the output bus. This is created by an already existing Matlab script called `helperCreateLanSensorBuses`, that creates a bus with 2 bus lines, one for left road line and one for the right one. Both lines are buses themselves that bring the line parameters identified by the `Vision Detection Generator` block. The angular quantities are converted in radians.

In this case, the lateral offset that is considered is the actual one, without lookahead. So, information about curvature of the lane ahead of the vehicle isn't necessary. All that is needed is the actual offset. This information is used to compute lane center. A Matlab function uses information about the strength of the lane lines captured by the vision system to conclude if both lines are observed or only one of them. In the first case, the lane center is computed by using the a-priori knowledge about the lane width (3.7 m in our situation) and adding or subtracting it to the offset depending if the line detected is the right or left one. In the second case both lines offset is used and an average is computed for noise filtering purposes (see Figure 6.7). Finally, a unitary minus is applied to be compliant with SAE J670E.

To connect lane detection data to the controller, specific buses created for this purpose have been used. The function `CreateLaneSensorBuses` creates a Simulink bus type that describes sensor structure interface for lane boundaries from lane sensor. It contains information about κ, $\dot{\kappa}$, ψ, e_1, e_2 and lane strength. This bus type is used by another bus, the `Lane Sensor` bus, to output left and right lane lines information in a single bus. This bus


Block Parameters: Scenario Reader

ScenarioReader

The Scenario Reader reads actor poses data saved using the drivingScenario record method and road boundary data saved as structs from the drivingScenario roadBoundaries method.

For open loop simulation, the ego actor pose must be saved to the scenario file. Choose the 'Auto' for the source of ego actor pose to specify the ego actor index in the recorded data. For closed loop simulation, the ego actor pose can be provided from an input port. Choose the 'Input port' for the source of the ego actor pose and pass a valid actor pose to it.

Source code

Scenario

File name for the recorded scenario: laneFollowingScenario

Specify the source of the ego actor: Input port

☐ Output road boundaries in ego coordinates

Lanes

☑ Output lane boundaries in ego coordinates

☐ Output all lane boundaries

Distances ahead of ego to compute boundaries: 0:0.5:60

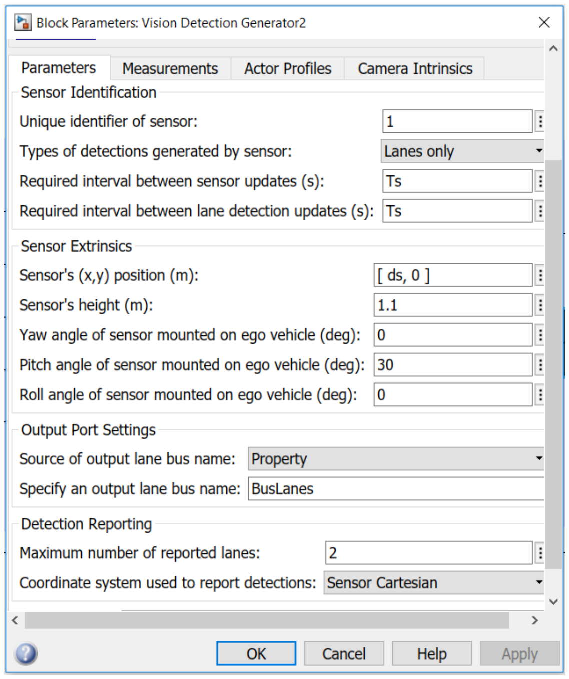
Location of boundaries on lane marking: inner

☐ Output current lane of ego

☐ Output number of lanes

☐ Output lane markings in ego coordinates

Figure 6.5: Scenario Reader block


Block Parameters: Vision Detection Generator2

Sensor Identification

Unique identifier of sensor: 1

Types of detections generated by sensor: Lanes only

Required interval between sensor updates (s): Ts

Required interval between lane detection updates (s): Ts

Sensor Extrinsics

Sensor's (x,y) position (m): [ds, 0]

Sensor's height (m): 1.1

Yaw angle of sensor mounted on ego vehicle (deg): 0

Pitch angle of sensor mounted on ego vehicle (deg): 30

Roll angle of sensor mounted on ego vehicle (deg): 0

Output Port Settings

Source of output lane bus name: Property

Specify an output lane bus name: BusLanes

Detection Reporting

Maximum number of reported lanes: 2

Coordinate system used to report detections: Sensor Cartesian

Figure 6.6: Vision Detection Generator block

type is used by the `Vision Detection Generator` block to send the information to the `Estimate Lane Center` block, see Figure 6.8.

Parameter	Value
Required interval between sensor updates (s)	T_s
Required interval between lane detection updates (s)	T_s
Sensor's (x,y) position (m)	$[d_s, 0]$
Sensor's height (m)	1.1 m
Yaw angle of sensor mounted on ego vehicle (deg)	0
Pitch angle of sensor mounted on ego vehicle (deg)	30
Roll angle of sensor mounted on ego vehicle (deg)	0
Maximum number of reported lanes	2
Coordinate system used to report detections	*Sensor Cartesian*
Maximum detection range (m)	5
Minimum lane size in image (pixels)	[20,3]
Accuracy of lane boundary (pixels)	3
Focal length (pixels)	[100,100]
Optical center of the camera (pixels)	[320,240]
Image size produced by the camera (pixels)	[800,600]
Radial distortion coefficients	[0,0]
Tangential distortion coefficients	[0,0]
Skew of the camera axes	0

Table 6.2: Vision system parameters

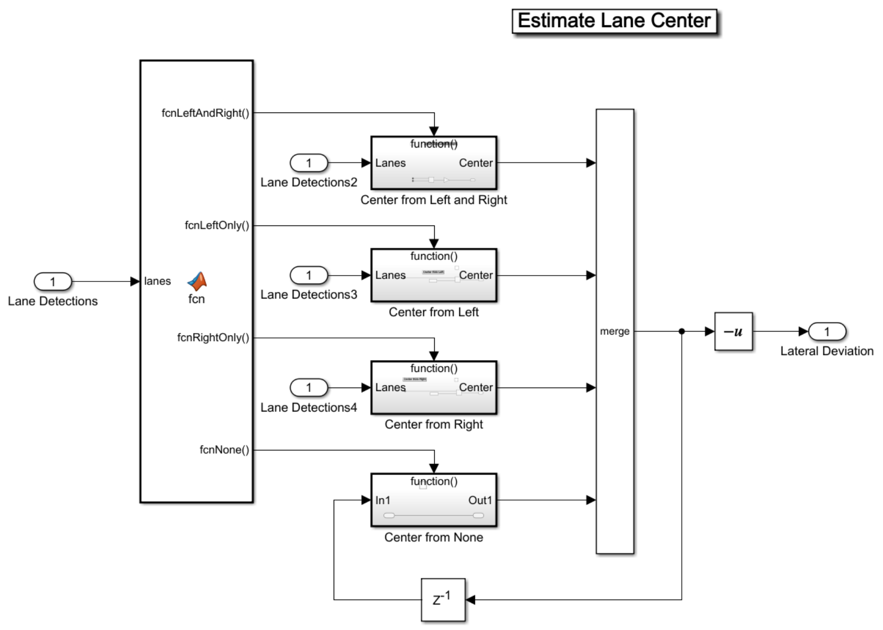
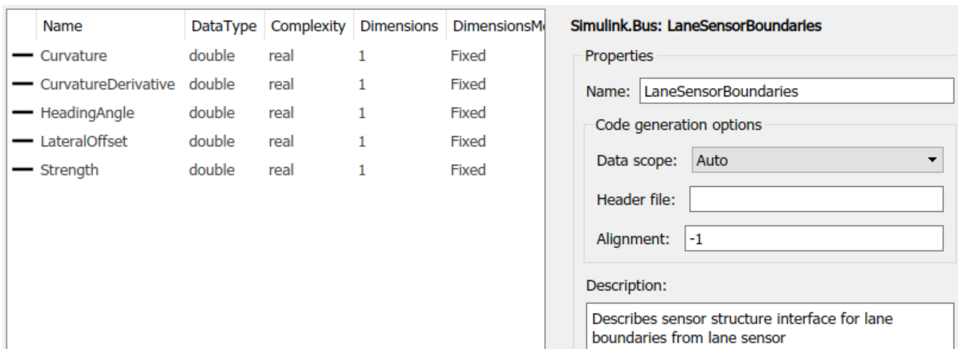


Figure 6.7: Lane center computation: the functions are selectively activated depending on which lanes are detected



Name	DataType	Complexity	Dimensions	DimensionsM
Curvature	double	real	1	Fixed
CurvatureDerivative	double	real	1	Fixed
HeadingAngle	double	real	1	Fixed
LateralOffset	double	real	1	Fixed
Strength	double	real	1	Fixed

Simulink.Bus: LaneSensorBoundaries

Properties

Name: LaneSensorBoundaries

Code generation options

Data scope: Auto

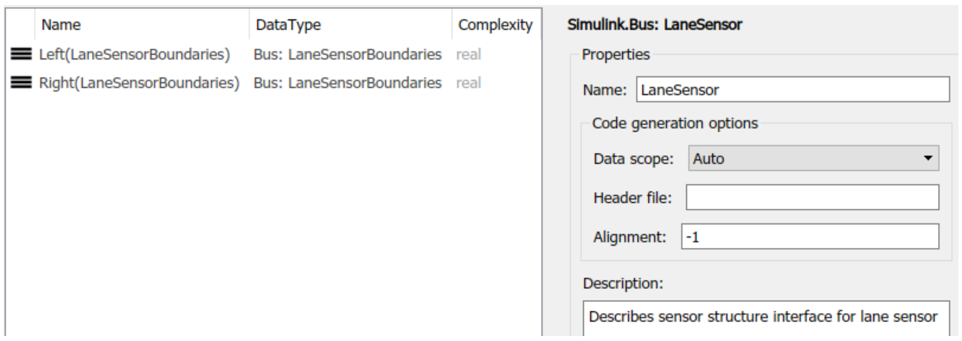
Header file:

Alignment: -1

Description:

Describes sensor structure interface for lane boundaries from lane sensor

(a) Lane Sensor Boundaries bus



Name	DataType	Complexity
Left(LaneSensorBoundaries)	Bus: LaneSensorBoundaries	real
Right(LaneSensorBoundaries)	Bus: LaneSensorBoundaries	real

Simulink.Bus: LaneSensor

Properties

Name: LaneSensor

Code generation options

Data scope: Auto

Header file:

Alignment: -1

Description:

Describes sensor structure interface for lane sensor

(b) Lane Sensor bus

Figure 6.8: Vision bus

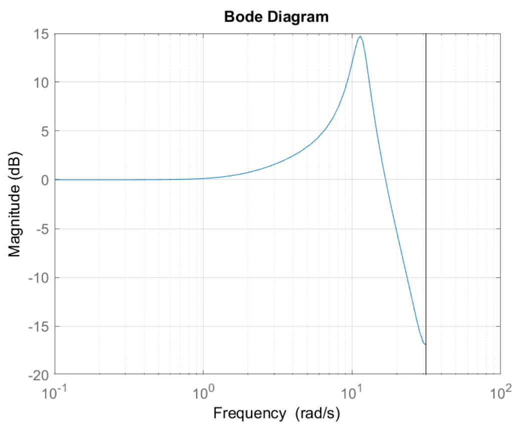
6.1.3 Controller

Now that all the information needed are available (both a-priori and real time ones), we can design and test our PID controller. As stated before, a classical Proportional-Integrative-Derivative controller is designed with the help of Matlab Control System Toolbox. The control law in continuous time is given by:

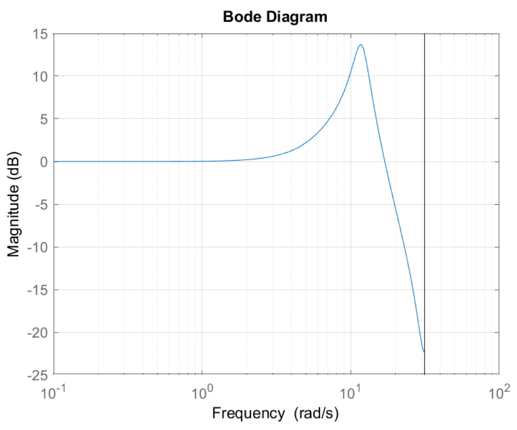
$$G_c(s) = \frac{\delta_f}{y_r} = K_p + \frac{K_I}{s} + K_D s \tag{6.1}$$

First, the three gains of the controller are computed with the Matlab function `pidtune`. Then these gains have been tuned manually to test the effect they have on the control performance and to get the best possible one. This controller in parallel form must be converted to a digital controller, so discretization must be applied after this manual tuning. The sampling time is $T_s = 0.1$ s for velocities up to 10 km/h and $T_s = 0.05$ s for higher speeds. A zero order hold (ZOH) is applied to the control input coming from the controller. The open loop transfer function is computed for each of the controllers and the bode diagrams with PID constants are reported in Figure 6.9. In Figure 6.10 are reported the corresponding graphs with plots of lateral velocity and steering angle.

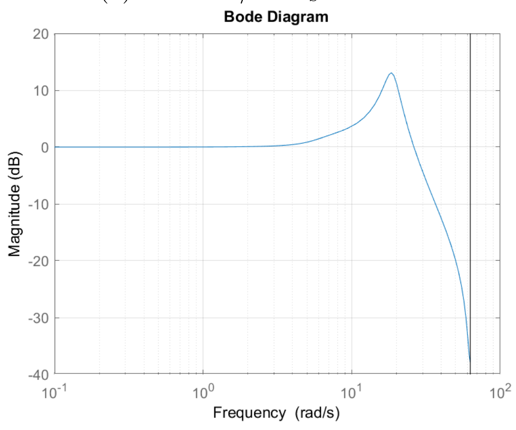
The tuning of the PID controller has to be done taking into consideration two conflicting goals: *handling*, that means requiring the lateral error to be bounded during all the travel, and *comfort*, that is directly related to the lateral acceleration and so to the steering rate.



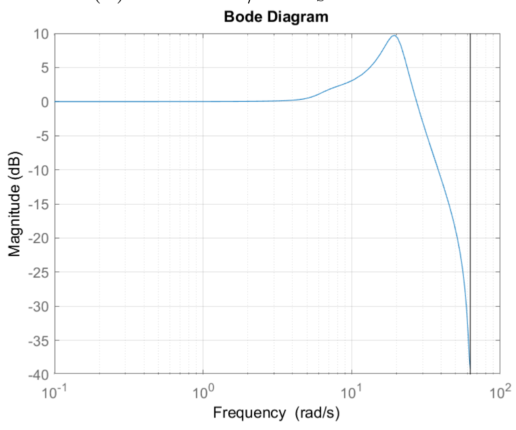
(a) u = 5 m/s - T_s = 0.1 s



(b) u = 10 m/s - T_s = 0.1 s



(c) u = 15 m/s - T_s = 0.05 s

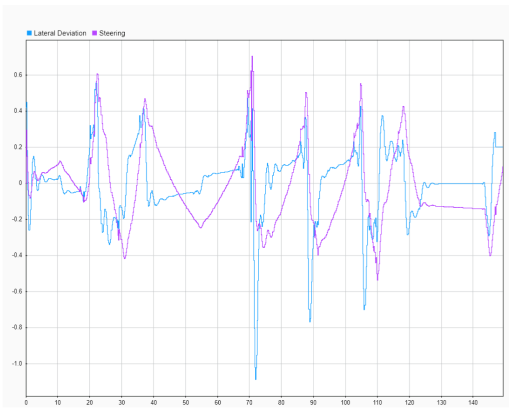


(d) u = 20 m/s - T_s = 0.05 s

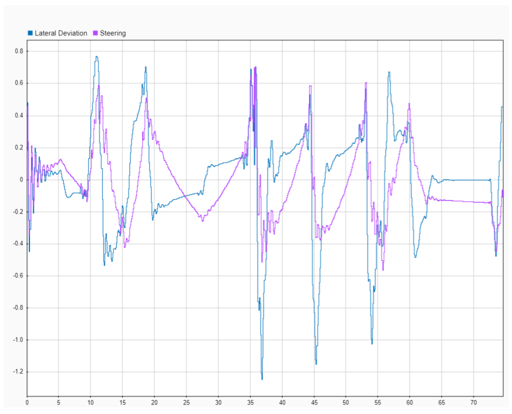
Figure 6.9: Bode plots of magnitudes for various longitudinal velocities

The trade-off can be set by tuning the PID coefficients. Some considerations have to be done about that:

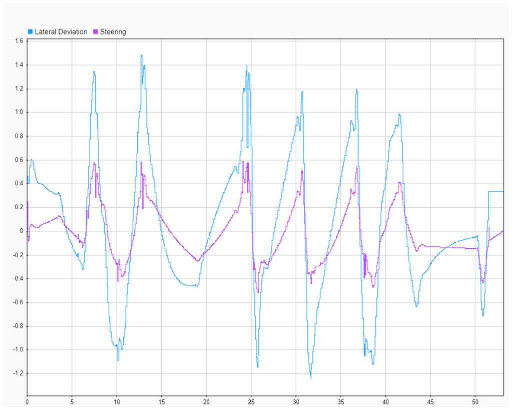
- $\mathbf{K_P}$: it multiplies directly the lateral deviation and is important to put it high enough to guarantee good lane-keeping action but moderate to avoid instability.
- $\mathbf{K_I}$: integral action is needed to have zero steady-state error. The higher the coefficient is, the faster the system will reach this condition. The counterpart of this action is that it has a phase delay of 90 degrees and this can bring problems in sharp changes of road curvature, in particular on S-shaped curves, where the lag can cause the vehicle to get out of the street.
- $\mathbf{K_D}$: derivative action that is very useful during high curvature rates to keep the vehicle on track. It produces a phase lead of 90 degrees that causes the system to react promptly on curves. The counterpart is that it amplifies the ripple due to errors in measurements, noise and unmodelled dynamics. In fact, when this term is too high, it forces the vehicle to steer right and left at a very high frequency even when the road is straight. This greatly worsens the comfort goal.



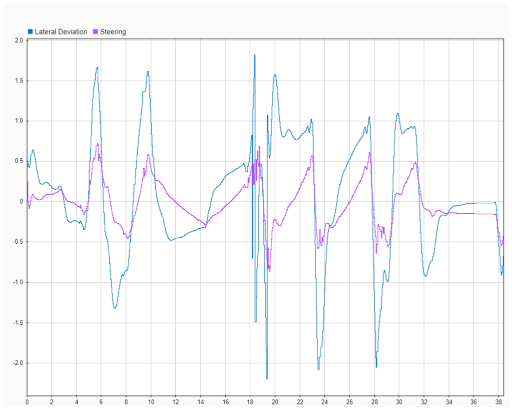
(a) $K_P = 4.16 \quad K_I = 5.25 \quad K_D = 0.28$



(b) $K_P = 3.5 \quad K_I = 6 \quad K_D = 0.7$



(c) $K_P = 7 \quad K_I = 3 \quad K_D = 0.7$



(d) $K_P = 6 \quad K_I = 7 \quad K_D = 0.8$

Figure 6.10: Plots of steering angle and lateral deviation in time for the four conditions reported in figure 6.9

By looking at the PID coefficients related to the various velocities, we see that when the speed is higher, the coefficients tend to be higher. This can be explained intuitively by the need of high controller action at high speeds, in particular in the derivative term that anticipates the changes in curvature and the integral one to reach zero steady state faster at higher speed.

A sort of adaptive control strategy has been obtained by simply dividing the speed range between 0 and 20 m/s in four equal intervals and by change the PID parameters depending on the interval on which the speed is. This allows speed variations, keeping in mind that high accelerations could be critical for this control system, that uses a 3DOF bicycle model.

A UI knob object, visible in Figure 6.11, has been created with the `uiknob` function to tune the speed of the car and adopt the gain-scheduling technique. Its initial value has been set to the initial velocity of the ego vehicle. The function `knobTurned` is called each time the knob is turned and updates the value of u_c and T_s in the running simulation. This last parameters is set to 0.1 s when $u_c < 12.5$ m/s and to 0.05 s when $12.5 < u_c < 20$ m/s. The scheduling logic and the modified PID gains for this case are:

- **C1**: $K_P = 4.16 \quad K_I = 5.25 \quad K_D = 0.28 \quad T_s = 0.1 \quad 0 \leq u_c < 7.5$
- **C2**: $K_P = 5 \quad K_I = 6.5 \quad K_D = 0.7 \quad T_s = 0.1 \quad 7.5 \leq u_c < 12.5$
- **C3**: $K_P = 7 \quad K_I = 5 \quad K_D = 0.7 \quad T_s = 0.05 \quad 12.5 \leq u_c < 17.5$
- **C4**: $K_P = 8 \quad K_I = 7 \quad K_D = 1 \quad T_s = 0.05 \quad 17.5 \leq u_c < 22$

The block `Adapter` runs each step a Matlab function that selects the correct controller parameters depending on u_c and updates the PID parameters at runtime. This speed is the one controlled by the knob filtered by a low-pass-filter of type $1/(\tau s + 1)$ to more realistically simulate the acceleration of the car. In fact, the 3DOF block takes as input the desired velocity but a step on it produces an abrupt acceleration on the vehicle that needs to be smoothed for comfort purposes. The time constant τ used to simulate longitudinal dynamics is 1 second.

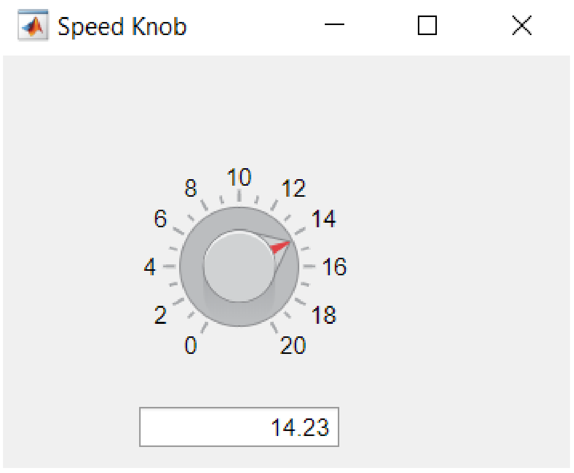


Figure 6.11: Knob used to tune the desired velocity of the ego vehicle. The real velocity changes with a first-order dynamics

The Matlab script and knob tuning function are presented below:

```
close all
%% PID deltas for tuning
Dd = 0.8;
Di = 7;
Dp = 6;
Ts = 0.05;
%% Input constraints
steering_saturation = 1.2;
steering_rate_saturation = 5;
%% Vehicle parameters
m = 2350;       %[kg] — mass
a = 1.4;        %[m]
b = 1.6;        %[m]
h = 0.5;        %[m] — Vertical distance from center of mass to axle plane
Cf = 80000;     %[N/rad]
Cr = 90000;     %[N/rad]
Iz = 4132;      %[kg*m^2]
Fx = 0;         %[N] — 0 if the vehicle drives at uniform velocity
ds = 0.9;       %[m] — distance between the center of the vehicle and the
    camera

%% Configuration of buses and simulation

% The drivingScenario session file is converted to a drivingScenario object
% initial conditions of ego car and actor profiles
fileName   = 'laneFollowingScenario';
[scenario,egoCar, actor_Profiles] = helperSessionToScenario('Scenario1');
helperSaveScenarioToFile(scenario,fileName);
load(fileName)
simStopTime = vehiclePoses(end).SimulationTime;

% Position and velocity selectors
posSelector = [1,0,0,0,0,0; 0,0,1,0,0,0]; % Position selector
velSelector = [0,1,0,0,0,0; 0,0,0,1,0,0]; % Velocity selector

% Initial condition for the ego car in ISO 8855 coordinates
v0_ego   = egoCar.v0;          % Initial speed of the ego car   (m/s)
uc       = v0_ego;
x0_ego   = egoCar.x0;          % Initial x position of ego car  (m)
y0_ego   = egoCar.y0;          % Initial y position of ego car  (m)
yaw0_ego = egoCar.yaw0;        % Initial yaw angle of ego car   (rad)
L = Ts*uc;                     % Lookahead distance             (m)

% Convert ISO 8855 to SAE J670E coordinates
```

```
y0_ego = -y0_ego;
yaw0_ego = -yaw0_ego;

% Save the driving scenario to the file to be read by Scenario Reader
modelName = 'PID_sim';
load_system(modelName)
CreateLaneSensorBuses;
blk=find_system(modelName,'System','helperScenarioReader');
s = get_param(blk{1},'PortHandles');
get(s.Outport(1),'SignalHierarchy');

%% Transfer function
s = tf('s');
A1=-(Cf+Cr)/m;
A2=-(Cf*a-Cr*b)/m;
A3=-(Cf*a-Cr*b)/Iz;
A4=-(Cf*a^2+Cr*b^2)/Iz;
B1=Cf/m;
B2=a*Cf/Iz;
Delta_s = s^2*(s^2-(A1+A4)/uc*s+(A1*A4-A2*A3)/uc^2+A3);
G=1/Delta_s*((ds*B2+B1)*s^2+(ds*(B1*A3-B2*A1)+B2*A2-B2*A4)/uc*s+B1*A3-B2*A1
    );
G = zpk(minreal(G,10e-3))
Gd = c2d(G,Ts);
Gd=zpk(Gd);
[C,info]=pidtune(G,'PID')
Kp = C.Kp + Dp;
Kd = C.Kd + Dd;
Ki = C.Ki + Di;
Cd = pid(Kp,Ki,Kd,1,Ts)
cl = feedback(Cd*Gd,1)
bodemag(cl), grid on


%% Plant analysis
load G1
load G2
load G3
load G4

figure
pzmap(G1,'r',G2,'g',G3,'b',G4,'k'), sgrid
legend('u=5 m/s','u=10 m/s','u=15 m/s','u=20 m/s')

%% Simulation
fig = uifigure('Name','Speed Knob','Position',[100 100 283 275]);
```

```
num = uieditfield(fig,'numeric','Position',[69 82 100 20]);
kb = uiknob(fig,'Position',[89 142 60 60],...
    'limits',[0 20],'ValueChangedFcn',@(kb,event) knobTurned(kb,event,num))
        ;
kb.Value = uc;
```

```
% Create ValueChangingFcn callback
function knobTurned(kb,event,num)
num.Value = event.Value;
assignin('base','uc',event.Value)
set_param('PID_sim/uc','Value','uc');
pause(0.5);
if event.Value <= 12.5
    assignin('base','Ts',0.1);
else
    assignin('base','Ts',0.05);
end
set_param('PID_sim/PID','SampleTime','Ts');
end
```

6.2 Model Predictive Control

A limitation of the PID controller in automated driving applications is the complete absence of predictions about the future state of the system in order to improve the performance. Information about future vehicle position and orientation w.r.t. the lane can be done using a dynamic model of the ego vehicle and vision system dynamics, reported in Chapter 5. Also disturbance and noise models can be used for prediction and state estimation. The model structure used in an MPC controller appears in Figure 6.12.

6.2.1 MPC fundamentals

Consider a discretized model of the continuous state space dynamical model of the vehicle, we have:

$$\vec{x}(k+1) = A\vec{x}(k) + B\vec{u}(k) + E\vec{w}(k) + F\vec{w}_{ud}(k) \tag{6.2a}$$
$$\vec{y}(k) = C\vec{x}(k) \tag{6.2b}$$

where $\vec{x}$ is the state, $\vec{u}$ is the manipulated input, $\vec{w}$ contains measured disturbance, $\vec{w}_{ud}$ the unmeasured one and $\vec{y}$ is the output of the system. This is a numeric LTI state-space model with stochastic disturbance. As common in this case, to achieve zero steady-state error, an integrator is included in the dynamical system:

$$\vec{x}(k+1) - \vec{x}(k) = A(\vec{x}(k) - \vec{x}(k-1)) + B(\vec{u}(k) - \vec{u}(k-1)) + E(\vec{w}(k) - \vec{w}(k-1))$$

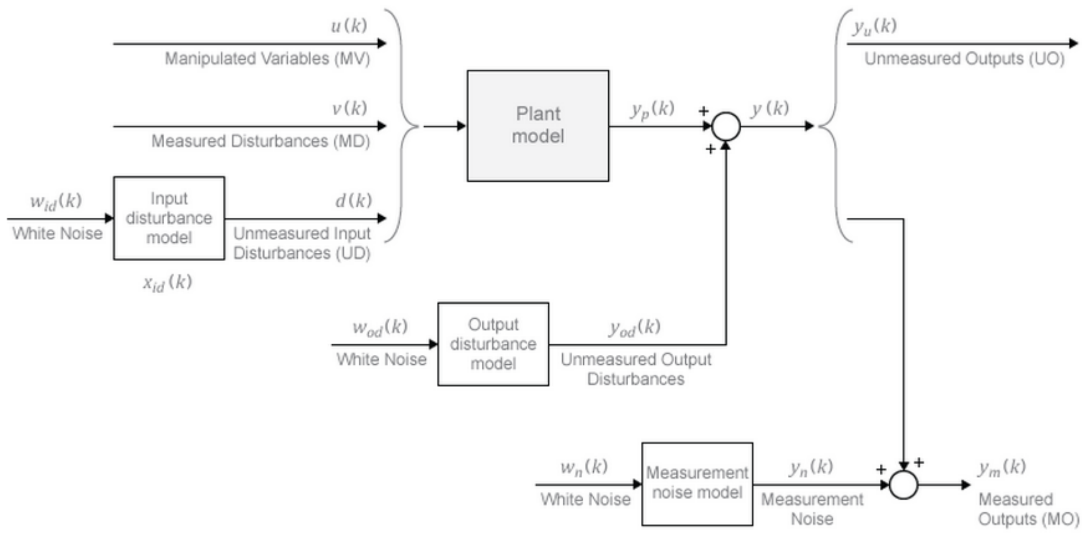


Figure 6.12: MPC model structure [23]

and, more compactly:

$$\Delta\vec{x}(k+1) = A\Delta\vec{x}(k) + B\Delta\vec{u}(k) + E\Delta\vec{w}(k) \tag{6.3a}$$

$$\vec{y}(k+1) - \vec{y}(k) = C(\vec{x}(k+1) - \vec{x}(k)) = CA\Delta\vec{x}(k) + CB\Delta\vec{u}(k) + CE\Delta\vec{w}(k) \tag{6.3b}$$

The state space equations of the augmented model can be rearranged in this form [21]:

$$\overbrace{\begin{bmatrix} \Delta\vec{x}(k+1) \\ \vec{y}(k+1) \end{bmatrix}}^{\vec{x}(k+1)} = \overbrace{\begin{bmatrix} A & \mathbf{0}^T \\ CA & \mathbf{1} \end{bmatrix}}^{A'} \overbrace{\begin{bmatrix} \Delta\vec{x}(k) \\ \vec{y}(k+1) \end{bmatrix}}^{\vec{x}(k)} + \overbrace{\begin{bmatrix} B \\ CB \end{bmatrix}}^{B'} \Delta\vec{u}(k) + \overbrace{\begin{bmatrix} E \\ CE \end{bmatrix}}^{E'} \Delta\vec{w}(k) \tag{6.4a}$$

$$\vec{y}(k) = \overbrace{\begin{bmatrix} \mathbf{0} & \mathbf{1} \end{bmatrix}}^{C'} \begin{bmatrix} \Delta\vec{x}(k) \\ \vec{y}(k) \end{bmatrix} \tag{6.4b}$$

The model predictive control is a digital control strategy based on the recursive computation of the state and the output of the dynamical system presented by equations 6.4 for H_p time steps. It minimizes a cost function of these variables with mathematical optimization solvers. This happens at each time step and the interval between two consecutive steps is equal to the sampling time T_s. The solver computes a trajectory of the control variable $\vec{u}$ to minimize the desired cost function. To reduce the computational complexity, only the first H_c control input components are computed, while the rest are put equal to the last one. In MPC terminology, H_p is called *prediction horizon* and H_c is called *control horizon*. Finally, only the first component of the control input vector is applied to the plant and the other are discarded. This technique is called *Receding Horizon Principle* and gives to the controller robustness against disturbances and unmodelled dynamics. Some considerations about these three coefficients has to be done here:

- $\mathbf{T_s}$ has to be chosen low enough to consider fast dynamics. Too broad discretization values can cause instability and weak control performances. Also the presence of delays must be considered before choosing the right T_s. Obviously, increasing the sampling time increases also the computational burden for the controller.

- $\mathbf{H_p}$ is the prediction horizon and indicates at how many steps ahead the controller will compute the state and the output of the system. The length of this time window must be at least equal to the slowest meaningful dynamics of the system. Its length has to be moderated to allow real time capabilities.

- $\mathbf{H_c}$ is the control horizon and indicates how many steps ahead the optimizer in the controller computes the control input. Usually is lower than half H_p.

Using Matlab, the controller updates its state automatically using the latest plant measurements [23]. It uses a steady-state Kalman filter. In our case, if we consider the linear system presented in Chapter 5, the predictions at each future time step can be analytically written. Note that these states are computed using linear predictions but then are filtered by the Kalman filter for the presence of an unmeasured disturbance w_{ud}, considered Gaussian. This input is a stochastically independent white noise, with zero mean and identity covariance.

$$\underbrace{\begin{bmatrix} \vec{x}(k+1|k) \\ \vec{x}(k+2|k) \\ \vec{x}(k+3|k) \\ \vdots \\ \vec{x}(k+H_p|k) \end{bmatrix}}_{\vec{\underline{x}}} = \underbrace{\begin{bmatrix} A \\ A^2 \\ A^3 \\ \vdots \\ A^{H_p} \end{bmatrix}}_{G} \vec{x}(k) + \underbrace{\begin{bmatrix} B & 0 & \dots & 0 \\ AB & B & \dots & 0 \\ A^2B & AB & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B \end{bmatrix}}_{H} \underbrace{\begin{bmatrix} \vec{\Delta u}(k) \\ \vec{\Delta u}(k+1) \\ \vec{\Delta u}(k+2) \\ \vdots \\ \vec{\Delta u}(k+H_p-1) \end{bmatrix}}_{\underline{\vec{\Delta u}}}$$

$$+ \underbrace{\begin{bmatrix} E & 0 & \dots & 0 \\ AE & E & \dots & 0 \\ A^2E & AE & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{H_p-1}E & A^{H_p-2}E & \dots & A^{H_p-H_c}E \end{bmatrix}}_{I} \underbrace{\begin{bmatrix} \vec{\Delta w}(k) \\ \vec{\Delta w}(k+1) \\ \vec{\Delta w}(k+2) \\ \vdots \\ \vec{\Delta w}(k+H_p-1) \end{bmatrix}}_{\underline{\vec{\Delta w}}} \tag{6.5a}$$

$$\underbrace{\begin{bmatrix} \vec{y}(k+1|k) \\ \vec{y}(k+2|k) \\ \vec{y}(k+3|k) \\ \vdots \\ \vec{y}(k+H_p|k) \end{bmatrix}}_{\underline{\vec{y}}} = \underbrace{\begin{bmatrix} CA \\ CA^2 \\ CA^3 \\ \vdots \\ CA^{H_p} \end{bmatrix}}_{S} \vec{x}(k) + \underbrace{\begin{bmatrix} CB & 0 & \dots & 0 \\ CAB & CB & \dots & 0 \\ CA^2B & CAB & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ CA^{H_p-1}B & CA^{H_p-2}B & \dots & CA^{H_p-H_c}B \end{bmatrix}}_{R} \underline{\Delta\vec{u}}$$

$$+ \underbrace{\begin{bmatrix} CE & 0 & \dots & 0 \\ CAE & CE & \dots & 0 \\ CA^2E & CAE & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ CA^{H_p-1}E & CA^{H_p-2}E & \dots & CA^{H_p-H_c}E \end{bmatrix}}_{Q} \underline{\Delta\vec{w}} \quad (6.5b)$$

So, in the case of a LTI system, the future predictions of our system can be computed with the a priori knowledge of A, B, C, E, the previewed measured disturbance vector $\Delta\vec{w}$ and the control input vector $\Delta\vec{u}$ computed by the optimizer. This part of the controller have to minimize a suitable cost function, chosen by the designer to meet some functional and performance goals. In the case of traditional linear MPC this cost function must be linear and this can be shown to lead to a quadratic program, that is a convex problem [21]. Moreover, the optimizer can handle constraints on the manipulated variables and their maximum variation between two successive steps. This is useful to incorporate physical constraints and limitations of the actuators inside the optimization directly and gives to the designer an important tool to reach his design goals. The constraints can be also imposed on the output vector:

$$\begin{aligned} \vec{u_{min}}(k_i) - \epsilon \leq \vec{u}(k_i) \leq \vec{u}_{max}(k_i) + \epsilon \\ \Delta\vec{u_{min}}(k_i) - \epsilon \leq \Delta\vec{u}(k_i) \leq \Delta\vec{u}_{max}(k_i) + \epsilon \\ \vec{y_{min}}(k_i) - \epsilon \leq \vec{y}(k_i) \leq \vec{y}_{max}(k_i) + \epsilon \end{aligned}$$

This for $i = 1, ..., H_p$. Matlab uses also a slack variable ϵ to soften constraints in case of necessity but this is important for numerical problems and is negligible in practice.

In the next subsection, it's presented the model for the design of the MPC controller. Many researchers have studied the MPC for the problem of lane keeping on low-curvature roads by linearisation of the dynamic model and by decoupling longitudinal and lateral dynamics, as have been done here with a simpler PID controller (see [24], [25], [26] and [27]). In this work, MPC has been used to control not only the steering, as in the previous case, but also the acceleration. This goal can be reached by implementing the longitudinal dynamics inside the model, side-by-side with lateral dynamics previously considered. This means that the longitudinal velocity can change. It enters in the state vector and the differential equations of the model. That is, the longitudinal velocity has a nonlinear characteristic and the problem can be solved by moving from linear models to nonlinear or adaptive models.

Adaptive models are computationally more efficient because the prediction model is updated at run time with the current values of the parameters and the model stays constant for all the prediction horizon. This condition has been found to impact a lot on lane keeping performance in case of high varying curvature roads, where velocity can change a lot and the model is highly nonlinear. For this reason, adaptive MPC has been studied and tested, but isn't shown in this thesis. Further works shall investigate the use of this control strategy in similar scenarios.

The nonlinear MPC (NMPC) has the same characteristics of traditional linear MPC but the prediction model, the constraints and the scalar cost function to be minimized can be nonlinear functions. This broadens a lot the possibilities of using a more realistic model of the system, but complicates a lot the mathematical problem, that now requires non-convex optimization techniques to find the optimal solution. The minimization algorithm that Matlab uses for the NMPC is inside the `fmincon` function. This kind of controller is the one used in this section to implement a lane keeping algorithm with curvature preview.

6.2.2 MPC vehicle model

So the MPC bases its entire prediction capability on the dynamical model that we give to it. That's why a suitable model has to be chosen as a trade-off between simplicity for real time needs and complexity for taking into consideration the significant dynamics. In this case the model is derived from the bicycle model used for PID control, but simple longitudinal dynamics and unmeasured disturbance have been added. The adopted model, with equations 6.6, has been found in [23]. To avoid confusion between the control input and the longitudinal velocity, another naming convention has been adopted, as visible in Figure 6.13. The model presented here is nonlinear because the longitudinal velocity varies in a nonlinear way, and depends on the input acceleration. This term varies a lot during rapid changes of curvature and enters in the state of the system.

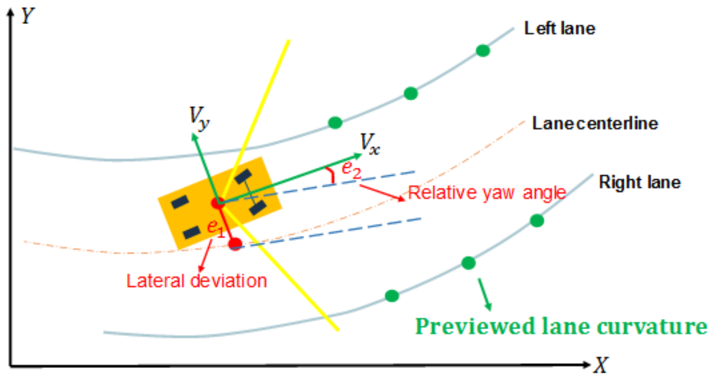


Figure 6.13: Conventions for the MPC dynamic model [23]

$$\frac{d}{dx}\begin{bmatrix} V_y \\ \dot{\psi} \\ V_x \\ \dot{V}_x \\ e_1 \\ e_2 \\ x_{ud} \end{bmatrix} = \begin{bmatrix} A_1 & A_2 & 0 & 0 & 0 & 0 & 0 \\ A_3 & A_4 & 0 & 0 & 0 & 0 & 0 \\ \dot{\psi} & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} & 0 & 0 & 0 \\ 1 & 0 & e_2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_y \\ \dot{\psi} \\ V_x \\ \dot{V}_x \\ e_1 \\ e_2 \\ x_{ud} \end{bmatrix} + \begin{bmatrix} 0 & B_1 \\ 0 & B_2 \\ 0 & 0 \\ \frac{1}{\tau} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ \delta_f \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -V_x \\ 0 \end{bmatrix} \rho + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} w_{ud} \tag{6.6a}$$

$$y = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} V_y \\ \dot{\varphi} \\ V_x \\ \dot{V}_x \\ e_1 \\ e_2 \\ x_{ud} \end{bmatrix} \tag{6.6b}$$

where:

$$A_1 = -\frac{2C_f + 2C_r}{mV_x} \qquad A_2 = -\frac{2C_f a - 2C_r b}{mV_x} - V_x \qquad B_1 = \frac{2C_f}{m}$$
$$A_3 = -\frac{2C_f a - 2C_r b}{I_z V_x} \qquad A_4 = -\frac{2C_f a^2 + 2C_r b^2}{I_z V_x} \qquad B_2 = \frac{2C_f a}{I_z}$$

Here these coefficients have to be recomputed at each iteration because the speed is a variable. The measured disturbance in this case is the curvature, previewed by the vision system and lane estimation block. The unmeasured disturbance w_{ud} instead is a white noise that enters in the state of the system in the state equation and contributes as an additive error to the yaw error e_2 at the output. The constraints on the input variables and their derivative are encoded directly in the controller.

It's evident that the equations are nonlinear because products between state variables happen. So the system is not linear nor time-invariant. For example, $\dot{\psi}$ and V_x multiply other state variables. To improve the efficiency by exploiting a linearization of the system, the functions that contains the Jacobians of the state and output equation are passed to the nonlinear MPC Matlab object. The Jacobians of the state equation are computed each time step easily:

$$J_A = \begin{bmatrix} \frac{\partial \dot{V}_y}{\partial V_y} & \frac{\partial \dot{V}_y}{\partial \dot{\psi}} & \frac{\partial \dot{V}_y}{\partial V_x} & 0 & 0 & 0 & 0 \\ \frac{\partial \ddot{\psi}}{\partial V_y} & \frac{\partial \ddot{\psi}}{\partial \dot{\psi}} & \frac{\partial \ddot{\psi}}{\partial V_x} & 0 & 0 & 0 & 0 \\ \frac{\partial \dot{V}_x}{\partial V_y} & \frac{\partial \dot{V}_x}{\partial V_x} & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} & 0 & 0 & 0 \\ 1 & 0 & \frac{\partial \dot{e}_1}{\partial V_x} & 0 & 0 & \frac{\partial \dot{e}_1}{\partial e_2} & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad J_B = \begin{bmatrix} 0 & \frac{\partial \dot{V}_y}{\partial \delta} \\ 0 & \frac{\partial \ddot{\psi}}{\partial \delta} \\ 0 & 0 \\ \frac{\partial \ddot{V}_x}{\partial a} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{6.7}$$

Being the output equation LTI, the Jacobian is not needed because it coincides with the C matrix of equation 6.6b. Now that the plant and disturbance models have been presented, the controller features are shown.

6.2.3 NMPC controller

The nonlinear optimization problem can be written as:

$$\begin{aligned} \min_{\vec{u}} \quad & J = \|\tilde{W}\underline{\vec{y}}\|^2 + \|\tilde{Y}\underline{\vec{u}}\|^2 + \|\tilde{V}\underline{\vec{\Delta u}}\|^2 \\ \text{s.t.} \quad & \begin{bmatrix} -I \\ I \end{bmatrix} \underline{\vec{u}} \leq \begin{bmatrix} -\underline{\vec{u}}^{min} \\ \underline{\vec{u}}^{max} \end{bmatrix} \\ & \begin{bmatrix} -I \\ I \end{bmatrix} \underline{\vec{\Delta u}} \leq \begin{bmatrix} -\underline{\vec{\Delta u}}^{min} \\ \underline{\vec{\Delta u}}^{max} \end{bmatrix} \end{aligned} \tag{6.8}$$

where $\tilde{W}$, $\tilde{Y}$ and $\tilde{V}$ are diagonal and positive definite matrices of weights of the form:

$$\tilde{W} = \begin{bmatrix} W_1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & W_1 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & W_1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & W_2 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & W_2 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & W_3 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & W_3 \end{bmatrix} \tag{6.9}$$

$$\tilde{Y} = \mathbf{0}_p \qquad \tilde{V} = \begin{bmatrix} V_1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & V_1 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & V_1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & V_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & V_2 \end{bmatrix} \tag{6.10}$$

The vectors $\underline{\vec{y}}$, $\underline{\vec{u}}$ and $\underline{\vec{\Delta u}}$ are computed as shown in equation 6.5 and are rearranged grouping their components w.r.t. the parameter type and chronological order, for example:

$$\underline{\vec{y}} = \begin{bmatrix} V_x(k+1|k) \\ V_x(k+2|k) \\ \vdots \\ V_x(k+N|k) \\ e_1(k+1|k) \\ e_1(k+2|k) \\ \vdots \\ e_1(k+N|k) \\ e_2(k+1|k) + w_{ud}(k) \\ e_2(k+2|k) + w_{ud}(k+1) \\ \vdots \\ e_2(k+N|k) + w_{ud}(k+N) \end{bmatrix} \qquad \underline{\vec{u}} = \begin{bmatrix} a(k+1|k) \\ a(k+2|k) \\ \vdots \\ a(k+N|k) \\ \delta(k+1|k) \\ \delta(k+2|k) \\ \vdots \\ \delta(k+N|k) \end{bmatrix} \qquad \underline{\vec{\Delta u}} = \begin{bmatrix} \Delta a(k+1|k) \\ \Delta a(k+2|k) \\ \vdots \\ \Delta a(k+N|k) \\ \Delta\delta(k+1|k) \\ \Delta\delta(k+2|k) \\ \vdots \\ \Delta\delta(k+N|k) \end{bmatrix}$$

The first matrix is used to weight the outputs, that in this case are the speed V_x, the lateral error e_1 and the yaw error e_2, all inside the prediction horizon. The second and third matrices are used to weight the control input $\vec{u}$ and its rate $\vec{\Delta u}$, that in our case are the steering and acceleration and their derivatives. As evident from definitions in 6.10, the input vector $\underline{\vec{u}}$ hasn't been considered in the cost function of the controller and to reach this goal its weigh matrix has been set as a null matrix. The motivation for this choice is that we don't want to penalize acceleration and steering as long as they fulfill the constraints on them. It's important that in sharp curves the car is allowed to brake and steer continuously with its full potential if necessary. The constraints instead will be active during some operations and a considerable effort has been put to choose the optimal constraints for the controller. Note that the constraints are not only necessary to take into account physical limitations of the mechanics of the car but also to limit some control inputs that would be simply unbearable for the system or annoying for the passengers. So they are much stricter than the real physical constraints. Opposite considerations has to be done about the input rate $\underline{\vec{\Delta u}}$ because the magnitude of its component has important consequences on stability and comfort of the passengers. Steering rate and jerk has to be limited with suitable constraints to have a smooth ride with minimal longitudinal and lateral jerks. By studying the scientific literature about passenger comfort in transport systems, it has been found that a good limit on jerk is $0.9\,m/s^3$ and a good limit on steering rate is $1.5\,rad/s$.

The controller and vehicle parameters are reported in Table 6.3 and the Simulink model in Figure 6.14. The `Collision Detection` block was pre-built in Matlab examples and

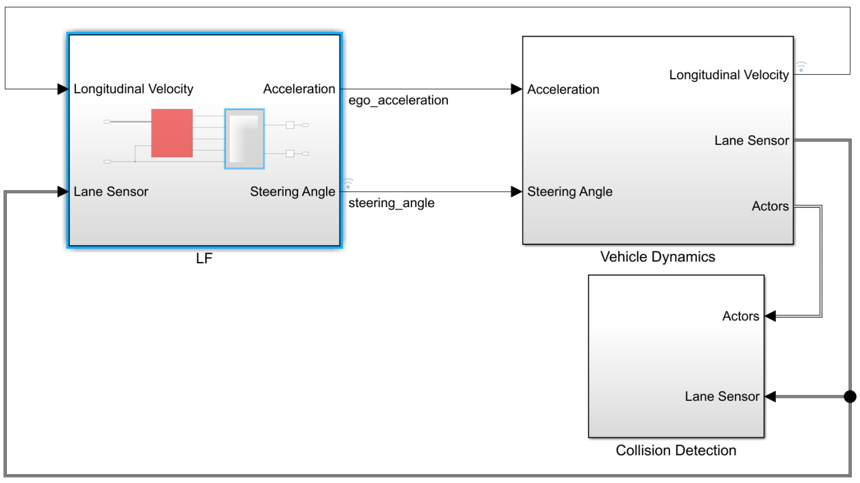


Figure 6.14: Simulink model of the Nonlinear MPC

simply stops the simulation in case of collisions with other actors. The vehicle dynamics and vision camera parameters used for simulation is the one used with PID control in the previous section, so all the considerations done before hold. The differences here regard the lane center estimation and the controller. The lane parameter estimation and controller blocks are presented in Figure 6.15. About the first, the informations of interest are not only the lateral offset as in PID control, but also the yaw error, the previewed curvature and its derivative. As before, the Simulink model checks how many lanes are detected by the `Vision Detection Generator` block. If both are detected, it computes the mean values for each lane parameter to filter noise. Else, it returns the unique identified lane parameters or the previous ones in case of failed detection. Then it computes the previewed curvature knowing the velocity V_x, the curvature and its derivative at time k:

$$\rho(k+i) = \rho(k) + V_x \dot{\rho} \Delta t_i, \qquad \forall i = 1, \ldots, H_p \tag{6.11}$$

where $\Delta t_i = T(k+i) - T(k)$ and i is the number of steps ahead at which curvature is

Parameter	Symbol	Value
Mass (kg)	m	1575
Yaw moment of inertia (kgm^2)	I_z	2875
Longitudinal distance from c.g. to front tires (m)	a	1.2
Longitudinal distance from c.g. to rear tires (m)	b	1.8
Vertical distance from center of mass to axle plane (m)	h	0.5
Cornering stiffness of front tires (N/rad)	C_f	19000
Cornering stiffness of rear tires (N/rad)	C_r	33000
Distance between c.g and camera (m)	d_s	0.9
Time constant for modelling longitudinal dynamics (s)	τ	0.2
Required interval between sensor updates (s)		$\mathrm{T_s}$
Required interval between lane detection updates (s)		$\mathrm{T_s}$
Sensor's (x,y) position (m)		$[\mathrm{d_s}, 0]$
Sensor's height (m)		1.1
Yaw angle of sensor mounted on ego vehicle (deg)		0
Pitch angle of sensor mounted on ego vehicle (deg)		1
Roll angle of sensor mounted on ego vehicle (deg)		0
Maximum number of reported lanes		2
Coordinate system used to report detections		*Ego Cartesian*
Maximum detection range (m)		30
Minimum lane size in image (pixels)		[20,3]
Accuracy of lane boundary (pixels)		3
Focal length (pixels)		[100,100]
Optical center of the camera (pixels)		[320,240]
Image size produced by the camera (pixels)		[800,600]
Radial distortion coefficients		[0,0]
Tangential distortion coefficients		[0,0]
Skew of the camera axes		0

Table 6.3: Vehicle and vision system parameters

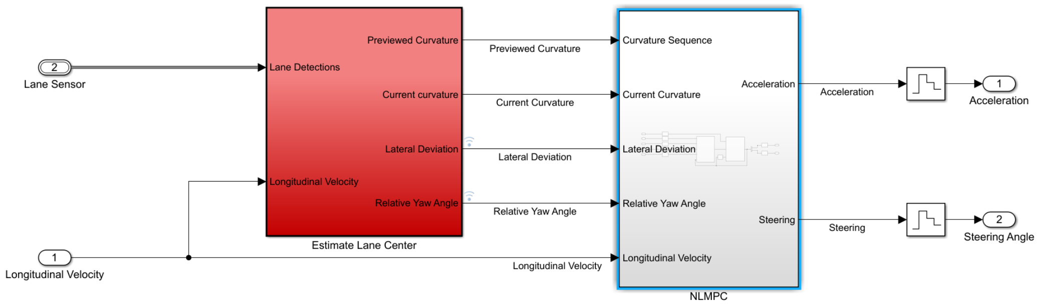
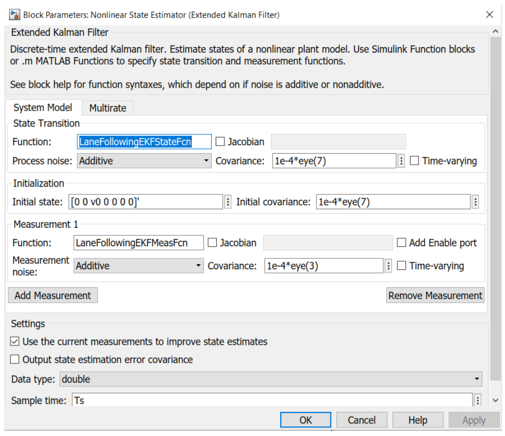


Figure 6.15: Lane center estimation and controller blocks

computed.
In order to estimate the state of the system through the estimates coming from the `Estimate Lane Center` block and the longitudinal velocity, an Extended Kalman filter is used. It estimate the state of a nonlinear system using the first-order discrete time extended Kalman filter algorithm [23]. The biggest contribution to the process noise comes from the lane detection. In fact, in the `Vision Detection Generator` block noise has been added. Other sources of noise could be added in the dynamic measurements of the 3DOF model, that by itself produces precise values, as the longitudinal velocity. In real applications, velocity must be estimated itself and the estimation error is modelled here adding noise to the simulation. The Kalman filter block needs the state transition and measurement functions of the system as input (see Figure 6.16). These functions (`LaneFollowingEKFStateFcn` and `LaneFollowingEKFMeasFcn`) are provided by the Matlab help and have been modified for the specific dynamic model. That's all the Kalman filter needs to estimate the state of the system. This estimate is used by the nonlinear MPC controller that predicts future outputs of the system and solves the problem 6.8. The design of the controller has been done using the Matlab class `nlmpc` that creates a nonlinear MPC controller object ready to be designed and deployed in a Simulink block. To instantiate this object the arguments in Table 6.4 must be passed. The controller parameters and constraints are reported in Table 6.5. In order to improve numerical conditioning for optimization, scale factors has been added for each input variable, looking at the operating range of each one.


Block Parameters: Nonlinear State Estimator (Extended Kalman Filter)

Extended Kalman Filter

Discrete-time extended Kalman filter. Estimate states of a nonlinear plant model. Use Simulink Function blocks or .m MATLAB Functions to specify state transition and measurement functions.

See block help for function syntaxes, which depend on if noise is additive or nonadditive.

System Model | Multirate

State Transition

Function: LaneFollowingEKFStateFcn ☐ Jacobian

Process noise: Additive Covariance: 1e-4*eye(7) ☐ Time-varying

Initialization

Initial state: [0 0 v0 0 0 0 0]' Initial covariance: 1e-4*eye(7)

Measurement 1

Function: LaneFollowingEKFMeasFcn ☐ Jacobian ☐ Add Enable port

Measurement noise: Additive Covariance: 1e-4*eye(3) ☐ Time-varying

Settings

☑ Use the current measurements to improve state estimates

☐ Output state estimation error covariance

Data type: double

Sample time: Ts

Figure 6.16: Extended Kalman Filter block

Parameter	Value
Number of state variables $\vec{x}$	7
Number of output variables $\vec{y}$	3
Indices of measured inputs $\vec{u}$	[1 2]
Index of the measured disturbance $\vec{w}$	3
Index of the unmeasured disturbance w_{ud}	4

Table 6.4: nmpc arguments

Parameter	Symbol	Value
Time gap (s)	T_G	0.1
Sampling time (s)	T_s	0.1
Maximum acceleration (m/s^2)	a^{max}	1
Minimum acceleration (m/s^2)	a^{min}	-1
Maximum jerk (m/s^3)	Δa^{max}	0.9
Minimum jerk (m/s^3)	Δa^{min}	-0.9
Maximum steering angle (rad)	δ^{max}	0.8
Minimum steering angle (rad)	δ^{min}	-0.8
Maximum steering rate (rad/s)	$\Delta\delta^{max}$	1.5
Minimum steering rate (rad/s)	$\Delta\delta^{min}$	-1.5
Set velocity (m/s)	V_{set}	15-20
Prediction Horizon (s)	H_p	6
Control Horizon (s)	H_c	2
Output's weight matrix non-zero entries	W_1, W_2, W_3	0.2, 1, 0
Input rate's weight matrix non-zero entries	V_1, V_2	2, 1

Table 6.5: Nonlinear MPC parameters

After having set all the MPC parameters, the dynamic model of the car and its Jacobian has been used in the controller. With this information, the optimizer computes the predictions and consequently the value of the cost function to be minimized.

The control inputs are then sent to the plant (Figure 6.18), that is similar to the one used in section 6.1. However, here the vehicle is controlled by steering and acceleration commands. There's no need of saturation blocks for these quantities here, because the constraints are embedded directly in the controller in the form of mathematical optimization constraints. Using a simple 1st order transfer function to model longitudinal dynamics behaviour [23, chap. 11] allows to use the bicycle model with force input instead of velocity input as with PID controller (see Figure 6.19). The force signal is applied by a half to the front axle and by the other half to the rear axle (see figure 6.17). Another difference already discussed is the presence of an additive noise on the longitudinal velocity measurements.

Finally the function `validateFcns` is used to validate the prediction model functions at an arbitrary operating point. The results of the simulations at 15 m/s and 20 m/s are presented in Figure 6.20.

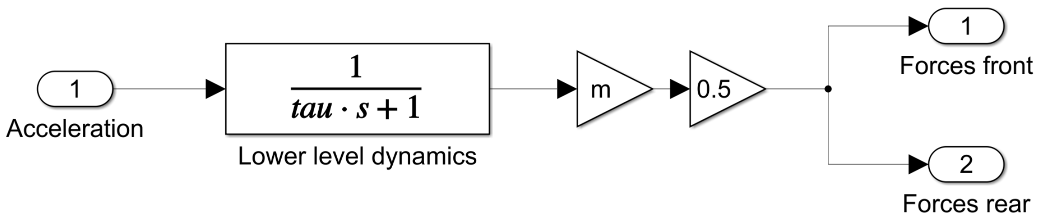


Figure 6.17: Lower level dynamics block

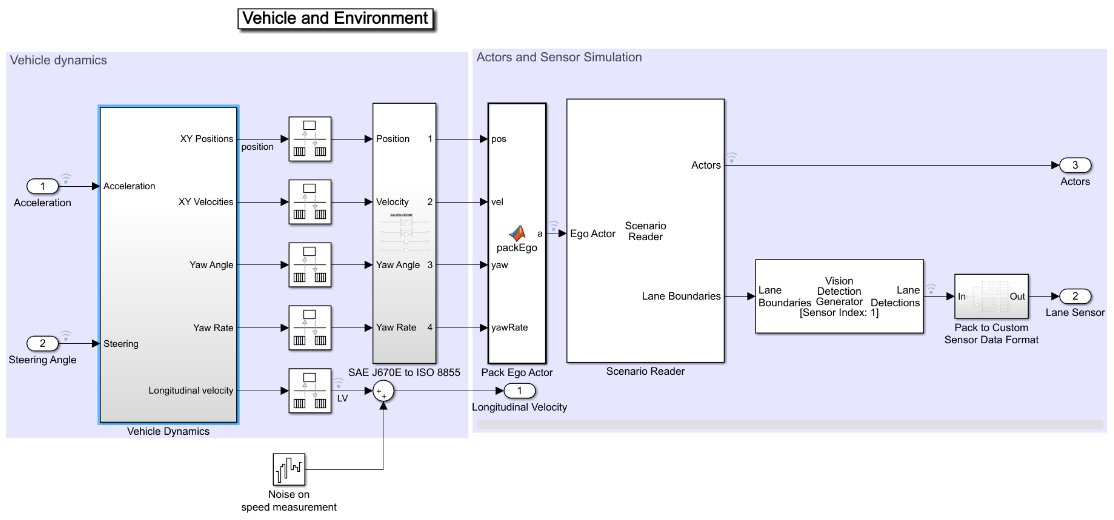


Figure 6.18: Vehicle Dynamics and environment Simulink scheme

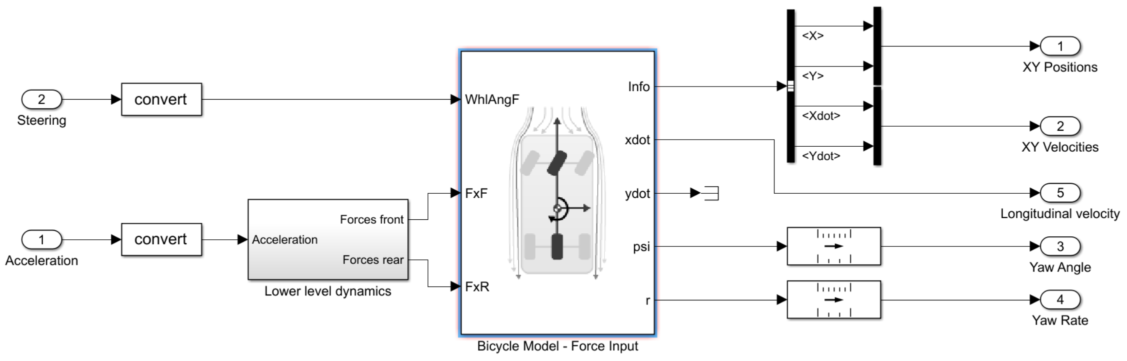
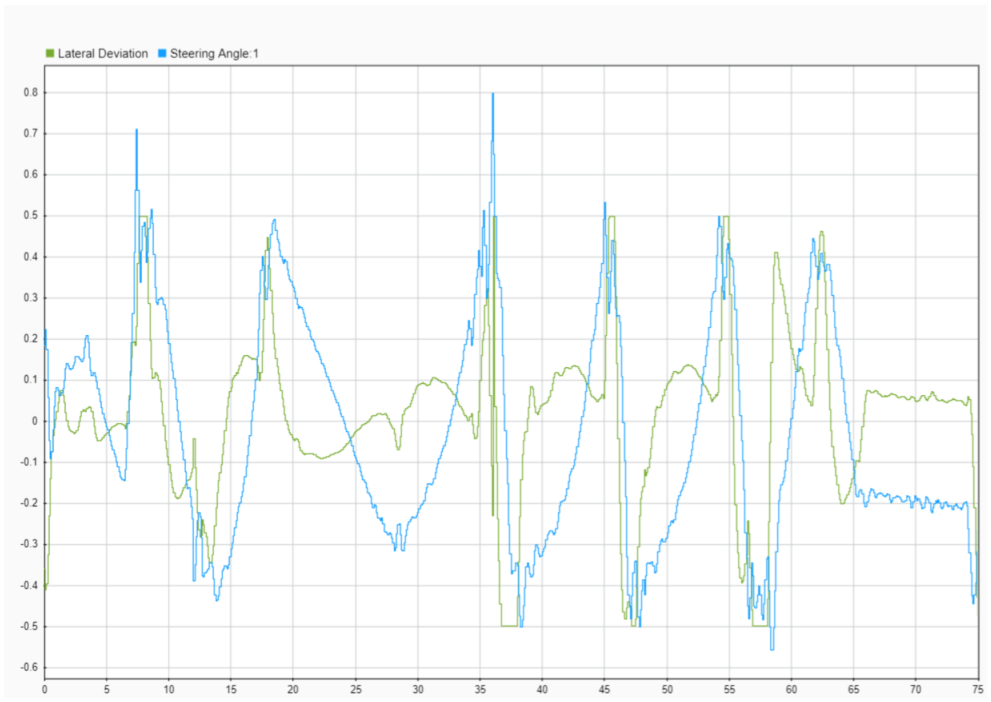
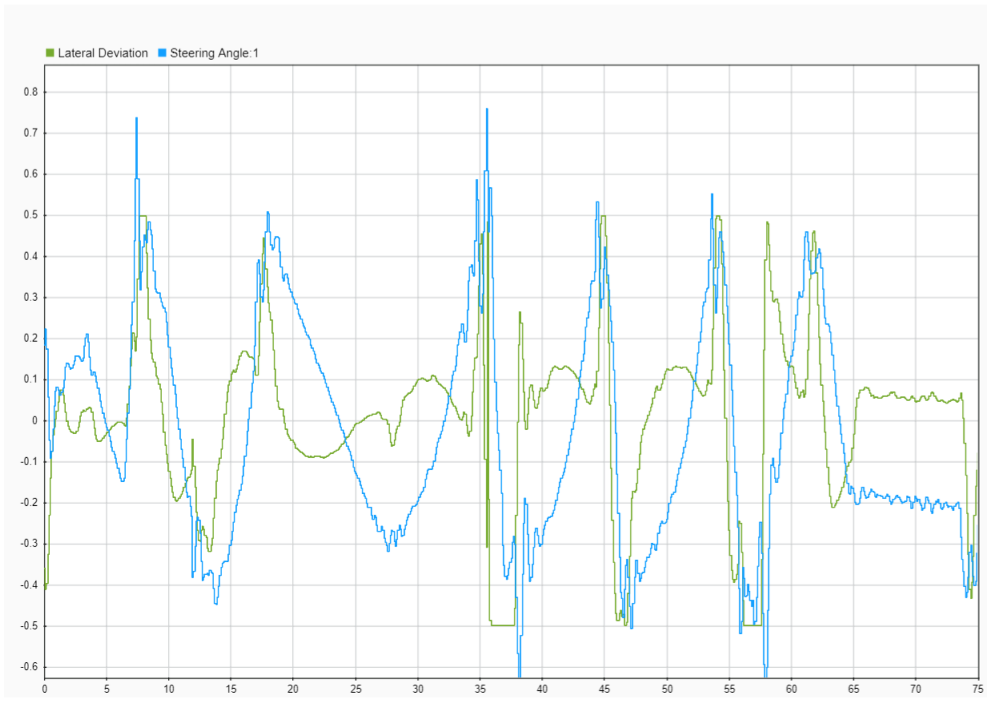


Figure 6.19: Vehicle Dynamics Simulink block



(a)



(b)

Figure 6.20: Plot of lateral deviation and steering timeseries at (a) 15 m/s and (b) 20 m/s

The Matlab code is presented below:

```
%% Lane Following Using Nonlinear Model Predictive Control

%% Path following Controller Parameters
time_gap            = 0.1;   % Time gap              (s)
Ts                  = 0.1;   % Sampling time         (s)
max_ac              = 1;     % Maximum acceleration  (m/s^2)
min_ac              = -2;    % Minimum acceleration  (m/s^2)
max_jerk            = 0.9;   % Maximum jerk          (m/s^3)
min_jerk            = -0.9;  % Minimum jerk          (m/s^3)
max_steer           = 0.8;   % Maximum steering      (rad)
min_steer           = -0.8;  % Minimum steering      (rad)
max_steer_rate      = 1.5;   % Maximum steering rate (rad/s)
min_steer_rate      = -1.5;  % Minimum steering rate (rad/s)
v_set               = 20;    % Driver set velocity   (m/s)
Measurement_noise   = 0.01;  % Measurement noise     (m/s)
PredictionHorizon   = 6;     % Prediction Horizon    (s)
ControlHorizon      = 2;     % Control Horizon       (s)
%% Create driving scenario
% The scenario name is a session file created by the Driving Scenario
    Designer App.
scenariosNames = {
    'Scenario1.mat',...          % scenarioId = 1
    'Scenario2.mat',...          % scenarioId = 1
    };
scenarioId = 2;
fileName   = 'laneFollowingScenario';

% The drivingScenario session file is converted to a drivingScenario object
% initial conditions of ego car and actor profiles
[scenario,egoCar,actor_Profiles] = helperSessionToScenario(scenariosNames{
    scenarioId});

% Save the driving scenario to the file to be read by Scenario Reader
helperSaveScenarioToFile(scenario,fileName);

%% Configuration of the simulation
% Initial condition for the ego car in ISO 8855 coordinates
v0_ego   = egoCar.v0;          % Initial speed of the ego car           (m/
    s)
x0_ego   = egoCar.x0;          % Initial x position of ego car          (m)
y0_ego   = egoCar.y0;          % Initial y position of ego car          (m)
yaw0_ego = egoCar.yaw0;        % Initial yaw angle of ego car           (
    rad)

```

```matlab
% Convert ISO 8855 to SAE J670E coordinates
y0_ego = -y0_ego;
yaw0_ego = -yaw0_ego;

% get the simulation stop time from scenario
load(fileName)
simStopTime = vehiclePoses(end).SimulationTime;

% Position and velocity selectors.
posSelector = [1,0,0,0,0,0; 0,0,1,0,0,0]; % Position selector   (N/A)
velSelector = [0,1,0,0,0,0; 0,0,0,1,0,0]; % Velocity selector   (N/A)

%% Ego Car Parameters
m   = 1575;    % Total mass of vehicle                              (kg)
Iz  = 2875;    % Yaw moment of inertia of vehicle                   (m*N*s
    ^2)
a   = 1.2;     % Longitudinal distance from c.g. to front tires     (m)
b   = 1.6;     % Longitudinal distance from c.g. to rear tires      (m)
h   = 0.5;     % Vertical distance from center of mass to axle plane (m)
Cf  = 19000;   % Cornering stiffness of front tires                 (N/rad
    )
Cr  = 33000;   % Cornering stiffness of rear tires                  (N/rad
    )
ds = 0.9;      % Distance between c.g and camera                    (m)
tau = 0.2;     % Time constant

%% Simulation settings

v0 = v0_ego;             % Initial ego velocity
Duration = 75;           % Simulation duration
t = 0:Ts:Duration;       % Time vector

%% Bus Creation
% Load the Simulink model
modelName = 'LaneFollowingNMPC2';
wasModelLoaded = bdIsLoaded(modelName);
if ~wasModelLoaded
    load_system(modelName)
end

% Create buses for lane sensor and lane sensor boundaries
createLaneSensorBuses;

% load the bus for scenario reader
blk=find_system(modelName,'System','helperScenarioReader');
s = get_param(blk{1},'PortHandles');
```

```
get(s.Outport(1),'SignalHierarchy');


%% Design Nonlinear Model Predictive Controller

% Create a NMPC object
nlobj = nlmpc(7,3,'MV',[1 2],'MD',3,'UD',4);
nlobj.Ts = time_gap;
nlobj.PredictionHorizon = PredictionHorizon;
nlobj.ControlHorizon = ControlHorizon;

% Specify the state function for the nonlinear plant model and its
% Jacobian.
nlobj.Model.StateFcn = @(x,u) LaneFollowingStateFcn(x,u);
nlobj.Jacobian.StateFcn = @(x,u) LaneFollowingStateJacFcn(x,u);

% Jacobians
nlobj.Model.OutputFcn = @(x,u) [x(3);x(5);x(6)+x(7)];
nlobj.Jacobian.OutputFcn = @(x,u) [0 0 1 0 0 0 0;0 0 0 0 1 0 0;0 0 0 0 0 1
    1];

% Set the constraints for manipulated variables.
nlobj.MV(1).Min = min_ac;
nlobj.MV(1).Max = max_ac;
nlobj.MV(1).RateMin = min_jerk*Ts;
nlobj.MV(1).RateMax = max_jerk*Ts;
nlobj.MV(2).Min = min_steer;
nlobj.MV(2).Max = max_steer;
nlobj.MV(2).RateMin = min_steer_rate*Ts;
nlobj.MV(2).RateMax = max_steer_rate*Ts;

% Set the scale factors.
nlobj.OV(1).ScaleFactor = v_set;                % Typical value of
    longitudinal velocity
nlobj.OV(2).ScaleFactor = 3.6;                  % Range for lateral
    deviation
nlobj.OV(3).ScaleFactor = 1;                    % Range for relative yaw
    angle
nlobj.MV(1).ScaleFactor = (max_ac-min_ac);      % Range of acceleration
nlobj.MV(2).ScaleFactor = 2*max_steer;          % Range of steering angle
nlobj.MD(1).ScaleFactor = 1;                    % Range of Curvature

nlobj.Weights.OutputVariables = [0.2 1 0];

% Penalize jerk more for smooth driving experience.
nlobj.Weights.ManipulatedVariables = [0 0];
nlobj.Weights.ManipulatedVariablesRate = [2 1];

%% Validate the nonlinear controller object

x0 = [0.1 0.5 25 0.1 0.1 0.001 0.5];
u0 = [0.125 0.4];
ref0 = [22 0 0];
md0 = 0.1;
validateFcns(nlobj,x0,u0,md0,{},ref0);
```

6.3 Controllers comparison

In this chapter a PID controller and a nonlinear MPC have been presented, together with their simulated behaviour in a two-lanes high-curvature road. By looking at Figures 6.10 and 6.20, it's evident that the second one gives better performances in terms of lateral deviations and steering rates. This was expected from the beginning and the goal of moving from PID to MPC controller was to improve performance and car handling in high curvature roads at a relatively high speed, up to 20 m/s.

The PID controller lacks the optimization and prediction of the further steps. This causes the car to depart from the center of the lane considerably during sharp curves and the controller correction becomes very high in that case, specially if the proportional and derivative coefficients are high. This produces high steering and steering rates that are unwanted but necessary at high speeds to maintain the car on track. To counteract this critical problem, a speed-tuning knob has been implemented to give the user the possibility to reduce the speed of the car depending on the street configuration. The speed variation follows a first order dynamics and causes the parameters of the PID controller to vary discretely. This solution works well and makes this technology suitable for a simple control. The simplicity is due to the absence of automatic throttle control, that requires a manual intervention. By the way, this is an effective solution for applications that have a low speed hardware.

With NMPC instead, the performances are better. They are obtained also by the longitudinal control embedded, that causes the car to decelerate in points where the lateral deviation would increase (i.e. curves). By imposing the weights higher for the lateral offset compared to the proximity to the set velocity, the optimizer is led to decelerate in order to promote center lane keeping more than speed keeping. The result are: the maximum lateral deviation is halved, the maximum steering has been decreased from 1.2 to 0.8 rad and the maximum steering rate from 5 to 1.5 rad/s. This traduces in much better comfort for the passengers in a real scenario and in better driveability. All this improvements are brought by the optimization and the receding horizon principle. That is, the future state of the nonlinear model are predicted by the optimizer and filtered with the Kalman Filter. Then only the first control input is applied and this improves the robustness of the system from noise and unmodelled dynamics. Being the controller nonlinear and the computational costs high, low prediction and control horizons have been chosen. By the way, they guarantee a good performance and are necessary for real time applications.

Note that in this work only model-in-the-loop simulations have been addressed. So the simulations aren't running on a specific hardware through compiled code but in Simulink on a personal computer. The NMPC is known for its high computing power demands and only with the rapidly reduction of computing platforms costs of the last years has found applications in real-time embedded systems. Further works shall verify real-time capabilities of controllers like the one presented in this work.

6.4 End-to-end control with Convolutional Neural Networks

Artificial intelligence is assuming an increasing importance in the field of autonomous driving for the improvement of the computing platforms and the accuracy and efficiency of the algorithms themselves. During the development of this thesis the huge world of deep learning has been addressed. The use of *Convolutional Neural Networks* (CNN) in objects recognition is tempting for the simplicity of the designing phase and the accuracy of the results. The counterpart of this is the big amount of computing power required for the training of the network. Nevertheless, the lowering of computational costs makes this approach attractive. In particular, end-to-end in this case means the use of raw data for training a neural network with the aim to map raw pixels from a single front facing camera directly to steering commands [28]. This section is organized in the following way: (1) the basic theory about convolutional neural networks is presented, (2) the model developed by NVIDIA and highly studied by researchers in this field is presented, (3) the algorithm written ad hoc for getting training and validation data from CARLA is shown.

6.4.1 Basic theory about CNN

A convolutional neural network is a class of deep neural networks highly used in image classification because they take into consideration the spatial structure of pixels. The main difference from multilayer fully connected perceptrons is the extraction of meaningful information and patterns from images through convolutions of the image with kernels whose weights are tuned in the training phase. Convolutional networks exploit spatially local correlation by connecting each neuron to only a small number of other neurons. The amount by which we are shifting the kernel at every operation is known as the *stride*. The convolution allows a reduction of the number of neurons and of the complexity of the network, by reducing the number of free parameters. The result of this convolution is another "image" that is more abstract but also more meaningful and is named *feature map*. Each feature map contains the information of a detected image feature with a specific kernel. The smaller the stride, the bigger will be the corresponding feature map. A layer that does this operation is called *convolutional layer* and the size of the kernel for each layer has to be set a priori. In case of RGB images the kernel will be NxNx3, with N choosen a priori. To reduce the number of parameters in computations in the network, the so called *pooling layers* can be inserted. This layer shortens the training time and is useful to avoid *overfitting*. A CNN usually have many convolutional and pooling layer in cascade to remove all the unnecessary information and extract the most important patterns that will be processed by the fully connected layers. The weights of the kernels are tuned during the training process, so during this phase the network learns which are the main features to be detected in an input image in order to classify it. Different filters are able to detect different features in an image and the translational invariance property of the convolution allows to use a single filter to detect features in any area of the image. To make the output non-linear, an activation function will be applied to the output of the convolutional layer and of the fully connected layers. This could be the *ReLU* activation function, which applies the non-saturating and continuous function $f(x) = max(0, x)$. For a detailed explanation of layers, activation functions and optimizers, consult the Keras guide or any book about deep learning, as [29].

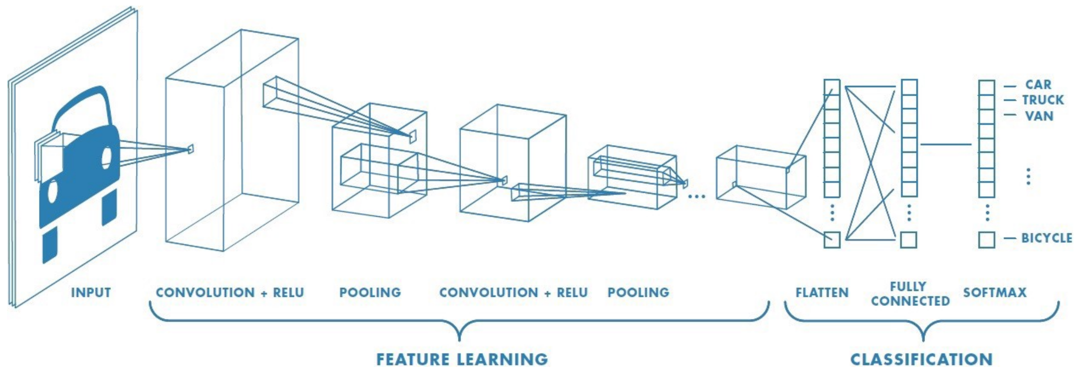


Figure 6.21: Architecture of a CNN [30]

6.4.2 NVIDIA model

The NVIDIA model [28] is a simple but quite effective deep learning model that can be used to automatically steer a car by using the information coming from the raw pixels of the frames coming from a single camera positioned in the front of the vehicle. As reported by the authors in [28]:

> With minimum training data from humans the system learns to drive in traffic on local roads with or without lane markings and on highways. It also operates in areas with unclear visual guidance such as in parking lots and on unpaved roads. The system automatically learns internal representations of the necessary processing steps such as detecting useful road features with only the human steering angle as the training signal. We never explicitly trained it to detect, for example, the out-line of roads. Compared to explicit decomposition of the problem, such as lane marking detection, path planning, and control, our end-to-end system optimizes all processing steps simultaneously. We argue that this will eventually lead to better performance and smaller systems. Better performance will result because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria, e. g., lane detection.

This model has become so popular for its simplicity that is cited in many articles and web courses about self driving cars [1]. The training data come from three camera positioned on the left, the center and the right of the front of the ego car. Each triple of camera's frames captured at the same time is labelled with a steering value as shown in Figure 6.22. Then the network can be driven with a single camera positioned in front of the vehicle as shown in Figure 6.23.

All the training details can be found in [28], some in Figure 6.24. The contribution of this thesis in this case concerns the data collection phase through CARLA, shown in the next subsection.

[1]e.g. Self-Driving Car Engineer Nanodegree, Udacity

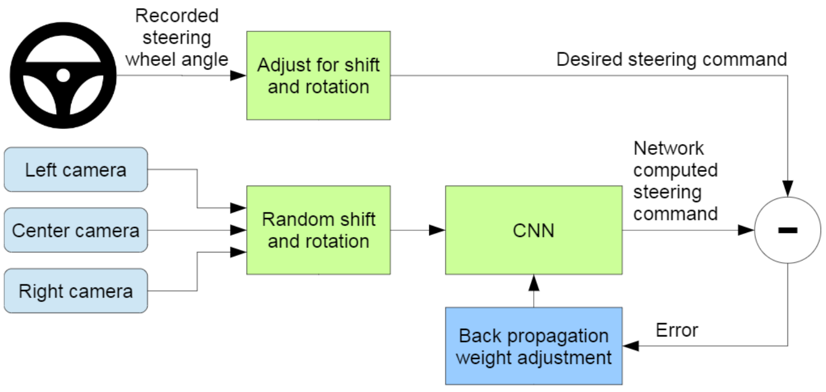

Recorded steering wheel angle
Adjust for shift and rotation
Desired steering command
Left camera
Center camera
Right camera
Random shift and rotation
CNN
Network computed steering command
Back propagation weight adjustment
Error

Figure 6.22: Training of the neural network [28]

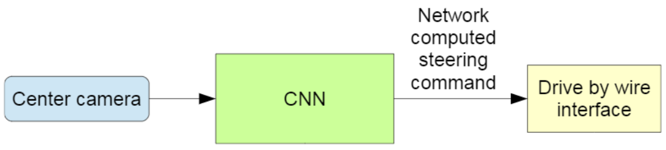


Figure 6.23: Once trained, the neural network is driven by images coming from a single front-facing centred camera to compute steer values [28]

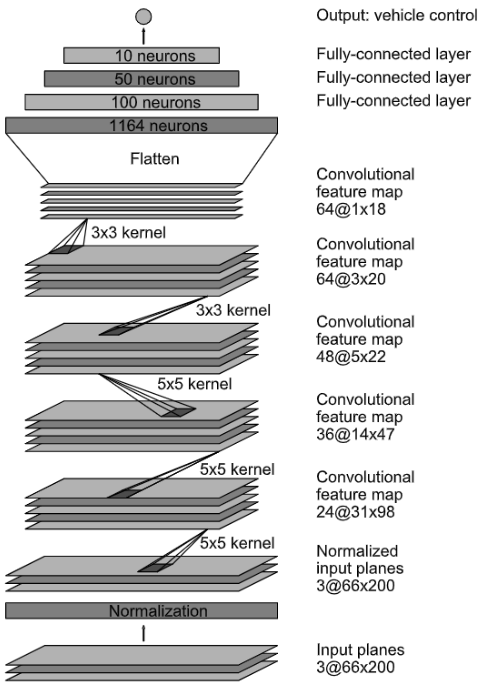


Figure 6.24: NVIDIA model's layers [28]

6.4.3 Data collection and preprocessing

For this part, CARLA 0.8.2 has been used because it is the stable version and at the beginning of this work version 0.9.5 presented several bugs. The Python API is quite different but easy to understand. The goal of data collection is to save the images captured by the three RGB cameras in three different folders. For each folder, each image is named based on its frame number. A steering value for each frame is saved in a python list and, after the end of data collection, the list is saved on a file using `pickle`. It is possible to chose the number of episodes and the number of frames per episodes, depending on the trade-off between training time and network accuracy. For each episode, the weather is selected randomly to have variability of the training data and give to the network robustness to work on all the virtual weather conditions. The `print_measurements` function prints on the terminal the position and speed of the cars, the number of collisions, the number of agents and the intersections with the other lane and offroad. The `run_carla_client` function is called at each iteration, until the last frame of the last episode is considered. This function is a sample already present in CARLA python API and has been modified to be used for NVIDIA model's data collection.

```python
# Copyright (c) 2017 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).

def run_carla_client(args):
    steering_angles = []
    # We assume the CARLA server is already waiting for a client to connect at
    # host:port. To create a connection we can use the `make_carla_client`
    # context manager, it creates a CARLA client object and starts the
    # connection. It will throw an exception if something goes wrong. The
    # context manager makes sure the connection is always cleaned up on exit.
    with make_carla_client(args.host, args.port) as client:
        print('CarlaClient connected')

        for episode in range(0, NUMBER_OF_EPISODES):
            # Start a new episode.
            # Create a CarlaSettings object. This object is a wrapper around
            # the CarlaSettings.ini file. Here we set the configuration we
            # want for the new episode. The weather is selected randomly to
            # add variety to training images and generalize the possible
            # scenarios. Synchronous mode is activated
            settings = CarlaSettings()
            settings.set(
                SynchronousMode=True,
                SendNonPlayerAgentsInfo=False,
                NumberOfVehicles=15,
                NumberOfPedestrians=20,
                WeatherId=random.choice([1, 3, 7, 8, 14]),
                QualityLevel=args.quality_level)
            settings.randomize_seeds()

            # Now we want to add three cameras to the player vehicle.
            # We will collect the images produced by these cameras every
            # frame.

            # The center camera.
            camera0 = Camera('CameraRGB_Center')
            # Set image resolution in pixels.
            camera0.set_image_size(800, 600)
            # Set its position relative to the car in meters.
            camera0.set_position(2.20, 0, 0.6)
            camera0.set_rotation(1, 0, 0)
            settings.add_sensor(camera0)

            # The right camera.
            camera1 = Camera('CameraRGB_Right')
            camera1.set_image_size(800, 600)
            camera1.set_position(2.20, 0.8, 0.6)
            camera1.set_rotation(1, 28, 0)
            settings.add_sensor(camera1)

            # The left camera.
            camera2 = Camera('CameraRGB_Left')
            camera2.set_image_size(800, 600)
            camera2.set_position(2.20, -0.8, 0.6)
            camera2.set_rotation(1, -28, 0)
            settings.add_sensor(camera2)

            # Now we load these settings into the server. The server replies
            # with a scene description containing the available start spots for
            # the player. Here we can provide a CarlaSettings object or a
            # CarlaSettings.ini file as string.
            scene = client.load_settings(settings)

            # Choose one player start at random.
            number_of_player_starts = len(scene.player_start_spots)
            player_start = random.randint(0, max(0, number_of_player_starts - 1))

            # Notify the server that we want to start the episode at the
            # player_start index. This function blocks until the server is ready
            # to start the episode.
            print('Starting new episode...')
            client.start_episode(player_start)

            # Iterate every frame in the episode.
            for frame in range(0, FRAMES_PER_EPISODE):

                # Read the data produced by the server this frame.
                measurements, sensor_data = client.read_data()

                # Print some of the measurements.
                print_measurements(measurements)
```

```
                # Set autopilot to collect data for the CNN autonomously,
                # without the human supervision
                control = measurements.player_measurements.autopilot_control

                # Save the images to disk if requested.
                for name, measurement in sensor_data.items():
                    filename = args.out_filename_format.format(episode, name, frame)
                    measurement.save_to_disk(filename)

                # Append steer values sent by the autopilot to the list
                steering_angles.append(control.steer)
                client.send_control(control)

# Save the list in a file that will be loaded during training
    with open('steering', 'wb') as fp:
        pickle.dump(steering_angles, fp)
```

Before the training of the neural network with these data, we must pre-process them. In fact, as visible in Figure 6.26a, most of the time the car goes straight or it's still at a red semaphore. In these cases, the steer value is zero. In order to avoid unbalanced training data, part of the data labelled with a 0 steer need to be removed from the dataset. The main cause is that they are too much. The function `clean_labels` uses the matplotlib.python python package to show the histogram of the number of data collected for each steer value interval. Then it reduces the samples of each bin of the histogram to `samples_per_bin`, if higher. In order to avoid the elimination of subsequent labelled frames, that could refer to a specific situation that should be captured by at least one frame, the list of frame's indices to be deleted is shuffled. Then the only `samples_per_bin` indices are kept, while the others are put in `remove_list` to be eliminated. For example, in the case of a number of frames per episode equal to 1000 and only one episode, if the maximum number of samples per bin is equal to 40 the number of the removed labelled data from the set is 802. Obviously, for a good training process, much more data are needed. The remaining data are plotted again in an histogram, visible in Figure 6.26b.

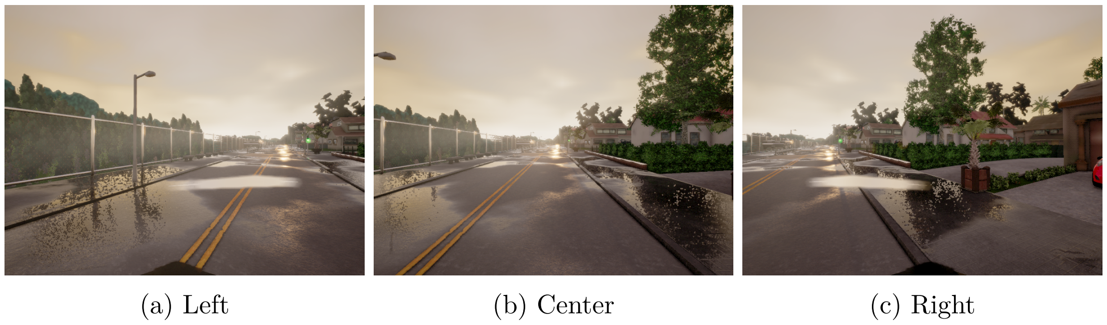

(a) Left (b) Center (c) Right

Figure 6.25: Camera frames for the training of the NVIDIA model.

Then the function `load_data` pre-precess the data by cleaning the redundant labels and their respective frames, in order to avoid biases on the neural network. Finally, it applies

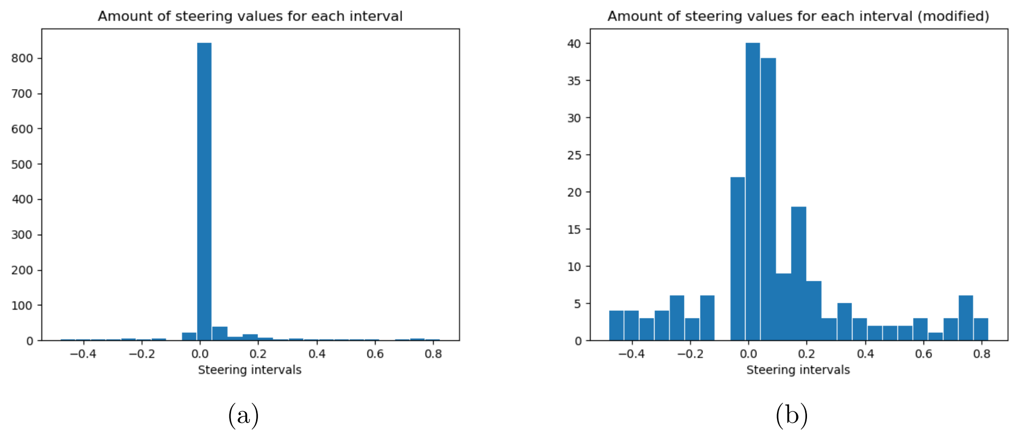


Figure 6.26: Histogram of steering values before (a) and after (b) the cleaning process

to the labelled dataset the `train_test_split` function to subdivide it into training and validation datasets. Now the NVIDIA model can be trained. Unfortunately, for lack of computing capability, the training process has not been done, although the NVIDIA model has been modified to reduce the training time.

```python
def clean_labels(data):
    """
    It shows the histogram of the number of image frames for each steer
    value interval. Then for each bin it creates a list of indices related
    to frames, it shuffle it and it adds to the remove_list these indices.
    Then it removes the values of steer corresponding to these indices from
    the list of steer values
    """
    num_bins = 25
    samples_per_bin = 40
    hist, bins = np.histogram(data, num_bins)
    center = (bins[:-1]+bins[1:])*0.5
    plt.bar(center, hist, width=0.05)
    plt.xlabel('Steering intervals')
    plt.title('Amount of steering values for each interval')
    plt.show()
    print("Total data:", len(data))
    remove_list = []
    for j in range(num_bins):
        list_ = []
        for i in range(len(data)):
            if data[i] >= bins[j] and data[i] <= bins[j+1]:
                list_.append(i)

        list_ = shuffle(list_) # shuffle to avoid to remove consequent frames
        list_ = list_[samples_per_bin:] # Save indices of data to be removed
        remove_list.extend(list_) # Put these data indices in a list
    print('removed', len(remove_list))
    # Remove the data on the list
    for i in sorted(remove_list, reverse=True):
        del data[i]

    # Plot the histogram after data reduction
    print("Remaining:", len(data))
    hist, _ = np.histogram(data, num_bins)
    plt.bar(center, hist, width=0.05)
    plt.xlabel('Steering intervals')
    plt.title('Amount of steering values for each interval (modified)')
    plt.show()

def load_data(args):
    """
-   Load and preprocess training data and split it into training and
    validation set. After having cleaned data, the function associate all
    the images of the reduced set to their steer labels. It saves in another
    variable the original labels and use this information to check if a triple
    of (left,center,right) images corresponds to the label it had before the
    cleaning process.
-   Then it reshape the array of triples and vertically stacks them for passing
    them as arguments to the train_test_split function
-   Finally it returns the train and validation datasets and labels
    """
    with open ('steering', 'rb') as fp:
        y = pickle.load(fp)


    yb = y[:]
    clean_labels(y)

    path_root = '/home/marco/CARLA_08/PythonClient/_out/'
    img_dict = {
        'center': 'CameraRGB_Center',
        'left': 'CameraRGB_Left',
        'right': 'CameraRGB_Right'
    }
    X = np.array([])
    # If the number of episodes is different from 5, update the following list
    for i in ['episode_0000', 'episode_0001', 'episode_0002',
            'episode_0003', 'episode_0004']:
        path = path_root + i
        X_temp = []
        for t in img_dict.keys():
            y_mod_iter = iter(y)
            y_iter = iter(yb)
            a = next(y_iter)
            b = next(y_mod_iter)
            for infile in sorted(glob.glob(os.path.join(path + '/' + img_dict[t],
                                    '*.png'))):

                # If the steer values (the labels) coincide, append the triple
                # of frames in the variable X_temp
                if a == b:
                    try:
                        X_temp.append(infile)
                        a = next(y_iter)
                        b = next(y_mod_iter)
                    except:
                        print("Stop loading")
                        break
                else:
                    a = next(y_iter)

        # Reshape the 1D array to have an array of 3D arrays [left, center, right]
        # This will be the dataset that will be divided in training and validation
        X_t = np.array(X_temp, dtype='object')
        X_t = np.reshape(X_t, (3, -1))
        X_t = X_t.T
        X = np.vstack((X, X_t)) if X.size else X_t

    assert(X.shape[0] == len(y)), """The number of frames is not equal
                                    to the number of steering values"""

    X_train, X_valid, y_train, y_valid = train_test_split(X, y,
                                                        test_size=args.test_size,
                                                        random_state=0)

    return X_train, X_valid, y_train, y_valid
```

Chapter 7

Conclusions and future works

The goal of this thesis was the development and the simulation of control algorithms for autonomous vehicles.

In this work some methodological aspects of lane keeping algorithms and some practical aspects related to their implementation on different simulation environments have been addressed.

From the methodological point of view, the results extend to the design of a PID controller, for its simplicity and low computational power requirements, and the design of a nonlinear model predictive controller, for its accuracy in predicting the state of the system. Both the dynamical models, that have been used for the design of these controllers, are well-known in the specific literature and are derived from the bicycle model.

The PID controller has shown an unexpected performance in lane keeping at speeds up to $20\,m/s$, that is a very high speed if we consider the kind of track that has been used during testing. This thanks to a discrete adaptive strategy, that allows to use different classical PID controllers for different longitudinal speeds. In particular, four different controllers have been tuned to work from quasi-zero up to $20\,m/s$, in four equally spaced ranges. In this way, the dynamical system can be considered linear in its speed range and the results show that this approximation is not drastic. Although this control strategy resulted in an effective lane keeping, it misses completely the goal of the passengers' comfort. Moreover, the fact that the dynamical model used neglects a good part of vehicle dynamics, suggests that in real systems this technology can find applications, and it does, only on low curvature roads with quasi-constant speed.

Higher expectations are directed towards the use of Nonlinear Model Predictive Control for lane keeping. The possibility to include longitudinal dynamics inside the vehicle model brings the algorithm closer to how a control algorithm for autonomous driving should be. In fact, not only the steer is controlled autonomously but also the acceleration and deceleration. With respect to the PID, this algorithm improves a lot the performances, minimizing the lateral offset and yaw error values while reducing considerably the jerk and steering rates during the manoeuvres. The kind of optimization required by this control strategy is non-convex, in general not computationally efficient. In order to counteract this characteristic, low prediction and control horizons have been adopted. In this way the computational burden for the CPU has been reduced considerably.

Another algorithm this thesis introduces is a lane detection algorithm, written in Python with the help of Numpy and OpenCV packages. This algorithm is able to extract the information needed by the NMPC from camera frames. It detects lane lines, transforms the resulting image in a top-view image and subdivides it into two sections, the near view and the far view. Then it applies Hough transform to the first section and least-squares to identify the parameters of a parabolic model for the second section. The algorithm works well when the car moves slow and the road has low curvature. If it's not the case, the identification of parameters of the far view section gets worse quickly, in particular at medium and high velocities. The main reason for this behaviour is the way the lane lines are searched in the image, that is simply divided vertically into three equispaced regions. The algorithm searches for the left line in the left region, for the center line in the center region and the same for the right line. The result is that when the curvature is high the lines invade the other regions and the algorithm considers their pixels outside the region for least-squares fitting. A better solution, that should be addressed in case of a further development of this work, is the use of a Kalman filter or a similar method to guide the search of the pixels associated to lane lines, by taking into account the information coming from previous measurements. In subsection 4.2.5 a possible model for the Kalman filter proposed in [12] is shown.

From the practical point of view related to simulation environments, two of them have been explored and tested. The first is the Automated Driving Toolbox provided by Matlab. Chapter 2 has been devoted to this tool, that allows to build driving scenarios, get lane lines information already processed to be used mathematically and an integration with Simulink that allows to test the algorithms directly in the driving scenario with the Bird's-Eye view mode. The merit of the toolbox is the possibility to access high-level information without computer vision algorithms and so to focus on controller design without looking too much at all the other aspects of a complete algorithm for autonomous driving. The main weakness is the lack of a photorealistic and ready-to-use urban environment, that limits the possibility to algorithms in the family of the ones presented here. The vision part is simulated through the `VisionDetectionGenerator` block and not by a camera that produces raw data, like the real ones. This toolbox is not made for these purposes.

A software closer to this philosophy is CARLA, an open source photorealistic simulator. It offers a Python API for the server-client communication that has been used to code a lane keeping algorithm with the aim of autonomously control steer, throttle and brake of the ego vehicle in CARLA Simulator. The NMPC, designed with the help of Matlab, has been exploited also by this algorithm in Python by using the Matlab-Python integration. In this way, the focus has been devoted to the understanding of the Python API and the OpenCV package for image processing and to the development of the structure of the algorithm. It has been divided in logical modules to give the possibility to change the various parts independently. The pipeline works correctly and the connection with Matlab is useful to retrieve data structures from the workspace and call Matlab functions on Python. By the way, an interesting development of this thesis could be to improve the model fitting algorithm inside the `RoadModel` class.

Finally, algorithms for the collection of data and the training of a Convolutional Neural Network model found in literature have been written, but the training has not been completed for lack of computational power.

References

Cited books and papers

[1] Reuters and Ipsos. *Autonomous Vehicles Readiness survey.* 2018. URL: `http://fingfx.thomsonreuters.com/gfx/rngs/AUTO-SELFDRIVING-SURVEY/010060NM16V/AUTO-SELFDRIVING-SURVEY.jpg`.

[2] Douglas A. Reece and Steven Shafer. *A Computational Model of Driving for Autonomous Vehicles.* Research Report. Carnegie Mellon University, 1991.

[3] Canale Massimo. "Technologies for Autonomous Vehicles". Slides at support of the course. 2019.

[4] Chiu-Feng Lin, A. Galip Ulsoy, and David J. LeBlanc. "Lane Geometry Perception and the Characterization of Its Associated Uncertainty". In: *Journal of Dynamic Systems Measurement and Control* (1999).

[5] J. McCall and M. Trivedi. "Video-based lane estimation adn tracking for driver assistance: Survey, System and evaluation". In: *IEEE Transactions on Intelligent Transportation Systems.* Vol. 7. 1. Mar. 2006, pp. 20–37.

[6] MathWorks. *Automated Driving Toolbox™ Getting Started Guide.* 2019.

[7] MathWorks. *Automated Driving Toolbox™ User's guide.* 2019.

[8] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning.* 2017, pp. 1–16.

[9] CARLA team. *CARLA Documentation.* 2019. URL: `https://carla.readthedocs.io/en/latest/`.

[10] Scott Drew Pendleton et al. "Perception, Planning, Control, and Coordination for Autonomous Vehicles". In: *Machines* (2017).

[11] Xinxin Du and Kiong Tan Kog. "Comprehensive and Practical Vision System for Self-Driving Vehicle Lane-Level Localization". In: *IEEE transactions on image processing* (2016).

[12] Sayanan Sivaraman and Mohan Manubhai Trivedi. "Integrated Lane and Vehicle Detection, Localization and Tracking: A Synergistic Approach". In: *IEEE Transactions on Intelligent Transportation Systems.* Vol. 14. 2. June 2013.

[13] M. Meuter et al. "A novel approach to lane detection and tracking". In: *12th Int. IEEE ITSC.* Oct. 2009, pp. 1–6.

[14] T. Veit et al. "Evaluation of road marking feature extraction". In: *11th Int. IEEE ITSC.* Oct. 2008, pp. 174–181.

[15] Byambaa Dorj and Deok Jin Lee. "A Precise Lane Detection Algorithm Based on Top View Image Transformation and Least-Square Approaches". In: *Journal of Sensors* (2016).

[16] A. Galip Ulsoy, Huei Peng, and Melih Cakmakci. *Automotive control systems.* 2012.

[17] Camillo J. Taylor et al. "A Comparative Study of Vision-Based Lateral Control Strategies for Autonomous Highway Driving". In: *IEEE* (1999).

[18] MathWorks. *Vehicle Dynamics Blockset™ User's Guide*. 2018.

[19] Wuweu Chen, Hansong Xiao, et al. *Integrated Vehicle Dynamics and Control*. 1st ed. John Wiley & Sons, 2016. Chap. 5.

[20] Huei Peng and Masayoshi Tomizuka. *Lateral Control Of Front-wheel-steering Rubber-tire Vehicles*. Research Report. California Partners for Advanced Transit and Highways, UC Berkeley, 1990.

[21] Nicola De Val and Andrea Fuso. "Model Predictive Control for an Autonomous Vehicle". Master thesis. Politecnico di Milano, 2013.

[22] Pan Zhao et al. "Design of a Control System for an Autonomous Vehicle Based on Adaptive-PID". In: *International Journal of Advanced Robotic Systems* (2012).

[23] MathWorks. *Model Predictive Control™ User guide*. 2018.

[24] Valerio Turri, Francesco Borrelli, et al. "Linear Model Predictive Control for Lane Keeping and Obstacle Avoidance on Low Curvature Roads". In: *Proceedings of the 16th International IEEE Annual Conference on Intelligent Transportation Systems (ITSC 2013)*. Ed. by IEEE. 2013.

[25] Chuanyang Sun et al. "Design of a Path-Tracking Steering Controller for Autonomous Vehicles". In: *energies* (2018).

[26] Yasuchika Mori and Fitri Yakub. "Autonomous Ground Vehicle of Path Following Control through Model Predictive Control with Feed Forward Controller". In: *12th International Symposium on Advanced Vehicle Control*. Ed. by Society of Automotive Engineers of Japan. 2014.

[27] Vito Cerone, Mario Milanese, and Diego Regruto. "Combined Automatic Lane-Keeping and Driver's Steering Through a 2-DOF Control Strategy". In: *IEEE transactions on control systems technology* (2008).

[28] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars". In: *CoRR* abs/1604.07316 (2016). arXiv: `1604.07316`. URL: `http://arxiv.org/abs/1604.07316`.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[30] MathWorks. *Deep Learning Toolbox™ User's Guide*. 2019.

Other suggested readings

[31] Dean Pomerleau. "RALPH: rapidly adapting lateral position handler". In: *Proceedings of the Intelligent Vehicles '95. Symposium*. Ed. by IEEE. Detroit, MI, USA, 2002.

[32] Ernst D. Dickmanns and Birger D. Mysliwetz. "Recursive 3-D Road and Relative Ego-State Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14 (2 Feb. 1992): *Special issue on interpretation of 3-D scenes. part II*. Ed. by USA IEEE Computer Society Washington DC, pp. 199–213.

[33] M. Bertozzi et al. "Artificial vision in road vehicles". In: *Proceedings of the IEEE* 90 (7 July 2002). Ed. by IEEE, pp. 1258–1271.

[34] M. Bertozzi and A. Broggi. “GOLD: A parallel real-time stereo vision system for generic obstacle and lane detection”. In: *IEEE Trans. on Image Processing* 7 (1 Jan. 1998). Ed. by IEEE, pp. 62–81.

[35] Giuseppe Calafiore and Laurent El Ghaoui. *Optimization Models*. Cambridge University Press, 2014. Chap. 15.

[36] V. Cerone, A. Chinu, and D. Regruto. “Experimental results in vision-based lane keeping for highway vehicles”. In: *Proceedings of the American Control Conference Anchorage*. 2002.

[37] Huei Peng and Masayoshi Tomizuka. “Preview Control for Vehicle Lateral Guidance in Highway Automation”. eng. In: *1991 American Control Conference*. IEEE, 1991, pp. 3090–3095. ISBN: 0879425652.

[38] Olivér Törő, Tamás Bécsi, and Szilárd Aradi. “Design of Lane Keeping Algorithm of Autonomous Vehicle”. In: *Periodica Polytechnica Transportation Engineering* (2016).

Structure of project files folders A

Matlab

In this appendix are listed the files contained in each folder of this thesis project. The role of each *file* or **folder** inside the project is explained briefly.

Software requirements: Matlab with Automated Driving Toolbox, Vehicle Dynamics Blockset, Control System Toolbox and Model Predictive Control Toolbox.

A.1 PID

- **Images**: contains the plots of signals logged during simulations, bode plots, step response and the pole-zero map.
- **slprj**: contains project configuration files needed by Matlab.
- *PID.m*: is the main Matlab script, that has to be run to set all the variables and data structures needed during the simulations.
- *PIDsim.slx*: is the Simulink file that has to be open after the main script has been run. It is preferable to run the simulation from the Bird's-eye view mode, after having clicked on `Find signals`. It is suggested to open also the Simulation Data Inspector mode to view the plots of lateral deviation, steering angle etc.
- *CreateDrivingScenario.m*: contains an example of how to create a driving scenario directly with a script rather than with the Driving Scenario Designer application. The results are equivalent, the application is easier to use for road modelling but the Matlab API is well documented in the help and is more straightforward in setting camera and vehicle parameters.
- *helperSessionToScenario.m*: it extracts from the driving scenario created by the application or the API the scenario data (in this case called `Scenario1.mat`), the ego car parameters and the other actors' profiles. Then this data structures are saved to a file named `laneFollowingScenario.mat`.
- *createSimulinkScenarioData.m*: it gets the data from the built scenario and create data in a format compatible with the Scenario Reader block. The new scenario will

not include ego vehicle because ego will be controlled by the simulation. Finally it saves the Scenario to a file format used by Scenario Reader.

- *CreateLaneSensorBuses.m*: it is a Matlab function, used in the help by many Autonomous Driving examples, that creates a bus type named LaneSensor, used by the Vision Detection Generator block to output lane lines measurements.
- *knobTurned.m*: it is a callback function that is executed each time the speed knob, used to change the speed of the car during PID lateral control, is turned. It changes accordingly the longitudinal speed value and the corresponding sampling time in the workspace.
- *G1,G2,G3,G4*: they are the plant transfer functions, computed through the vehicle state-space model at speeds of 5-10-15-20 m/s respectively.
- *C1,C2,C3,C4*: they are the controller transfer functions, whose coefficients has been determined through the `pidtune` function and then tuned through a trial-and-error procedure.

A.2 NMPC

- **slprj**: contains project configuration files needed by Matlab.
- *LaneFollowingUsingNMPC2.m*: it is the main script that has to be run to set all the variables and data structures needed during the simulations.
- *LaneFollowingNMPC2.slx*: it contains the Simulink model. As said for the PID controller section, is possible to access the Bird's-eye view and run from there the simulation.
- *LF.slx*: it is the Simulink subsystem that does the lane's center estimation and the control input computation.
- *createLaneSensorBuses.m*: same as in PID folder
- *helperSessionToScenario.m*: same as in PID folder
- *LaneFollowingEKFStateFcn.m and LaneFollowingEKFMeasFcn.m*: are the functions that contain the model of the Kalman filter used to estimate the states for the nonlinear MPC.
- *LaneFollowingStateFcn.m*: this function represents the nonlinear dynamical system state function. From the current state and the input vector, it computes the state derivative. It is the model used by the nonlinear controller for its predictions. The output function is passed to the controller structure directly in the main script.
- *LaneFollowingStateJacFcn.m*: this function represents the Jacobians of the nonlinear state function. It is needed by the nonlinear MPC to improve the efficiency of its computations, although it isn't mandatory. The Jacobian of the output equation is so simple that is passed to the controller structure directly in the main script.

- *Scenario1.mat,Scenario2.mat*: are two possible driving scenario that can be used to test the algorithm. To select one of the two, simply change the `scenarioId` value to 1 or 2. To add other scenarios, add the `.mat` file in the NMPC folder and add their names to the `scenariosNames` list.
- *laneFollowingScenario.mat*: is the file produced by the `helperSessionToScenario` function, that is used by the Scenario Reader block.
- The other files are Simulink cache files that have to remain untouched.

A.3 CKLA CARLA

- *cklaCARLA.m*: is the main Matlab script that is needed to set the workspace with the nonlinear controller and all the ego-vehicle, controller and simulation parameters. These data are then retrieved by the NMPC.py script that communicates with CARLA Simulator to compute the control input. This Matlab script must be run before the Python script and every first time in a Matlab session the command `matlab.engine.shareEngine` has to be passed in command window to share the workspace with Python.
- The other 4 functions have been already explained in the previous sections. The only difference is that in this case the dynamical model they represent has been slightly changed to include in vision dynamics the lookahead at which to compute lateral and yaw errors.

Structure of project files folders B

CARLA

Software requirements: Python 3.7 64-bit with Numpy, OpenCV, Matplotlib, PIL, imageio packages.

B.1 NMPC

Additional software requirements: CARLA 0.9.5 (latest), Matlab Engine for Python and Matlab licence with Model Predictive Control Toolbox.

- *NMPC.py*: it is the main Python script which contains the configuration of the simulation, the main algorithm and all the classes needed by it. It must be run after the CARLA Simulator. Use CARLA 0.9.5.
- *transform.py*: Contains two functions for the Inverse Perspective Mapping. The first is the `four_point_transform` and it is the one actually used in the algorithm. The points that it needs as arguments have been found with a trial-and-error procedure. The second is the `_top_view_transform` and it is based on the theory presented in subsection 4.2.3.

B.2 Data collection and preprocessing

Additional software requirements: CARLA 0.8.2 (stable); Keras, Tensorflow-gpu and Scikit-learn Python packages.

- *CollectData.py*: it is the script that configures the simulation and the cameras to acquire three images at each step and labelling them with the actual steering value. It saves the images, with the frame number as a name, in a folder with one subfolder for each camera. The steering values are saved in a file through `pickle`.
- *Train.py*: it is the script that does the preprocessing of the data and builds the training and validation datasets in a correct format for the training of the NVIDIA model through Keras.