

**Politecnico di Torino**

Master's Degree in Mechatronic Engineering



**Politecnico  
di Torino**

Master's Degree Thesis

**Model based automated DDT  
fallback for L4 Autonomous vehicles**

Relatori:

Prof. Massimo Violante

Eng. Claudio Russo (Bylogix srl)

Eng. Jose Isola (Bylogix srl)

Candidati:

La Ciacera Salvatore

April 2025



## **Summary**

The thesis hereby presented aims at implementing a safety strategy, capable of bringing an automated level 4 vehicle, to a minimum risk condition in case a performance related failure occurs, 2 different software's are used to generate the model-based algorithm, MATLAB and Qt using state machine xml.

Chapter 1 gives insight into what is autonomous driving, the difference between each level of autonomy in an automated vehicle and a brief explanation of the CARLA autonomous driving simulator.

Chapter 2 explains the design and the equipped sensors of the automated vehicle

Chapter 3 describes how the autonomous driving simulator CARLA can connect and interact with the different software used in this thesis.

Chapters 4 and 5 explains what the scenario definition for the implementation of the DDT-Fallback strategy is, and how it is modeled in MATLAB and Qt using State Chart XML, insight in each function that forms the algorithm is given.

Chapter 7 the most relevant results associated with the proposed DDT Fallback strategy are analyzed.

Chapter 6 concludes the work, summarizing the main objectives and results of this thesis.

## **Acknowledgements**

The author would like to thank the people of Bylogix srl, in special Eng. Claudio Russo and Eng. Jose Isola, for presenting their thesis idea, and giving me the opportunity to work alongside them in one of their projects, gaining valuable knowledge in the autonomous driving field, this thesis wouldn't have been possible without their technical expertise to guide me when it was needed the most.

Also, a special thanks goes to all the friends that became family in all these years at Politecnico, that made this journey more pleasant a fun along them.

A big thank you, goes to my family, to my mom, dad, and brother, that were always at my side, giving advice and supporting me in each possible way to continue and reach the end line that this work represents. A big hug goes to my dad, Angelo, that is not with me anymore, thanks for teaching me everyday the person I want to be, and making me think as an engineer since I was very young with your explanations about how things works.

*“Twenty years from now you will be more disappointed by the things that you didn't do than by the ones you did do. So throw off the bowlines. Sail away from the safe harbor. Catch the trade winds in your sails. Explore. Dream. Discover.” – Mark Twain*

# **Table of Contents**

Summary .....	3
Acknowledgements.....	4
List of figures.....	7
Acronyms.....	9
1. Introduction.....	10
1.1. What is autonomous driving .....	10
1.2. SAE Levels of automation .....	11
1.3. The Importance of simulation in Autonomous driving .....	13
1.4. Autonomous driving simulators.....	15
1.4.1. CARLA Simulator .....	15
1.4.2. The Simulator.....	16
2. Physical system design .....	18
2.1. Sensors.....	18
2.1.1. Lidar.....	18
2.1.2. GNSS sensor.....	19
3. Preparing for the development environment.....	20
3.1. Setting up CARLA with a virtual environment.....	20
3.2. Running CARLA with MATLAB.....	21
3.3. Running CARLA with Qt SCXML state machine.....	24
4. Problem definition for DDT fallback strategy .....	26
4.1.1. Scenario and road network.....	26
4.1.2. Failure.....	26
4.2. Fallback strategy.....	27
4.3. Testing platform.....	28
4.4. Different approaches for model-based DDT-fallback strategies .....	29
4.4.1. Model based modeling of the DDT-fallback in MATLAB .....	29
4.5. CARLA environment .....	30
4.6. PID Controllers.....	31
4.6.1. Longitudinal control.....	32
4.6.2. Lateral control .....	34
4.7. MATLAB functions for DDT-fallback.....	37

4.7.1.	MATLAB function get_route. ....	37
4.7.2.	MATLAB function dis2shoulder.....	38
4.7.3.	Getting Lidar data.....	39
5.	Qt SCXML model-based.....	42
5.1.	Functions.....	43
5.2.	How variables are sent and received with events.....	43
5.3.	Pid controllers.....	46
5.3.1.	Longitudinal control PID.....	46
5.3.2.	Lateral control PID.....	48
5.4.	Qt SCXML Functions for DDT fallback.....	49
5.4.1.	Distance to shoulder.....	49
5.4.2.	Get route.....	49
5.4.3.	Lidar data.....	50
6.	Results.....	52
7.	Conclusion and Future developments.....	55
7.1	Conclusion.....	55
7.2	Future developments.....	55
	Appendix A: CARLA environment.....	57
	Appendix B: SCXML code generated with Qt state machine.....	60
	Appendix C: Python code used to run CARLA with Qt.....	62
	Appendix D: Model based design in MATLAB.....	68
	Appendix F: MATLAB function for callback of lidar sensor through a python script .	69
	Bibliography.....	72

## **List of figures**

Figure 1.1 Bylogix VeGA autonomous vehicle .....	11
Figure 1. 2 SAE Levels of automation from [2].....	12
Figure 1. 3 V&V cycle validation process from [15].....	14
Figure 1. 4 CARLA Simulator Logo from [4]. .....	15
Figure 1. 5 Carla client-server architecture from [4] .....	16
Figure 1. 6 Carla development build in Unreal engine 4 .....	17
Figure 2. 1 Lidar in an autonomous vehicle from [16].....	18
Figure 2. 2 Point cloud data of a Lidar sensor from [4].....	18
Figure 3. 1 Development environment. ....	21
Figure 3. 2 MATLAB 2021 running python 3.7.....	22
Figure 3. 3 Result of running the code in MATLAB. ....	23
Figure 3. 4 Qt running in virtual environment with python 3.7.....	24
Figure 3. 5 Example of code execution using Qt. ....	25
Figure 4. 1 Dual carriageway with emergency lanes.....	26
Figure 4. 2 Flow chart for DDT fallback strategy, from [6] .....	27
Figure 4. 3 DDT fallback scenarios, from [18]. ....	28
Figure 4. 4 Model based approach in MATLAB for DDT fallback.....	29
Figure 4. 5 PID block diagram, from [7].....	32
Figure 4. 6 Schematic of longitudinal control in MATLAB. ....	33
Figure 4. 7 Schematic of Lateral control in MATLAB. ....	34
Figure 4. 8 Route under DDT fallback (note road shoulder is present). ....	37
Figure 4. 9 Route under normal navigation (note road shoulder is not present).....	37
Figure 4. 10 Bounding box used in object detection for the Lidar point cloud data... 40	
Figure 5. 1 State Chart XML used in Qt to run the main control loop and trigger calls to send/read data between python and CARLA. ....	43

Figure 6. 1 Velocity when parking is permitted vs velocity when parking is not permitted.....	52
Figure 6. 2 Comparison of CPU time and framerate .....	53
Figure 6. 3 Distance from vehicle to emergency shoulder.....	53
Figure 6. 4 Avg, max, min and std deviation.....	54



## **Acronyms**

### **AD**

Autonomous Driving

### **ADS**

Automated Driving System

### **ADAS**

Advanced Driver Assistance System

### **DDT**

Dynamic Driving Task

### **ODD**

Operational Design Domain

### **SCXML**

State Chart XML

### **CARLA**

Car Learning to Act

### **FOV**

Field of View

### **GNSS**

Global Navigation Satellite System

# Chapter 1

## **1. Introduction**

With the advancements of new technologies, the automotive industry is one that has seen lots of improvements towards adapting new safety systems that aid the driver during normal driving, but in the recent years many companies have been developing software aiming at having a full autonomous vehicle, at the present moment no vehicle is fully autonomous, but technology is slowly approaching this goal.

Therefore this thesis focuses on developing a DDT fallback strategy to be implemented in an autonomous vehicle and reach a further level of autonomy and rely less and less on the driver taking the controls, this fallback strategy is a safety feature that aims to bring the vehicle to a minimum risk condition, that is, putting the vehicle out of the active traffic lanes in a safe manner, without being a thread for other road users.

This thesis uses the CARLA simulator, an open source software that is used to implement autonomous driving software, facilitating the definition of sensors such as Lidars, RGB cameras, depth cameras, GNSS, Inertial sensors and others, that are a fundamental part on the development of autonomous driving software, because they allow to test the algorithms before being implemented in a real self-driving car.

### **1.1. What is autonomous driving**

An autonomous car, also called self-driving car or driverless car, is defined as one that uses technology to partially or entirely replace the human driver in navigating a vehicle from an origin to a destination while avoiding road hazards and responding to traffic conditions. Autonomous vehicles are responsible for all driving activities, such as perceiving the environment, monitoring important systems, and controlling the vehicle, which includes navigating its surroundings.

Self-driving cars (Figure 1.1) will bring many advantages, one of the most important of them is the ability to reduce the chance of accidents due to loss of attention during heavy traffic conditions, and their ability to react very fast to changes in road conditions.

Initial tests in the 80s and 90s demonstrated that vehicles could move autonomously under certain conditions, during this time development in GPS and

sensor technology allowed vehicles to more accurately locate and perceive objects around them, as of late of 2024 no system has achieved full autonomy (SAE level 5).



Figure 1.1 Bylogix VeGA autonomous vehicle

With the advancements in the development of technologies such as cameras, radars, lidars and artificial intelligence algorithms in vehicles, allowed them to navigate in complex traffic scenarios, many car manufacturers and technologies companies are now accelerating their efforts to develop fully autonomous vehicles, the goal of these vehicles is to reduce traffic accidents, increase transport efficiency and provide greater independence for people. However technical challenges as well as ethical. And safety issues continue to shape the advancement in this field. The future of autonomous driving depends on overcoming these challenges and gaining social acceptance of this technology.

However, incidents and casualties involving vehicles equipped with Automated Driving Systems (ADS) are still rising. For the merits of autonomous vehicles to be recognized more extensively, the immediate problem of ADS must be appropriately dealt with, namely, the perception and sensing ability in adverse weather conditions. [1]

## **1.2. SAE Levels of automation**

The *Society of Automotive Engineers* (SAE) has published a standard (J3016) to classify the different levels of automation and functionality in an automated driving system (ADS) (Figure 1. 2), the levels range from 0 to 5, being “0” a vehicle with no automation at all, while a level “5” corresponds to a high level automation or fully driverless vehicle, This classification is based on the role of the driver, rather than the vehicle's capabilities, at the present time, no vehicle has achieved full autonomy

(Level 5), there are currently level 3 vehicles, which have been commercialized and it is expected that level 5 vehicles will be commercialized in the near future. [2]

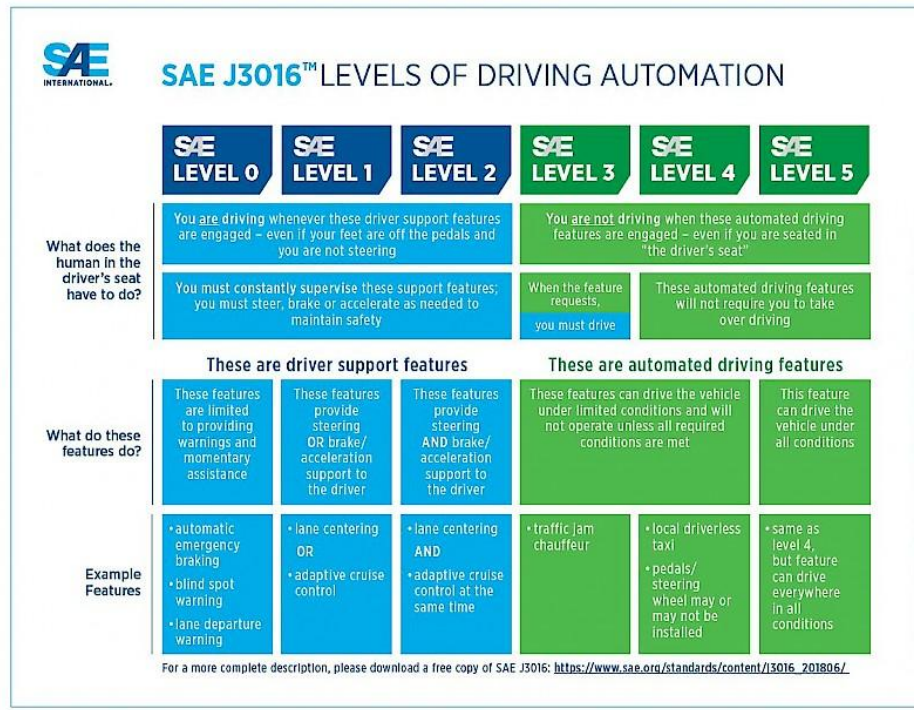


Figure 1. 2 SAE Levels of automation from [2]

- Level 0, No driving automation:

Full-time performance by the driver of all aspects of driving, even when "enhanced by warning or intervention systems, no automation.

- Level 1, Driver assistance:

The current driver assistance systems support drivers on the road and help ensure additional safety and comfort. Examples of this include the Active Cruise Control, driver is responsible for the DDT fallback.

- Level 2, Partial Driving automation:

The driving automation systems is capable of performing both lateral and longitudinal motion, but there are some events is not capable of recognizing or responding to; therefore the driver supervises the feature performance by completing the subtask of the DDT fallback.

- Level 3, Conditional driving automation:

With conditional automation systems, the automated driving system will be able to drive autonomously over the entire DDT under routine/normal conditions, in certain traffic situations, such as on motorways. With the expectation that the DDT

fallback user is receptive to request to intervene and take over control within a few seconds, such as at road construction sites.

- Level 4, High driving automation:

Level 4 is fully autonomous driving, although a human driver can still request control, and the car still has a cockpit. In level 4, the car can handle most driving situations independently such as performing the DDT fallback by itself. The technology in level 4 is developed to the point that a car can handle highly complex urban driving situations, such as the sudden appearance of construction sites, without any driver intervention, the car has the authority to move into minimal risk conditions.

- Level 5, Full driving automation:

Unlike levels 3 and 4, the “Full Automation” of level 5 is where true autonomous driving becomes a reality: Drivers don’t need to be fit to drive and don’t even need to have a license. The car performs any and all driving tasks – there isn’t even a cockpit. Therefore every person in the car becomes a passenger, opening up new mobility possibilities for people with disabilities, for example.

Cars at this level will clearly need to meet stringent safety demands, and will only drive at relatively low speeds within populated areas. They are also able to drive on highways but initially, they will only be used in defined areas of city centres. [3]

### **1.3. The Importance of simulation in Autonomous driving**

One very important tool for software development, and in particular for autonomous driving applications is the difference choice of simulators available in the market, these simulators provide us with many advantages, one of the most important parts in software development is testing and validation, the testing phase allows us to determine if our application fulfills its intended function, and whether it is safe for deployment in the target hardware.

In industries like aviation and automotive testing and simulation are very critical parts during the development process, specifically in model-based software applications after the model is created, it moves to the verification stage (Figure 1. 3), before being loaded in the target hardware it must go through a series of verification steps such as Model in the loop (MIL), Software in the loop (SIL) and hardware in the loop (HIL).

- **Model in the loop**

MIL testing is a simulation technique used in all stages of a product's controller development cycle. It involves testing a controller model at the component, composition, or system level within a simulation environment, before downloading the code into a physical Electronic Control Unit (ECU) and testing on a vehicle

- **Software in the loop**

Software-in-the-Loop testing, also called SiL testing, means testing embedded software, algorithms or entire control loops with or without environment model on a PC. In fact, SiL Testing is an integral part of automotive software testing. The source code for the embedded system is compiled for execution on the PC and then tested on the PC.

- **Hardware in the loop**

HIL is a technique that is used in the development and testing of embedded systems. The complexity of the plant under control is included in testing and development by adding a mathematical representation of all related dynamic systems. These mathematical representations are referred to as the "plant simulation". The embedded system to be tested interacts with this plant simulation.

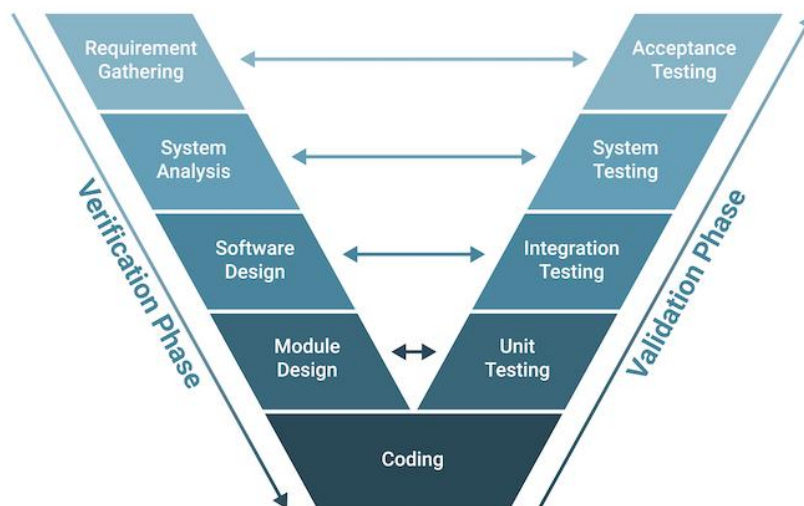


Figure 1. 3 V&V cycle validation process from [15]

Simulators fall under the SIL testing, the SIL testing method has many advantages, these benefits explain why SIL tests are an important part during the software development process, the main advantages are:

- Cost efficiency
- Rapid feedback loop
- Risk reduction
- Broad test scenarios
- Acceleration of the development process
- Preparation for integration and system test
- Flexibility in the development process

## **1.4. Autonomous driving simulators**

### **1.4.1. CARLA Simulator**

CARLA (Figure 1. 4) is an open-source autonomous driving simulator. It was built from scratch to serve as a modular and flexible API to address a range of tasks involved in the problem of autonomous driving.

One of the main goals of CARLA is to help democratize autonomous driving



Figure 1. 4 CARLA Simulator Logo from [4].

R&D, serving as a tool that can be easily accessed and customized by users. To do so, the simulator has to meet the requirements of different use cases within the general problem of driving (e.g. learning driving policies, training perception algorithms, etc.). CARLA is grounded on Unreal Engine to run the simulation and uses the ASAM OpenDRIVE standard (1.4 as today) to define roads and

urban settings. Control over the simulation is granted through an API handled in Python and C++ that is constantly growing as the project does. [4]

## 1.4.2. The Simulator

CARLA simulator consists of a scalable client-server architecture (Figure 1. 5). The server is responsible for everything related with the simulation itself: sensor rendering, computation of physics, updates on the world-state and its actors and much more. As it aims for realistic results, the best fit would be running the server with a dedicated GPU, especially when dealing with machine learning.

The client side consists of a sum of client modules controlling the logic of actors on scene and setting world conditions. This is achieved by leveraging the CARLA API (in Python or C++), a layer that mediates between server and client that is constantly evolving to provide new functionalities.

That summarizes the basic structure of the simulator. Understanding CARLA though is much more than that, as many different features and elements coexist

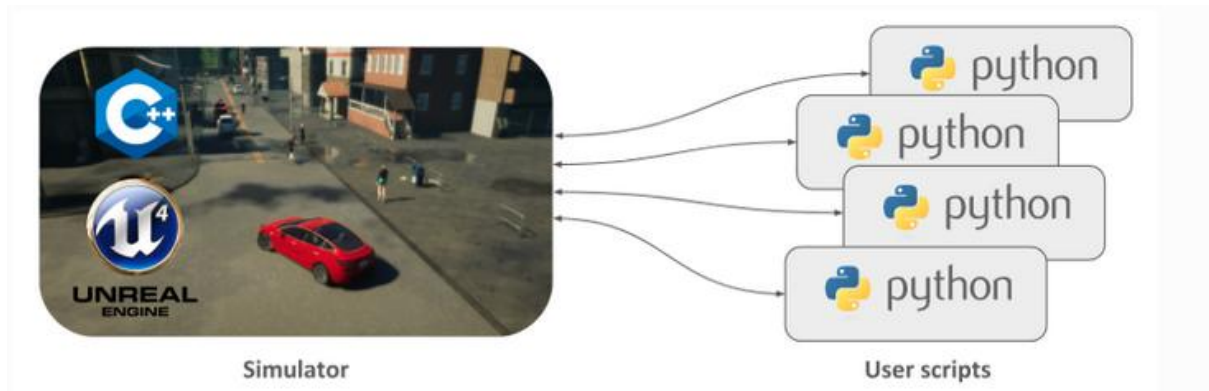


Figure 1. 5 Carla client-server architecture from [4]

within it. Some of these are listed hereunder, as to gain perspective on the capabilities of what CARLA can achieve.

- **Traffic manager.** A built-in system that takes control of the vehicles besides the one used for learning. It acts as a conductor provided by CARLA to recreate urban-like environments with realistic behaviors.
- **Sensors.** Vehicles rely on them to dispense information of their surroundings. In CARLA they are a specific kind of actor attached the vehicle and the data they receive can be retrieved and stored to ease the process. Currently the project supports different types of these, from cameras to radars, lidar and many more.



- **Recorder.** This feature is used to reenact a simulation step by step for every actor in the world. It grants access to any moment in the timeline anywhere in the world, making for a great tracing tool.
- **ROS bridge and Autoware implementation.** As a matter of universalization, the CARLA project ties knots and works for the integration of the simulator within other learning environments.
- **Open assets.** CARLA facilitates different maps for urban settings with control over weather conditions and a blueprint library with a wide set of actors to be used. However, these elements can be customized and new can be generated following simple guidelines.
- **Scenario runner.** In order to ease the learning process for vehicles, CARLA provides a series of routes describing different situations to iterate on.

In recent years CARLA has been growing and receiving active development, about once a year the simulator receives a major update, fixing bugs and adding new functionality with each update, it does so while never forgetting its open-source nature. The project is transparent meaning that anyone can have access to the tools and the development community, everybody is free to explore with CARLA, find their own solutions and then share their achievements with the rest of the community. [4]

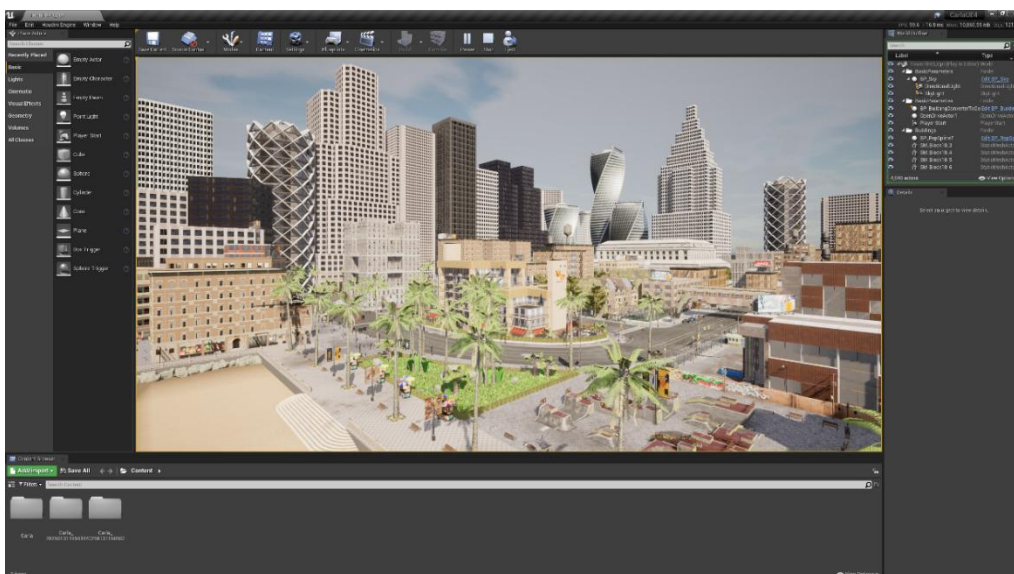


Figure 1. 6 Carla development build in Unreal engine 4

# Chapter 2

## 2. Physical system design

### 2.1. Sensors

#### 2.1.1. Lidar

Lidar sensors (Figure 2. 1) (Light detection and ranging) are used for determining distances, their working principle is based in targeting an object or surface with a laser and measuring the time the reflected light takes to return to the receiver.

In autonomous driving applications Lidar sensors are mounted in the top of the car in order to have a full view, and they generate a 3D point cloud data (Figure 2. 2) in

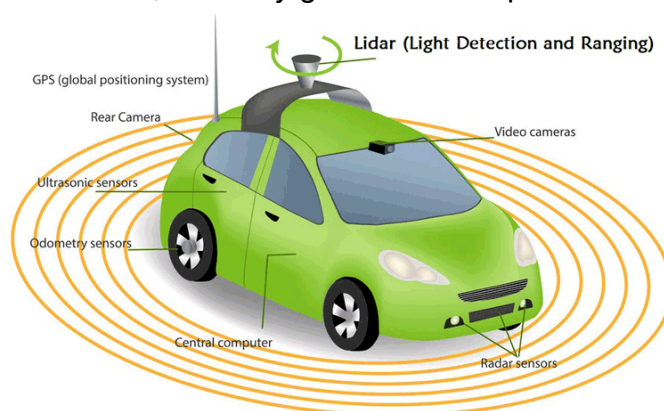


Figure 2. 1 Lidar in an autonomous vehicle from [16]

the form of x, y and z coordinates of the surroundings, that data can be further processed to extract the information given by the sensor, such as distance and angle to a detected object.

CARLA offers a wide range of customization options for the different types of sensors available and to adapt the simulated sensors to their real-world counterparts some of these customization options are:



Figure 2. 2 Point cloud data of a Lidar sensor from [4].

- **Channels:** Numbers of lasers
- **Range:** maximum distance Lidar can measure
- **Rotating frequency:** Lidar rotation frequency
- **Upper and lower FOV:** angle in degrees of higher and lower lasers
- **Horizontal FOV:** horizontal field of view in degrees

### 2.1.2. GNSS sensor

Another of the most relevant pieces of equipment in an autonomous car is its GNSS sensor, this device connects to GPS satellites to obtain latitude and longitude data for position calculation of the autonomous car.

GNSS performance is assessed using four criteria:

- **Accuracy:** the difference between a receiver's measured and real position, speed or time.
- **Integrity:** a system's capacity to provide a threshold of confidence and, in the event of an anomaly in the positioning data, an alarm.
- **Continuity:** a system's ability to function without interruption.
- **Availability:** the percentage of time a signal fulfils the above accuracy, integrity and continuity criteria.

This performance can be improved using regional Satellite-based Augmentation Systems (SBAS), such as the European Geostationary Navigation Overlay Service (EGNOS).

In CARLA GNSS sensors reports the current GNSS position of the self-driving car, this is calculated by adding the metric position to an initial geo reference location defined within the OpenDRIVE map definition.

Some properties of the GNSS sensors in CARLA that can be customized to test the sensor in different scenarios are,

- standard deviation for latitude, longitude and altitude: Standard deviation parameter in the noise model
- noise bias for latitude longitude and altitude: Mean parameter in the noise model

# Chapter 3

## **3. Preparing for the development environment**

The development and testing for autonomous driving software technologies requires a robust simulation environment, such as to represent the real world as closely as possible, the simulator is in charge of simulating the physics, vehicles, pedestrians, environmental conditions while also providing support for simulating the different types of sensors available in autonomous driving applications and allowing to connect and interact with different analytical tools such as MATLAB and Qt that will be used later in this thesis.

For our research project the ability to integrate the simulation data with MATLAB and other software through the Python API that CARLA offers was an important consideration for choosing the simulator platform, CARLA simulator's advanced graphics, physics engine, traffic and pedestrian simulation set and important base for this autonomous driving research, its compatibility with different software make it a versatile simulator for testing different driving scenarios.

### **3.1. Setting up CARLA with a virtual environment**

CARLA allows to communicate and control de simulation through the Python API, in order to have a working connection to the simulator it is necessary to have the proper python version to work with the specific version of CARLA we want to use, to make the process of connecting to CARLA and having different python versions in our OS without affecting other applications, a virtual environment tool called anaconda will be used, this tools allows us to have different python versions through different environments in our OS, thus we can have multiple environments and CARLA models on the same PC that run in separated environments without affecting the others, the steps to set up anaconda are as follows (Figure 3. 1 Development environment.Figure 3. 1).

1. The first step is to get CARLA from the developer website, there will be two versions to download, a precompiled version and a build version for development purposes, in our case the precompiled version is sufficient for our purposes.



Figure 3. 1 Development environment.

2. Download the anaconda software from its website.
3. Create the virtual environment, we specify the name of the virtual environment (Carla-sim) and the python version we will be using (3.7).

```
Conda create --name Carla-sim python=3.7
```

4. Activate the virtual environment with the command

```
Activate carla-sim
```

5. We can install some python libraries to be available for our scripts with the pip command.

```
Pip install carla  
Pip install numpy
```

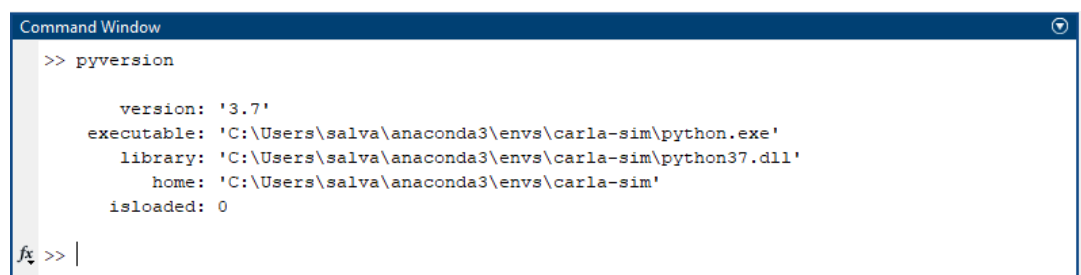
### **3.2. Running CARLA with MATLAB**

Since there is no a direct method to interface MATLAB with CARLA the first option available to allow communication is using the ROS bridge to connect to MATLAB with CARLA, this option requires more compatibilities with the operating system

and is more appropriate for Linux operating systems, or as a second choice and the one used in this thesis, is to rely on the python bridge, being simpler to set up and use, we must use a MATLAB version compatible with the python version we are using, in our case Python 3.7

1. Download and install MATLAB R2021B, this is the last version of MATLAB compatible with python 3.7.
2. To make MATLAB and CARLA communicate we must tell MATLAB to use correct python version we created in our virtual environment (Figure 3. 2).

```
Pyversion("C:\Users\salva\anaconda3\envs\carla-sim\python.exe")
```



```
Command Window
>> pyversion

version: '3.7'
executable: 'C:\Users\salva\anaconda3\envs\carla-sim\python.exe'
library: 'C:\Users\salva\anaconda3\envs\carla-sim\python37.dll'
home: 'C:\Users\salva\anaconda3\envs\carla-sim'
isloaded: 0

fx >> |
```

Figure 3. 2 MATLAB 2021 running python 3.7.

```
insert(py.sys.path,int32(0),'C:\CARLA_0.9.15\WindowsNoEditor\PythonAPI\carla\dist\carla-0.9.15-py3.7-win-amd64.egg');
```

```
py.importlib.import_module('carla');
```

These last two steps must be added to each script, so the necessary connections and libraries are included with the script.

This is the standard initialization step to connect to CARLA from MATLAB

```
1. insert(py.sys.path,
int32(0),['C:\CARLA_0.9.15\WindowsNoEditor\PythonAPI\carla\dist\carla-
0.9.15-py3.7-win-amd64.egg']);
2. py.importlib.import_module('carla');
3.
4. port = int16(2000);
5. client = py.carla.Client('localhost', port);
6. client.set_timeout(10.0);
7. world = client.get_world();
```

This section of code spawns a specific vehicle (model3) in the map at a specified spawn point.

```
8. % Spawn Vehicle
9. blueprint_library = world.get_blueprint_library();
10. car_list = py.list(blueprint_library.filter("model3"));
11. car_bp = car_list{1};
12. spawn_point = world.get_map().get_spawn_points();
13. start_point = spawn_point{72};
14. tesla = world.spawn_actor(car_bp, start_point);
15. %tesla.set_autopilot(true);
```

And finally, we move the spectator camera behind the vehicle to have a better view, (Figure 3. 3).

```
16. %%Move spectator behind the car
17. spectator = world.get_spectator();
18. vehicle_transform = tesla.get_transform();
19. location_offset = py.carla.Location(-10, 0, 2.5);
20. transform =
    py.carla.Transform(vehicle_transform.transform(location_offset),vehicle_tra
    nsform.rotation);
21. spectator.set_transform(transform);
```



Figure 3. 3 Result of running the code in MATLAB.

### 3.3. Running CARLA with Qt SCXML state machine

Qt is a cross-platform application development framework for desktop, embedded and mobile devices, some supported platforms include Linux, OS X, Windows, Android, iOS and others.

Qt is *not* a programming language on its own. It is a framework written in C++. In this thesis Qt is used to develop a Qt python application that interfaces with a state-machine application in the SCXML language, which is a general-purpose event-based state machine language that combines concepts from CCXML and Harel State Tables, Qt offers a user interface that allows to create the state machines. [5]

The first step is to download Qt from <https://www.qt.io/download-qt-installer-oss> and add the Qt state machines library from the Qt maintenance tool.

Connecting Qt to CARLA is simpler, since Qt allows us to write in python language, we just need to point Qt to our python executable in the virtual environment (Figure 3. 4).

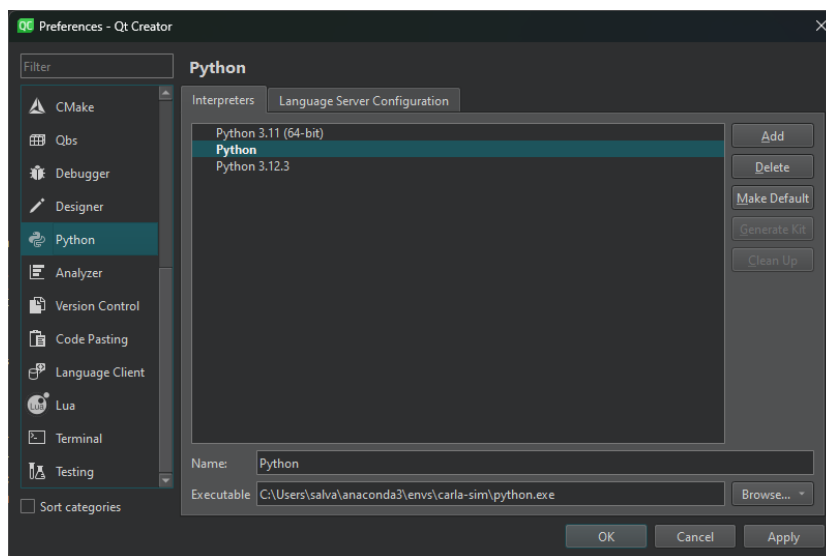


Figure 3. 4 Qt running in virtual environment with python 3.7.

To run our scripts to CARLA we just need to run the initialization steps in our python script

```
1. try:
2.     sys.path.append(glob.glob('../carla/dist/carla-.*d.*d-%s.egg' % (
3.         sys.version_info.major,
4.         sys.version_info.minor,
5.         'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
6. except IndexError:
7.     pass
8.
9. import carla
10. # Starting the simulation
```



```
11. client = carla.Client('localhost', 2000)
12. world = client.get_world()
```

Then the script to spawn a car and move the spectator behind the car

```
1. # Spawning the car
2. bp_lib = world.get_blueprint_library()
3. vehicle_bp = bp_lib.filter('*mini*')[0]
4. spawn_points = world.get_map().get_spawn_points()
5. #start_point = spawn_points[60]
6. start_point = carla.Transform(carla.Location(x=-27,y=69.7,
z=0.5),carla.Rotation(yaw=0.073))
7. vehicle = world.spawn_actor(vehicle_bp, start_point)
8.
9. spectator = world.get_spectator()
10. transform =
    carla.Transform(vehicle.get_transform().transform(carla.Location(x=-10,
z=2.5)),vehicle.get_transform().rotation)
11. spectator.set_transform(transform)
```

Giving a similar result to that obtained with MATLAB (Figure 3. 5).

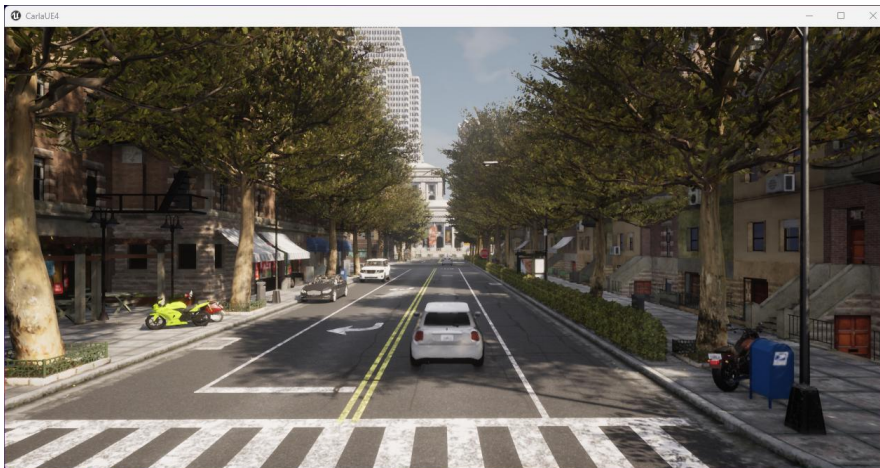


Figure 3. 5 Example of code execution using Qt.

# Chapter 4

## **4. Problem definition for DDT fallback strategy**

In the following section the case study to evaluate the DDT fallback strategy is presented, first the driving scenario is defined, followed by the parameters for the decision and control are mentioned.

The objective of this section is to evaluate the behavior of an automated driving system experiencing a DDT performance relevant system failure.

### **4.1.1. Scenario and road network**

An autonomous vehicle is driving in an urban environment when it experiences a DDT performance-related system failure.

A dual carriageway (Figure 4. 1) with one lane for each direction of travel, having an emergency lane, or road shoulder on its right side. The road was chosen from one of the CARLA maps specifically called town 10, having the required characteristics to test the DDT fallback strategy that will be proposed.

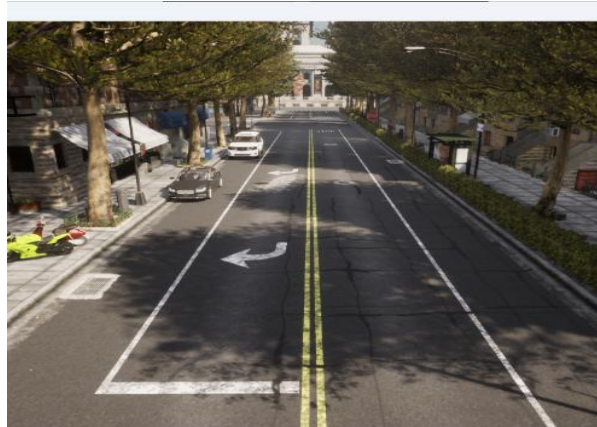


Figure 4. 1 Dual carriageway with emergency lanes.

### **4.1.2. Failure**

The Automated Driving System experiences a DDT performance relevant failure, that requires itself to be removed from the active lanes of traffic to a minimum risk condition.

## 4.2 Fallback strategy

A fallback strategy usually entails automatically bringing the vehicle to a stop within its current travel path but could also pertain to a more extensive maneuver designed to remove the vehicle from an active lane of traffic, it is assumed that bringing the vehicle to a hard stop is risky for other road users.

Nowadays, most automated driving systems rely on a DDT fallback-ready user to achieve a minimal risk condition, however a human driver may not respond to a request to intervene appropriately and in good time, which is why a fallback strategy is to be defined regardless of the presence of a human driver and his ability to intervene. The dynamic driving task fallback must also be defined for ADS-dedicated vehicles, designated to be operated exclusively by level 4 or level 5 ADS for all trips. [6]

In the case of a failure, the DDT fallback strategy starts, first the velocity is instantly reduced to a degraded value, and the current path is kept unchanged, the route is not modified until a safe parking spot, a road shoulder is found, empty without other cars occupying it, in this case the self-driving car is allowed to park in the road shoulder.

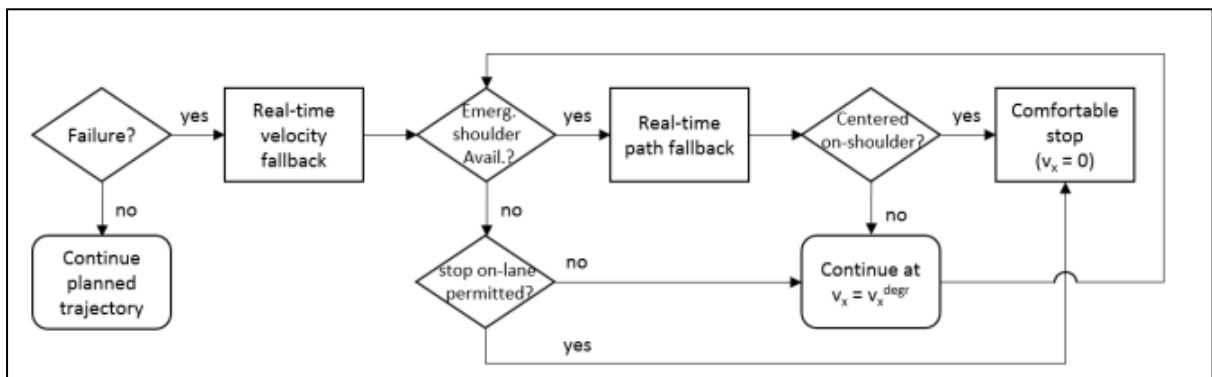


Figure 4. 2 Flow chart for DDT fallback strategy, from [6]

As an additional issue, the case is considered in which the emergency shoulder cannot be used (Figure 4. 3), as another vehicle is already parked on it, requiring driving a longer distance to the next permitted stop.

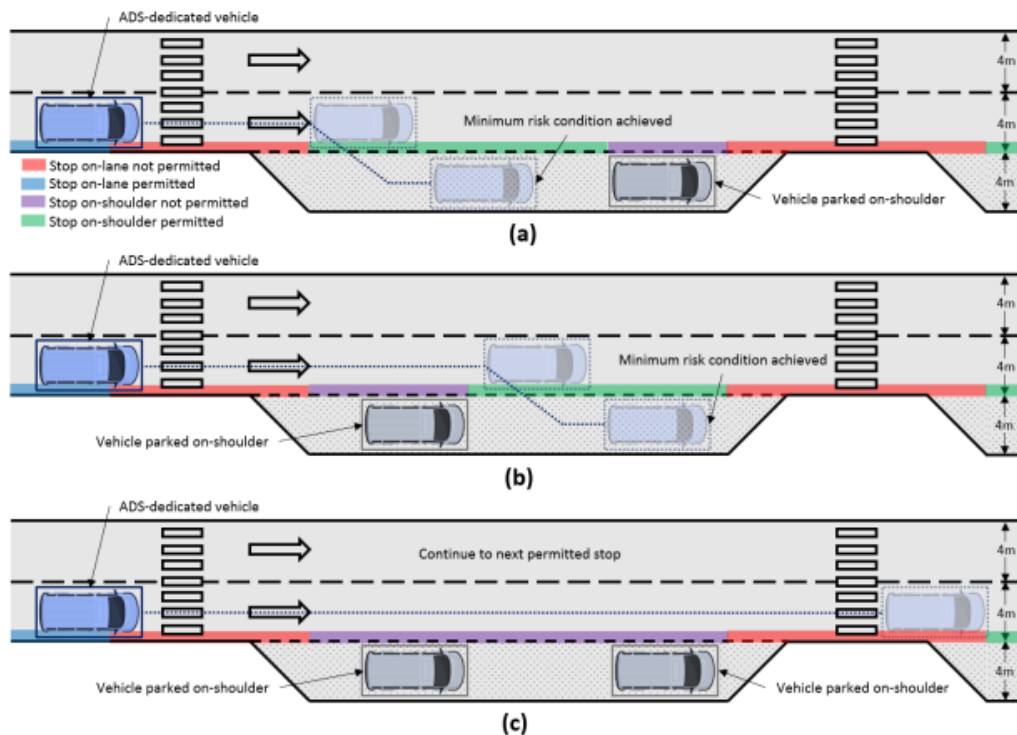


Figure 4. 3 DDT fallback scenarios, from [18].

### 4.3. Testing platform

A standard city vehicle has been selected as the test platform for the case study.

This way the test platform has been modeled in CARLA simulator, interfaced with MATLAB and later on with Qt state machine, using the unreal engine for graphics and physics simulation and with the CARLA Python API, the sensors have been simulated in each platform adding GNSS sensor and lidar sensor for object detection with a maximum range of 50m.

## 4.4. Different approaches for model-based DDT-fallback strategies

### 4.4.1. Model based modeling of the DDT-fallback in MATLAB

In order to model the CARLA environment with MATLAB and communicate with it, a connection must be established first, as shown in previous chapters.

To be able to control the car inside an urban environment we must ensure both the longitudinal control and lateral control, this is the first step to control and implement

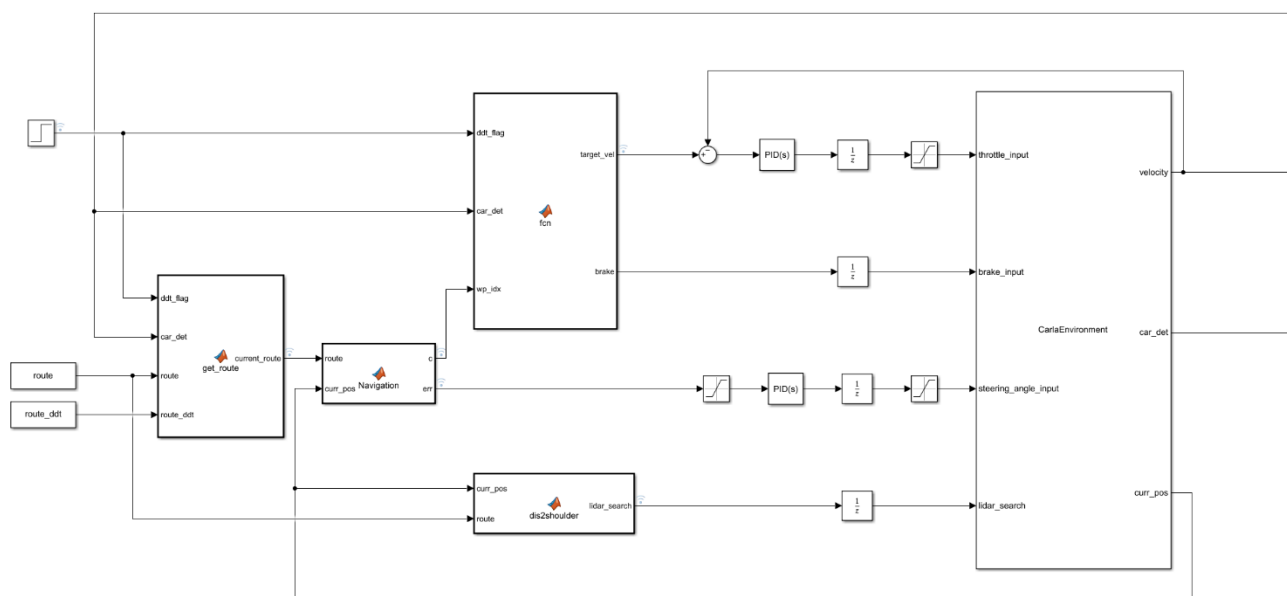


Figure 4. 4 Model based approach in MATLAB for DDT fallback.

a DDT-fallback strategy, longitudinal control deals with throttle, braking and reverse, while lateral control deals with navigation acting on the steering wheel of the ADS vehicle.

The way MATLAB communicates with CARLA in Simulink, is through a main block, which is CARLA environment on the image above, this block is essential to run the simulations since it outputs and inputs data to the simulation, and for MATLAB to process and generate a control signal, all the function blocks (Figure 4. 4) in this application will be further described below, and the same image is shown enlarged in appendix D.

## 4.5. CARLA environment

Creating the CARLA environment is done through the MATLAB s-function block, in our model, it has 3 outputs and 4 inputs, the outputs are, velocity, which is the current velocity of the car, car\_det, being a Boolean signal coming from the lidar sensor, and curr\_pos is the current position of the vehicle in x, y coordinates.

The inputs to the simulation are:

- **Throttle**, that is a value in the range from 0 to 1, controls the throttle of the vehicle.
- **Brake**, similarly, is a value in the range 0 to 1 that controls braking.
- **Steering** angle is a value in the range from -1 to 1, that controls the amount of steering.
- **Lidar** search is a Boolean value that activates the lidar sensor to search for vehicles when the parking position is approaching,

To apply, the commands `control.throttle`, `control.brake` and `control.steer` are used in the code.

In the first section, the car is spawned at our specified location, after this we move the spectator behind the car to follow the vehicle through the simulation, we initialize and define the sensors properties, this section of the code is run only once, the main function inside is used to apply the control commands, calculate the speed, and output the data required to compute the algorithms such as the x, y position of the vehicle, below is shown a portion of the script, the full code is shown in appendix A.

At the end of the simulation all actors, including vehicles and sensors are destroyed from the world.

```
1. classdef CarlaEnvironment < matlab.System & matlab.system.mixin.Propagates
2.
3.     function [velocity, car_det, curr_pos] = stepImpl(obj, throttle_input,
4.         brake_input, steering_angle_input, lidar_search)
5.         %
6.         pause(0.001);
7.         x_vel = obj.car.get_velocity.x;
8.         y_vel = obj.car.get_velocity.y;
9.         z_vel = obj.car.get_velocity.z;
10.        velocity = 3.6 * sqrt((x_vel)^2 + (y_vel)^2 + (z_vel)^2);
11.
12.        v_vec = obj.car.get_transform().get_forward_vector();
13.        % v_vec = [v_vec.x v_vec.y 0]
14.        ego_trans = obj.car.get_transform().location;
15.        x_pos = ego_trans.x;
16.        y_pos = ego_trans.y;
17.        % ego_loc= [x_pos y_pos 0]
```

```

17.     curr_pos = [x_pos y_pos 0 v_vec.x v_vec.y 0];
18.
19.     control = obj.car.get_control();
20.     control.steer = 0;
21.     control.throttle = throttle_input;
22.     control.brake = brake_input;
23.     control.steer = steering_angle_input;
24.     obj.car.apply_control(control);
25.
26.     lidarData = single(py.getattr(obj.moduleLidar, 'array'));
27.     lidarData = double(lidarData(:, 1:3)); % Convert to double and
    extract x, y, z coordinates
28.     point_cloud = lidarData; % Output the LIDAR data
29.     [m, n] = size(point_cloud);
30.     within_range_mask = (point_cloud >= obj.lower_thresholds) &
    (point_cloud <= obj.upper_thresholds);
31.
32.     % Filter out elements not meeting the threshold (optional: set to
    NaN or zero)
33.     valid_rows = all(within_range_mask, 2);
34.     filtered_rows = point_cloud(valid_rows, :);
35.     distances = sqrt(sum(filtered_rows.^2, 2));
36.
37.     if (distances < 25) & (obj.car_det == 0)
38.         obj.car_det = 1;
39.     elseif lidar_search == 0 && obj.car_det == 1
40.         obj.car_det = 0;
41.     end
42.
43.     car_det = obj.car_det;
44. end
45. end
46.

```

## 4.6. PID Controllers

A proportional–integral–derivative controller (PID controller) is a feedback-based control loop mechanism commonly used to manage machines and processes that require continuous control and automatic adjustment. It is typically used in industrial control systems. The PID controller automatically compares the desired target value (setpoint or SP) with the actual value of the system (process variable or PV).

It then applies corrective actions automatically to bring the PV to the same value as the SP using three methods: The proportional (P) component responds to the current error value by producing an output that is directly proportional to the magnitude of the error. The integral (I) component, in turn, considers the cumulative sum of past errors to address any residual steady-state errors that persist over time. Lastly, the derivative (D) component predicts future error by assessing the rate of change of the error.

A common example is a vehicle's cruise control system. When a vehicle encounters a hill, its speed may decrease due to constant engine power. The PID controller adjusts the engine's power output to restore the vehicle to its desired speed, doing so efficiently with minimal delay and overshoot. [7]

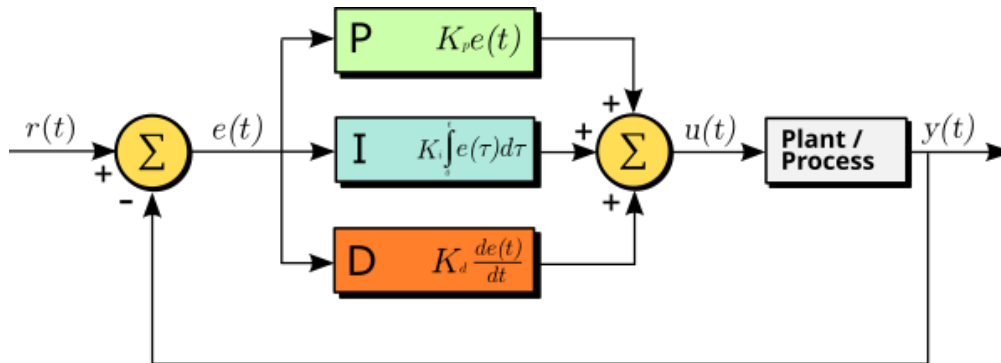


Figure 4. 5 PID block diagram, from [7]

The balance of these effects is achieved by loop tuning to produce the optimal control function. The tuning constants are shown below as "K" and must be derived for each control application, as they depend on the response characteristics of the physical system, external to the controller. These are dependent on the behavior of the measuring sensor, the final control element (such as a control valve), any control signal delays, and the process itself. Approximate values of constants can usually be initially entered knowing the type of application, but they are normally refined, or tuned, by introducing a setpoint change and observing the system response.

The overall control function is.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

Where  $K_p$ ,  $K_i$  and  $K_d$ , are all non-negative, denote the coefficients for the proportional, integral and derivative terms.

### 4.6.1. Longitudinal control

The main control strategy for the longitudinal axis of the vehicle is through a PID controller that acts directly on the throttle or braking (Figure 4. 6) to reach a constant velocity in the ADS vehicle.



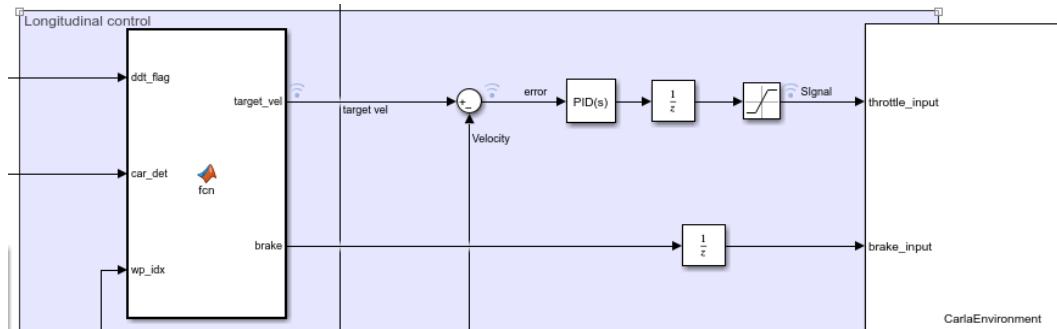


Figure 4. 6 Schematic of longitudinal control in MATLAB.

In the DDT fallback strategy the target velocity is set based on conditions, such as if there is any ADS performance relevant failure, the velocity is reduced to a degraded value, or if the vehicle has reached the parking position it will apply braking to stop the vehicle, first the target velocity is fed to the control system initially is 20 km/h. It calculates the error between the current velocity and the target velocity, and it generates a control signal that must be between 0 and 1 since the PID controller can output a value greater than 1 we must ensure the value fed to the ADS car is inside the range 0 - 1 so a saturation block is added, this will be the amount of throttle applied in the vehicle.

```

1. function [target_vel, brake] = fcn(ddt_flag, car_det, wp_idx)
2.
3. target_vel = 20;
4. brake = 0;
5.
6. if ddt_flag == 1
7.     target_vel = 10;
8. end
9.
10. if ddt_flag == 0 || car_det == 1
11.     %no parking in shoulder
12. elseif ddt_flag == 1 && wp_idx > 38 && wp_idx < 50
13.     target_vel = 0;
14.     brake = 0.3;
15. end

```

This function block sets the target velocity the vehicle must follow based on the conditions, if there is any performance relevant failure, the target velocity is instantly reduced to 10km/h

- `ddt_flag`: is a variable used to tell the ADS a failure has been detected, and it must perform the DDT-fallback as soon as possible.
- `car_det`: is a variable given by filtering the lidar point cloud data to detect if a vehicle is parked in the road shoulder.

```

10. if ddt_flag == 0 || car_det == 1
11.     %no parking in shoulder
12. elseif ddt_flag == 1 && wp_idx > 38 && wp_idx < 50
13.     target_vel = 0;
14.     brake = 0.3;
15. end

```

This sets of conditions determine if the vehicle is allowed to stop at the road shoulder if no vehicles are detected, there is a performance relevant issue and we are in the road shoulder, the vehicle will stop, the position of the road shoulder must be known at priory since the ADS vehicle is not equipped with vision system, it relies on GPS waypoints to navigate.

## 4.6.2. Lateral control

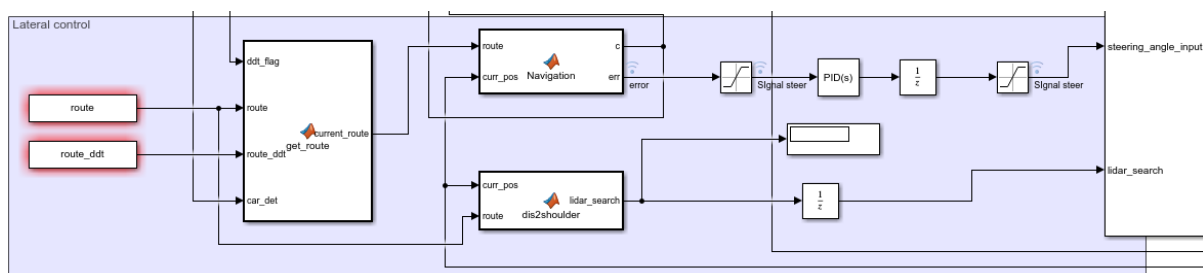


Figure 4. 7 Schematic of Lateral control in MATLAB.

The lateral control similarly to the longitudinal control is realized though a PID controller to fed the steering value to the ADS car (Figure 4. 7), the goal of lateral control is to navigate the vehicle through a determined set of waypoints that form a route, the function Navigation plays an important role, generating the error between the current position of the car and the waypoint it is tracking, to feed the PID controller.

```

1. function [c, err] = Navigation(route, curr_pos)
2.
3. ego_loc = curr_pos(1:3); %x,y,z coordinates
4. v_vec = curr_pos(4:6);

```

Route: is a matrix [N x 3] containing the set of waypoints that form the route in the form [x, y, yaw]

- Curr\_pos: this input denotes the actual position of the vehicle with at least the x and y coordinate.

- `v_vec`: represents the forward vector of the vehicle in the form  $[x, y, 0]$ , is obtained from CARLA with the command,

```
v_vec = obj.car.get_transform().get_forward_vector();
```

1. Calculate the number of waypoints in the route.

```
N = size(route,1);
```

2. Matrix containing the distance from the current position to the next waypoints in the route, this line computes the Euclidean distance between each waypoint in the route.

```
dis = vecnorm(route(:,1:2)' - curr_pos(1:2)'.*ones(1,N))';
```

3. Identify the closest waypoint.

```
[~,c] = min(dis);
```

4. Switch to the next waypoint if the vehicle is within a threshold, this is done to prevent the vehicle from tracking a waypoint that is very near and could cause issues in the PID controller.

```
% Add threshold to switch to the next waypoint if close enough
min_distance_threshold = 1; % Distance threshold in meters
if dis(c) < min_distance_threshold && c < N
    c = c + 1; % Move to the next waypoint if within threshold
end
```

5. First the tracking waypoint is selected in `w_loc`, then the difference `w_loc - ego_loc` is calculated and finally `wv_linalg` computes the product of the magnitudes of `w_vec` and `v_vec`.

```
w_loc = route(c,:);
w_vec = [w_loc(1) - ego_loc(1), w_loc(2) - ego_loc(2), 0];
wv_linalg = norm(w_vec) * norm(v_vec);
```

6. Compute the heading error angle between vectors.

```
% Compute dot product
dot_product = dot(w_vec, v_vec);
```

```

if wv_linalg == 0
    cos_theta = 1; % Use a small positive value
else
    % Calculate the angle (clipping between -1 and 1 to avoid numerical
    % issues)
    cos_theta = min(max(dot_product / wv_linalg, -1), 1);
end

```

7. Compute the error angle and determine the sign, if vehicle is on the left of the reference trajectory the error is negative, if it is on the right the error is positive.

```

err = acos(cos_theta);
cross_product = cross(v_vec, w_vec);

if cross_product(3) < 0
    err = -err;
end

```

The MATLAB function navigation provides the error for path tracking which is essential to steer the vehicle along a route, provides essential metrics for correction, allows the PID controller to correct and adjust the position of the vehicle, which is critical for navigation precision and efficiency in ADS vehicles. [8] [9] [10] [11]

## 4.7. MATLAB functions for DDT-fallback

The previous section focused on the algorithm that controls the car both laterally and longitudinally. These controllers are essential to implement a DDT-fallback strategy to the ADS vehicle, next the following functions aims at having a set of conditions to control the behavior of the car under a DDT-fallback scenario.

### 4.7.1. MATLAB function get route.

This function aims at feeding the navigation algorithm with the appropriate route given the situation the ADS vehicle is in, 2 routes are designed a priori (Figure 4.8 and Figure 4.9), one route is used under normal navigation of the ADS vehicle, while the other route, is used only under DDT-fallback and this route contains the road shoulders.



Figure 4. 8 Route under DDT fallback (note road shoulder is present).



Figure 4. 9 Route under normal navigation (note road shoulder is not present).

When the vehicle enters in DDT-fallback, it will switch and follow the ddt\_route and will park in the road shoulder if no other vehicle is occupying the place, if a vehicle is present, meaning it was detected by the lidar, the route is changed to the original one, before the car enters in the road shoulder, thus continuing on the main route, until a safe free road shoulder is found.

```
5. function current_route = get_route(ddt_flag, route, route_ddt, car_det)
6.
7. if ddt_flag == 1 && car_det == 0
8.     current_route = route_ddt;
9. elseif ddt_flag == 1 && car_det == 1
10.    current_route = route;
11. else
12.    current_route = route;
13. end
```

#### 4.7.2. MATLAB function dis2shoulder

This MATLAB function has a simple objective, its main goal is to compute the distance the ADS vehicle is from the road shoulder, it is important to know how far is the vehicle, because in this manner the filtering of the data from the lidar is enabled when the vehicle is at a specific distance from the road shoulder, and determine whether a vehicle is present or not.

Its inputs are curr\_pos (current position), route (to store the position of the road shoulder), and output lidar\_search (variable to enable the filtering of the lidar data)

```
1. function lidar_search = dis2shoulder(curr_pos, route)
2.
3. wp = route(35,1:2);
4. dis = vecnorm(curr_pos(1:2)'-wp(1:2)');
5.
6. lidar_search = 2;
7.
8. if dis < 25 && dis > 22
9.     lidar_search = 1;
10. elseif dis > 50
11.     lidar_search = 0;
12. end
```

- wp: stores the position of the road shoulder
- dis: computes the distance between the ADS vehicle and the road shoulder

The conditional statements turn on and off the filtering of the lidar, when the vehicle is between 25 and 22 meters from the road shoulder the filtering of the lidar data is enabled to detect for vehicles.

### 4.7.3. Getting Lidar data

For autonomous driving, Lidar is pretty much a standard. It allows the car to view its surroundings using a point cloud. CARLA also allows to simulate it by using ray cast (think of it as a laser) and generating the 3D points. For transferring the data between CARLA and Simulink, a Python script will be used. Depending on the sensors and the settings chosen, it automatically generates the Python files behind the scenes.

We start by launching MATLAB, initialize the CARLA environment and add some vehicle to the CARLA world. The next step is to add the sensor. It is typically attached to an actor. A sensor is also an actor, the way it is spawned is similar to vehicle. We start by getting the blueprints from the world and filtering it for Lidar sensor:

```
1. % Lidar
2. blueprint = world.get_blueprint_library().find('sensor.lidar.ray_cast');
```

Before spawning the Lidar, we can change the attributes.

- Number of points generated per second
- How far the lidar can see
- Time for one complete rotation
- Field of view
- Sensor location relative to the actor it will be attached to

```
3. blueprint = world.get_blueprint_library().find('sensor.lidar.ray_cast');
4. blueprint.set_attribute('points_per_second', '250000');
5. blueprint.set_attribute('range', '50');
6. blueprint.set_attribute('upper_fov', '45.0');
7. blueprint.set_attribute('lower_fov', '-30.0');
8. blueprint.set_attribute('rotation_frequency', '20');
9. blueprint.set_attribute('channels', '64');
10. transform = py.carla.Transform(py.carla.Location(pyargs('x',0.8,
    'z',1.7)));
```

The way CARLA publishes the sensor data is through callback functions. Now, here is the tricky part. As Python and MATLAB run asynchronously, it is not so straightforward to exchange data between them. The solution for this is to make globally scoped variables on the Python side and let MATLAB read from it

asynchronously. This way, the Python part is responsible for writing to the buffer (variable), whereas MATLAB just reads from it, in appendix D the complete python script used to exchange data is shown.

Basically, sensor is the sensor object we created in the block before. It is required to bind it to the callback function. The argument file\_name specifies the generated python file name that contains the callback function. Depending on the sensor chosen, sensor\_mode allows different outputs formats. The data that the sensor returns need to be stored somewhere. The parameter var\_name allows us to give name to that variable. The function returns a module that can be queried later for the value of var\_name. So, for this example:

```
1. moduleLidar = sensorBind(sensor, 'lidar_file', 'lidar', 'array');
```

in our case

```
2. obj.moduleLidar = sensorBind(obj.sensor, 'sensor', 'lidar', 'array');
```

In the following section of code, the 3d point cloud data obtained from the lidar sensor is processed and filtered to scan for a particular region around the car (Figure 4. 10).

- **Upper\_thresh1ods:** [1 x 3] matrix containing the upper limits in meters for filtering the lidar data in x, y, z coordinates, In our case [25, 3, 0], the lidar is allowed to scan, 25m ahead 3m to the right side, and 0m from the height of the lidar, lidar is at 1.7m from the ground
- **Lower\_threshold:** [1 x 3] matrix containing the lower limits in meters for filtering the lidar data in x, y, z coordinates, In our case [0, -3, -1.5], the lidar

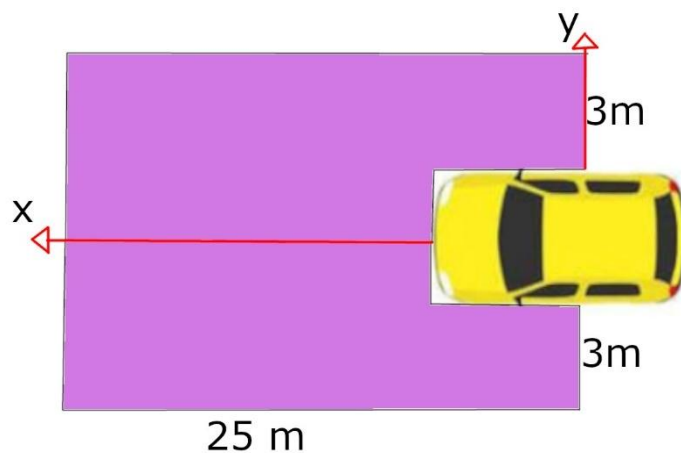


Figure 4. 10 Bounding box used in object detection for the Lidar point cloud data.



is allowed to scan starting from, 0m ahead, 3m to the left side, and 1.5m down from the height of the lidar, lidar is at 1.7m from the ground.

Extract x, y, z coordinates from the point cloud data obtained from the lidar sensor.

```
1. lidarData = single(py.getattr(obj.moduleLidar, 'array'));
2.     lidarData = double(lidarData(:, 1:3)); % extract x, y, z
   coordinates
```

- **within\_range\_mask:** is a matrix that stores 1 if the condition is true, and zeros if the condition is false, if they are within the thresholds we defined earlier.
- **Valid\_rows:** is a vector of zeros or ones, ones if in the given row the condition was met, that is x,y,z coordinates from the lidar are inside the threshold.
- **Filtered\_rows:** stores the x,y,z coordinates of the measurement that are inside the threshold
- **Distances:** Computes the distance between the lidar sensor and the object the laser hit.

If a measurement is inside 25 meters given the current filter for the lidar sensor an object is detected, in this case, a car is detected in the shoulder and the ADS vehicle is not allowed to park.

```
3.     point_cloud = lidarData; % Output the LIDAR data
4.     [m,n]=size(point_cloud);
5.     within_range_mask = (point_cloud >= obj.lower_thresholds) &
   (point_cloud <= obj.upper_thresholds);
6.
7.
8.     valid_rows = all(within_range_mask, 2);
9.     filtered_rows = point_cloud(valid_rows, :);
10.    distances = sqrt(sum(filtered_rows.^2, 2));
11.
12.    if (distances < 25) & (obj.car_det == 0)
13.        obj.car_det = 1;
14.    elseif lidar_search == 0 && obj.car_det == 1
15.        obj.car_det = 0;
16.    end
```

## Chapter 5

### **5. Qt SCXML model-based**

In the following section the DDT-fallback strategy is implemented using State Chart XML, with a graphical approach to model the state charts in the Qt application and using Qt for python Pyside 6 module that provides functionality to create and connect our script to the State Chart XML file

The Qt SCXML module provides classes for embedding state machines created from State Chart XML (SCXML) files in Qt applications. The SCXML files can be created using any suitable tool, such as a text editor or a simulator, as long as they comply to the SCXML Specification, with the restrictions and extensions described in SCXML Compliance.

The basic state machine concepts, state, transition, and event are based on those in the SCXML Specification. State charts provide a graphical way of modeling how a system reacts to stimuli. This is done by defining the possible states that the system can be in, and how the system can move from one state to another (transitions between states). A key characteristic of event-driven systems (such as Qt applications) is that behavior often depends not only on the last or current event, but also the events that preceded it. With state charts, this information is easy to express. [5] [12]

## 5.1. Functions

Since Qt allows for creating applications using the python language with the Pyside 6 module, connecting and getting data from CARLA is more straightforward, below is shown the state chart (Figure 5. 1) used to run the main loop and trigger call of events in the python side.

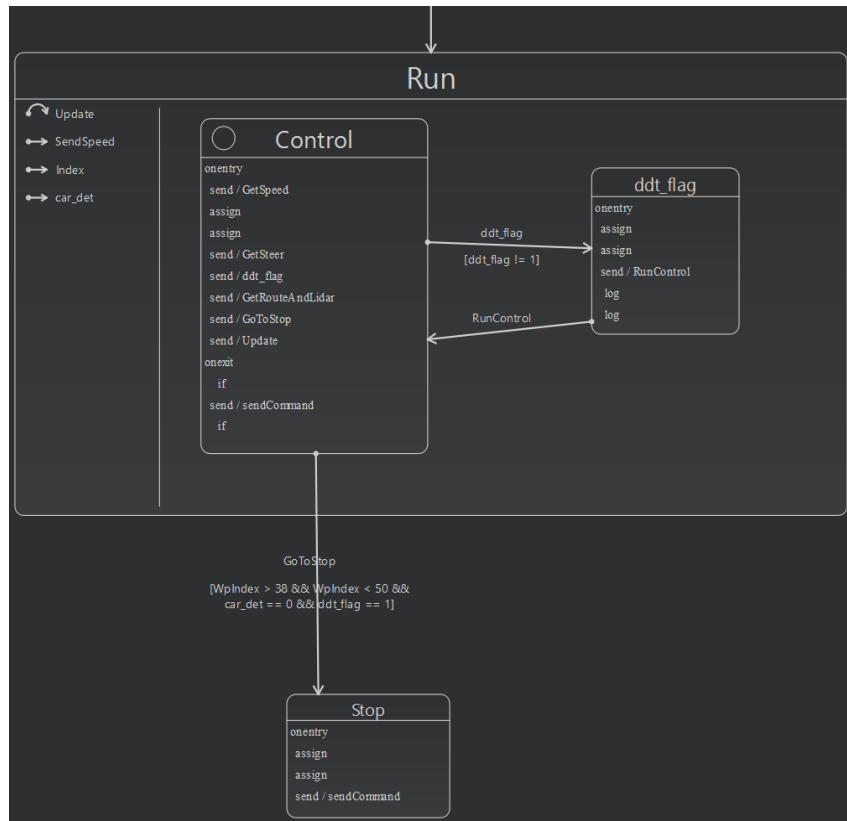


Figure 5. 1 State Chart XML used in Qt to run the main control loop and trigger calls to send/read data between python and CARLA.

## 5.2. How variables are sent and received with events

The way Qt with pyside 6 interacts with the State Chart XML, is through events and transition, every time a transition or event is triggered in the state chart, that is, every time we move from one state to the other, or by calling events inside the state we can trigger a call to exchange data for further processing. [5] [13]

The working principle in this state chart is the following, first there are 2 main states, run and stop, the run state is active while the ADS vehicle is under normal operation and all the main control algorithms are executed there, PID for lateral control and longitudinal control, as well as exchanging variables with CARLA and python, the run state is run continuously by calling the transition *update*, it is a transition to the same state, similar to running inside a loop, it is executed every 10 ms.

- **Sendspeed:** command used to send the speed value from python to the State Chart XML
- **Getspeed:** command used to trigger the event in python and get the speed value from python to the State Chart XML
- **SendCommand:** applies the control signals to CARLA every time the event is called in the SCXML.

The following section of code shows how the state machine is initialized and how the events are handled.

First a class called ScxmlHandler is created and it contains a QObject, then the SCXML file is stored in self.stateMachine, then we create the events we want to read from the SCXML file with the command self.stateMachine.connectToEvent, we must give the following parameters, **scxmlEventSpec – str, receiver object, method – str, and finally we start the state machine with the command self.stateMachine.start()**

```

17. class ScxmlHandler(QObject):
18.     def __init__(self, scxml_file, parent=None):
19.         super().__init__(parent)
20.         self.stateMachine = QScxmlStateMachine.fromFile(scxml_file)
21.
22.         # Check if the state machine is initialized properly
23.         if not self.stateMachine:
24.             print("Failed to initialize the state machine. Check SCXML file
and syntax.")
25.             return # Exit initialization if state machine failed to load
26.
27.         # Connect event from SCXML to a slot in Python using the correct
signature
28.         self.stateMachine.connectToEvent("GetSpeed", self,
SLOT("handleDataFromScxml(QScxmlEvent)"))
29.         self.stateMachine.connectToEvent("sendCommand", self,
SLOT("handleDataFromScxml(QScxmlEvent)"))
30.         self.stateMachine.connectToEvent("GetSteer", self,
SLOT("handleDataFromScxml(QScxmlEvent)"))
31.         # Connect the state machine's "Log" signal to the outputReceived
slot
32.         #self.stateMachine.Log.connect(self.outputReceived)
33.
34.         # Start the state machine before connecting slots to ensure all
signals are caught
35.         self.stateMachine.start()

```

The next section of code shows when an event is intercepted by the python script, that is, when an event is triggered in the SCXML state chart, for example, if we want to connect to the event 'GetSpeed', first we must send an event from the SCXML state chart with the name 'GetSpeed', this event is used to compute the speed of the vehicle, and then is sent to the SCXML state chart with the command

'SendSpeed', this command is read in the state chart and assign to a variable for further processing, in appendix C the complete code is shown.

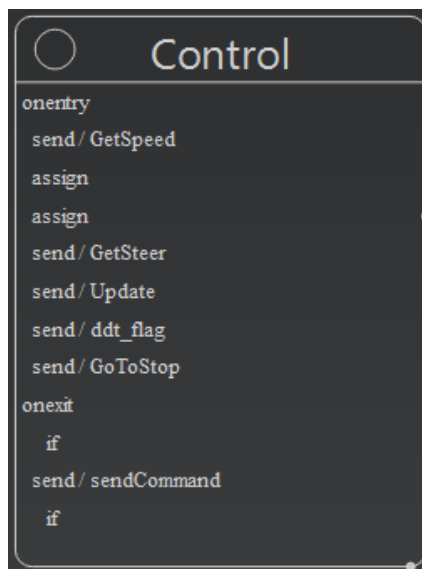
```
1.     @Slot(QScxmlEvent)
2.     def handleDataFromScxml(self, event: QScxmlEvent):
3.         #print("Received SCXML event:", event.name())
4.         data = event.data() # data() returns a QVariantMap, accessed like
   a dictionary in Python
5.
6.         #prt(data)
7.         #print(data["cmd_throttle"])
8.         if event.name() == 'GetSpeed':
9.             current_speed = compute_speed(vehicle)
10.            #print(f'Entered get speed, Speed is [{current_speed}]')
11.            self.send_event_with_data(current_speed, "SendSpeed")
12.
13.            #if event.name() == 'GetSteer':
14.            #    indx = dist_to_wp(vehicle, route)
15.            #    cmd_steer = lateral_control(vehicle, route[indx], k_p_s=0.4,
   k_d_s=0, k_i_s=0)
16.
17.            if event.name() == 'sendCommand':
18.                global route, ddt_flag # Declare global variables
19.                cmd_throttle = data["cmd_throttle"]
20.                cmd_brake = data["cmd_brake"]
21.                ddt_flag = data["ddt_flag"]
22.                #print('recieved cmd throttle', cmd_throttle)
23.
24.                if ddt_flag == 1:
25.                    route = route_ddt
26.                    print("route fallback!")
27.
28.                    indx = dist_to_wp(vehicle, route)
29.                    self.send_event_with_data(indx, "Index")
30.                    cmd_steer = lateral_control(vehicle, route[indx], k_p_s=0.4,
   k_d_s=0, k_i_s=0)
31.                    vehicle.apply_control(carla.VehicleControl(throttle =
   cmd_throttle, steer = cmd_steer, brake = cmd_brake))
32.                    #print(cmd_brake)
```

## 5.3. Pid controllers

### 5.3.1. Longitudinal control PID

The longitudinal control algorithm in Qt is implemented in the following way, first inside our main loop the *run state*, the variables such as `current_speed`, `error`, `target_speed`, `cmd_throttle` and `cmd_brake` are initialized.

Every time the control state is run by calling the event *update* every 10ms, an event `GetSpeed` is called, this event is read in the python script and gets the current speed of the car using the function `compute speed` and then the variable is sent to the state machine with the event `SendSpeed`



```
1. if event.name() == 'GetSpeed':
2.     current_speed = compute_speed(vehicle)
3.     #print(f'Entered get speed, Speed is [{current_speed}]')
4.     self.send_event_with_data(current_speed, "SendSpeed")
```

The function `compute_speed`, reads the current velocity from CARLA in the form `[x,y,s]` (m/s) and then it computes the resultant vector and converts the velocity to km/h.

The `SendSpeed` event is read by the state machine and is assigned to the variable `current speed`

*location	current_speed
expr	_event.data['var']

Following this, we compute the current error

*location	error
expr	target_speed-current_speed

Then we assign the command to the throttle value

*location	cmd_throttle
expr	(k_p*error)

In this way we have a simple proportional controller that tracks the target velocity, finally we set the upper and lower limits for the throttle command that must be between 0 and 1.

Once the value for the throttle is computed it is sent back to the python script so it can also be applied to CARLA, a sendCommand event on exit is created with the following data.

Finally, the control variables are applied to CARLA with the command `vehicle.apply_control`

*name	cmd_brake
expr	cmd_brake
location	

*name	cmd_throttle
expr	cmd_throttle
location	

```
1.     if event.name() == 'sendCommand':
2.         global route, ddt_flag # Declare global variables
3.         cmd_throttle = data["cmd_throttle"]
4.         cmd_brake = data["cmd_brake"]
5.         ddt_flag = data["ddt_flag"]
6.
7.         vehicle.apply_control(carla.VehicleControl(throttle =
8.             cmd_throttle, steer = cmd_steer, brake = cmd_brake))
```

### 5.3.2. Lateral control PID

The lateral control pid is realized in a similar manner to that one in matlab, in this case the algorithm to compute the error is run directly in the python script, given the complexity of it

First we get the current location of the vehicle with the command `vehicle.get_transform().location` and the values `[x, y]` are assigned to `ego_loc`.

```
1. def lateral_control(vehicle, waypoint, k_p_s, k_d_s, k_i_s):
2.     ego_loc = vehicle.get_transform().location
3.     ego_loc = np.array([ego_loc.x, ego_loc.y, 0])
4.
5.     v_vec = vehicle.get_transform().get_forward_vector()
6.     v_vec = np.array([v_vec.x, v_vec.y, 0.0])
```

- **w\_loc**: stores the current waypoint being tracked by the algorithm in `[x,y]` form.
- **w\_vec**: computes the vector from the car to waypoint.
- **wv\_linalg**: is used to compute the error in radians between the target waypoint vector and the forward vector
- **\_dot**: is the error in radians that is calculated using the dot product and the magnitudes of `w_vec` and `v_vec`:

```
7.         w_loc = carla.Location(x=waypoint[0], y=waypoint[1])
8.
9.         w_vec = np.array([w_loc.x - ego_loc[0], w_loc.y - ego_loc[1], 0.0])
10.
11.        wv_linalg = np.linalg.norm(w_vec) * np.linalg.norm(v_vec)
12.
13.        _dot = math.acos(np.clip(np.dot(w_vec, v_vec) / (wv_linalg), -1.0,
1.0))
```

To determine whether the vehicle is to the left or right of the desired path (i.e., the sign of the error), the cross product between the forward vector and waypoint vector is calculated:

The sign of the third component of the cross product (z-component) determines whether the angle should be positive (to the right) or negative (to the left). If the z-component is negative, the angle is inverted:



Thus, the lateral error (`_dot`) is a signed angle representing how much the vehicle needs to steer to align itself with the waypoint

```
14. cross = np.cross(v_vec, w_vec)
15.     if cross[2] < 0:
16.         _dot *= -1.0
```

## 5.4. Qt SCXML Functions for DDT fallback

### 5.4.1. Distance to shoulder

This function has a simple objective, its main goal is to compute the distance the ADS vehicle is from the road shoulder, it is important to know how far is the vehicle, because in this manner the filtering of the data from the lidar is enabled when the vehicle is at a specific distance from the road shoulder, and determine whether a vehicle is present or not.

Its inputs are `wp` (Waypoints in the route) used to store the position of the shoulder and vehicle (to get the current position)

```
17. def distance_to_shoulder(wp, vehicle):
18.     v_pos = vehicle.get_transform().location
19.     wp = wp[35]
20.     wp = wp.location
21.
22.     w_vec = np.array([wp.x - v_pos.x, wp.y - v_pos.y,
23.                      0.0])
24.
25.     dist = math.sqrt(w_vec[0] **2 + w_vec[1] **2 + w_vec[2] ** 2)
```

### 5.4.2. Get route.

This function aims at feeding the navigation algorithm with the appropriate route given the situation the ADS vehicle is in, 2 routes are designed a priori, one route is used under normal navigation of the ADS vehicle, while the other route is used only under DDT-fallback, and this route contains the road shoulders.

When the vehicle enters in DDT-fallback, it will switch and follow the `ddt_route` and will park in the road shoulder if no other vehicle is occupying the place, if a vehicle is present, meaning it was detected by the lidar, the route is changed to the original one, before the car enters in the road shoulder, thus continuing on the main route, until a safe free road shoulder is found.

```

1. def get_route(ddt_flag, route, route_ddt, car_det):
3.
4.     if ddt_flag == 1 and car_det == 0:
5.         current_route = route_ddt
6.     elif ddt_flag == 1 and car_det == 1:
7.         current_route = route
8.     else:
9.         current_route = route
10.
11.     return current_route

```

### 5.4.3. Lidar data

Getting lidar data from the sensor in Qt is more straightforward since we are working in python, manipulating the data is simpler, the first step to get the 3D point cloud data is to initialize the and its properties.

```

12. lidar_blueprint = bp_lib.find('sensor.lidar.ray_cast')
13.     lidar_blueprint.set_attribute('channels', str(64))
14.     lidar_blueprint.set_attribute("points_per_second", str(56000))
15.     lidar_blueprint.set_attribute("rotation_frequency", str(100))
16.     lidar_blueprint.set_attribute("range", str(50))

```

In the following section the sensor is spawned and attached to our vehicle, the next step is reading the data.

```

17. lidar_init_trans = carla.Transform(carla.Location(z=2.5))
18.     lidar = world.spawn_actor(lidar_blueprint, lidar_init_trans,
    attach_to=vehicle)

```

The information of the LIDAR measurement is encoded 4D points. Being the first three, the space points in xyz coordinates and the last one intensity loss during the travel.

In the following line of code we enable the sensor and we can start reading information from it

```

19. lidar.listen(lambda point_cloud_data: lidar_data(point_cloud_data))

```

Inside the function lidar\_data the 3D point cloud data is processed and filtered to detect other vehicles inside our region of interest, being it the road shoulder.

First we store all the points in separate coordinates x,y and z and compute the distance from the lidar to the object the ray hit.

```

20. def lidar_data(point_cloud_data):
21.     global car_det
22.     distance_name_data = {}
23.     for detection in point_cloud_data:
24.         x = detection.point.x
25.         y = detection.point.y
26.         z = detection.point.z # z can be used for elevation if needed
27.
28.         # Calculate the distance (optional, for reference)
29.         distance = math.sqrt(x**2 + y**2 + z**2)
30.

```

In a similar manner to that used in matlab we filter the data in each coordinate, being it our box of detection.

```

31. if ((0 < angle_deg < 45) and (-2.2 < z < 0) and (0 < y < 3.25) and (22 <
    dist < 25) and (car_det == 0)):
32.     car_det = 1
33.     print('Distance: ', distance, ' Angle: ', angle_deg, ' Car Det
    = 1')
34.     elif ((dist > 50) and (car_det == 1)):
35.         car_det = 0
36.         print('Car_det = 0 ')

```

## 6. Results

In this section the most relevant results associated with the proposed DDT fallback strategy are analyzed. The proposed fallback strategy performance is evaluated in two scenarios, stopping when the road shoulder is free and continuing to the next permitted stop due to no space availability.

In all scenarios the performance relevant failure is triggered at the same time for each test, after this point the fallback strategy will take control of the automated vehicle and lead it to a minimum risk condition.

The performance data in the 2 scenarios are shown in figures 6.1 through 6.4, In figures 6.1 and 6.2 which represent the vehicle velocity during the complete fallback maneuver, the vertical dashed lines represent the starting point of the DDT fallback strategy at about 20s, triggering the vehicle to enter in degraded mode for velocity reducing its velocity from 20 km/h to 10km/h and the initialization of the braking phase when the vehicle has reached the emergency shoulder.

Second, the velocity and distance figures 6.1 6.2 and 6.3 show that when the failure occurs, the vehicle enters degraded mode for velocity, reducing its target speed to 10 km/h, in this state the control algorithm will look for the next permitted stop, and park if the emergency shoulder is free.

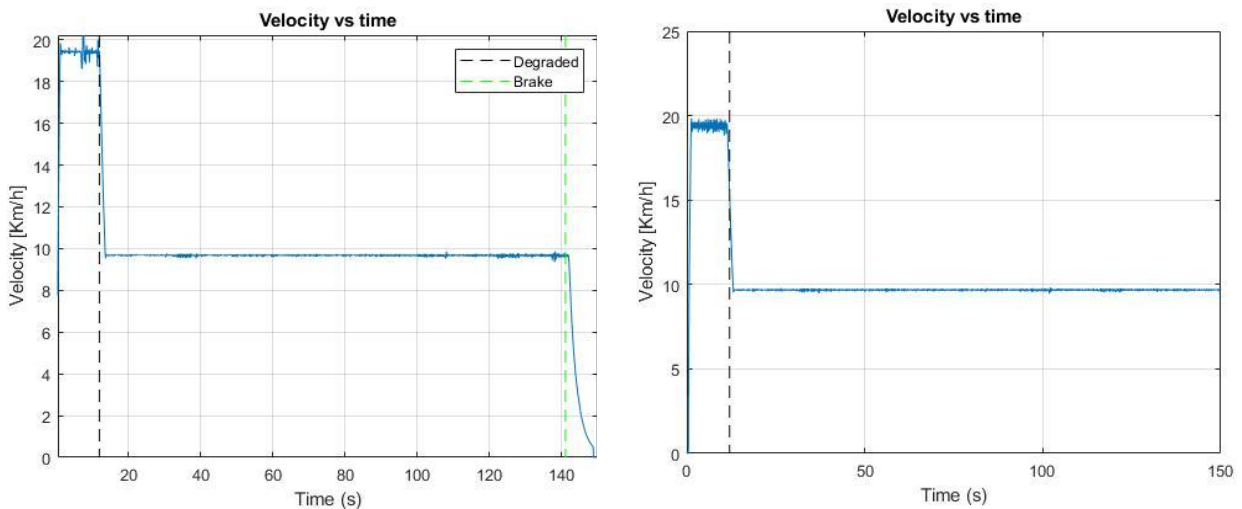


Figure 6. 1 Velocity when parking is permitted vs velocity when parking is not permitted.

The next performance data used to evaluate the fallback strategy is the distance from the vehicle to the emergency shoulder used for stopping, these distances are computed by the control algorithm to ensure the vehicle stopped at the required location.

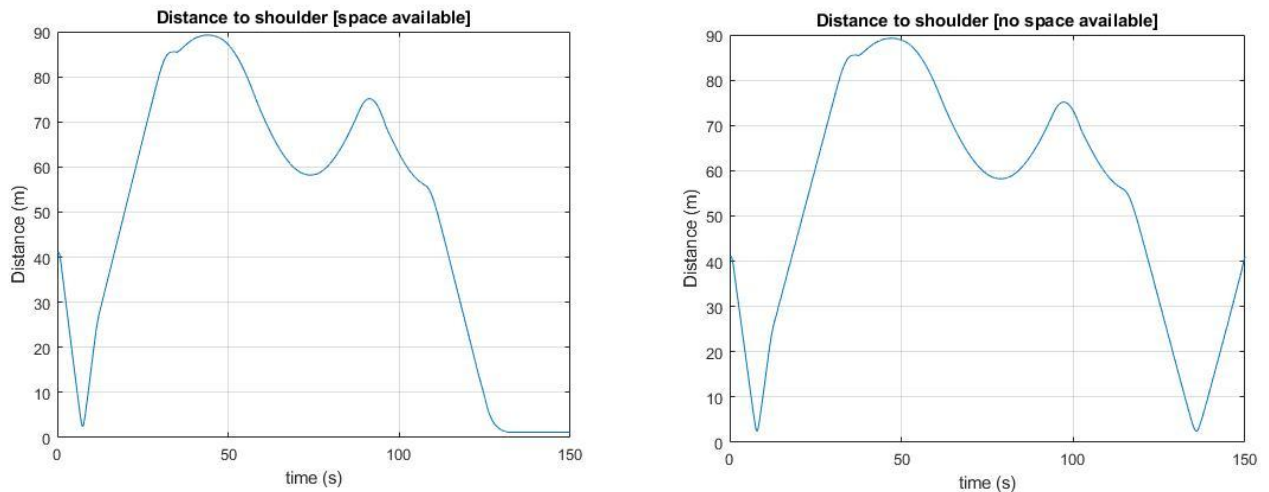


Figure 6. 3 Distance from vehicle to emergency shoulder.

The computational cost to run the control algorithm for MATLAB and Qt are shown, specifically the CPU time and the framerate of the CARLA simulator during the simulation, which are important metrics to deploy the control algorithm in specific hardware, it can be noted also that Qt was about 20% faster than MATLAB while running the control algorithm under the proposed DDT Fallback scenario.

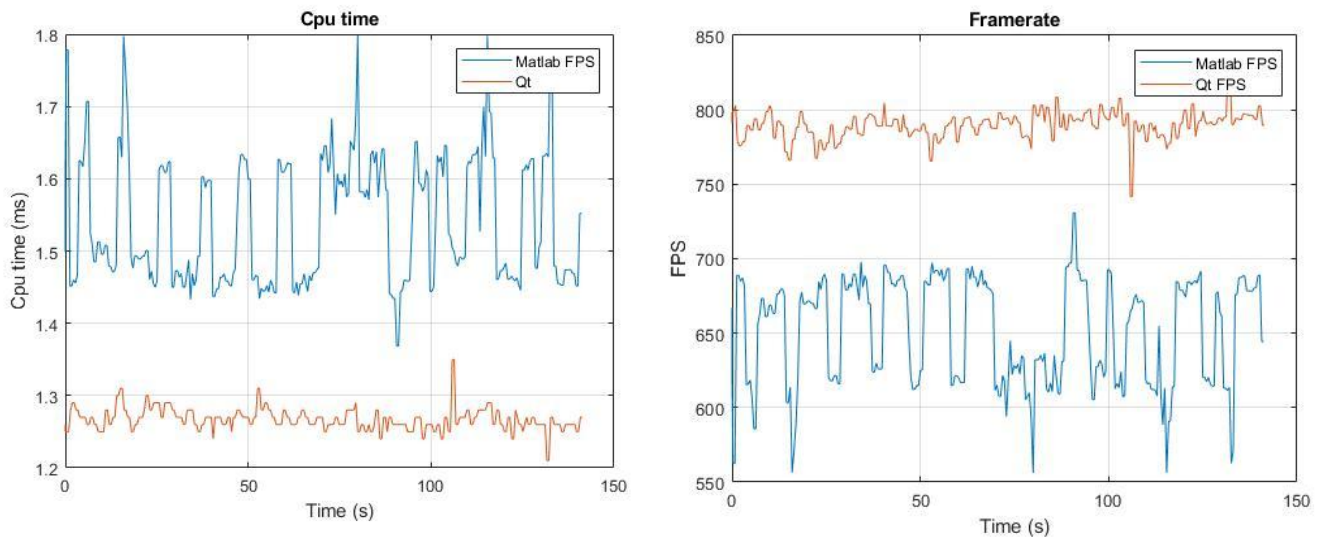


Figure 6. 2 Comparison of CPU time and framerate

In figure 6.4, Qt showed to be lighter and have more stable framerate, having 825 FPS as maximum, 741 FPS as minimum and average of 789 FPS, while MATLAB had higher swings in performance, having 730 FPS as maximum, 556 FPS as minimum and average of 650 FPS.

It is important to remember that both the simulator and the control algorithm are running in the same system (CPU: R5600x, GPU: 3060ti and 16Gb of ram), which can show higher CPU times than expected, a further test could be running the control algorithm and the simulator in separated hardware for further testing and demonstrate that this approach can be implemented in real time.

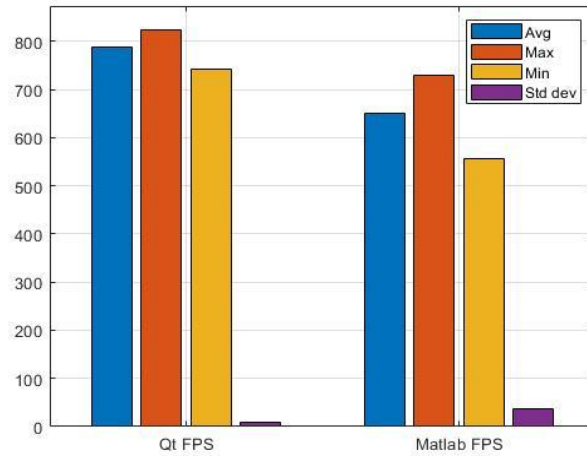


Figure 6. 4 Avg, max, min and std deviation.

From these figures several conclusions can be drawn, first, the design and implementation of the fallback strategy with MATLAB was able to complete and follow the required strategy to park, as well as detecting for other vehicles in the emergency shoulder, this design was then ported to Qt with python and SCXML a lighter program in which the same control strategy was used, and giving similar results to MATLAB but with a lower computational cost as seen in figure 6.4.

## **7. Conclusion and Future developments**

### **7.1 Conclusion**

Although the research and development in automated driving systems has considerably helped the implementation of higher SAE automation levels, current control architectures rely on the driver as a backup in case of systems failures. Moreover, hardware redundancy is the usual plan of action to mitigate problems derived from failures in software or hardware.

The present work targets the issue of the vehicle bringing itself to a safe state, and reach a minimum risk condition in degraded mode, meaning not all systems are operational, the vehicle reduces its speed in the current lane, and the system focuses on following the route until it reaches a road shoulder, and executes a safe stop, in case the road shoulder is occupied the vehicle continues the main route until the road shoulder is free, and performs a safe stop.

The first scenario (parking when the emergency shoulder is free), is the simplest one to perform by the automated vehicle, once degraded mode is enabled the control algorithm looks for the next emergency shoulder and moving through until the maneuver is complete, in the second scenario (vehicle is occupying the road shoulder), the degraded state is activated at the same time but in this case the vehicle will continue to drive to the next available shoulder and park if it is free.

The control architecture is based by lateral and longitudinal PIDs are capable of maneuvering the vehicle and the model based approach designed in MATLAB and then ported to Qt with python and SCXML proved itself robust against different scenarios, following the DDT fallback proposed, first, when the parking place is completely free the vehicle is capable of performing the lane change maneuver and stopping, as well as when there is a car parked in the road shoulder, the ADS vehicle was capable of continuing the route.

### **7.2 Future developments**

There are still several possible ways to keep improving and advancing in DDT fallback strategies, the first is to test the proposed strategy in Bylogix's VeGA to analyze how the proposed strategy behaves in a real urban environment, next, is to add more functionality to the DDT fallback strategy to try to adapt to more diverse scenarios and types of failures, this would make the vehicle more redundant in its systems and mitigate the damage a failure could cause during normal driving.

On the software side, more complexity could be added to some functions to make them even more robust and to rely less on sensor data, for example in the navigation control algorithm, instead of using GNSS data, a position estimator could be implemented, in cases where connectivity is low, but being aware that the estimated position will drift over time and corrections would be needed if the vehicle drives for a long time.

As a closing remark, this thesis marks the ending of a long journey at Politecnico di Torino, as any new task, it faced a lot of difficulties at the beginning that were tackled one by one with a good amount of success to keep advancing on the project and addressing all problems.



## Appendix A: CARLA environment

```
1.     classdef CarlaEnvironment < matlab.System &
matlab.system.mixin.Propagates
2.     %
3.     % This template includes the minimum set of functions required
4.     % to define a System object with discrete state.
5.
6.     % Public, tunable properties
7.     properties
8.         throttle_input = 0;
9.         brake_input = 0;
10.        steering_angle_input = 0;
11.        upper_thresholds = [25, 3, 0]; % Must have the same number of columns
    as 'matrix'
12.        lower_thresholds = [0, 0, -1.5];
13.    end
14.
15.    properties(DiscreteState)
16.    end
17.
18.    % Pre-computed constants
19.    properties(Access = private)
20.        car;
21.        car2;
22.        sensor;
23.        moduleLidar;
24.        car_det=0;
25.    end
26.
27.    methods(Access = protected)
28.        function setupImpl(obj)
29.            insert(py.sys.path,
int32(0), 'C:\CARLA_0.9.14\PythonAPI\carla\dist\carla-0.9.14-py3.7-win-
amd64.egg')
30.            py.importlib.import_module('carla')
31.
32.            % Perform one-time calculations, such as computing constants
33.            port = int16(2000);
34.            client = py.carla.Client('localhost', port);
35.            client.set_timeout(10.0);
36.            world = client.get_world();
37.
38.            blueprint_library = world.get_blueprint_library();
39.            car_list = py.list(blueprint_library.filter("model3"));
40.            car_bp = car_list{1};
41.            spawn_point = world.get_map().get_spawn_points();
42.            start_point = spawn_point{73};
43.            start_point.location.x = -27;
44.            start_point.location.y = 69.7;
45.            start_point.rotation.yaw = 0.073;
46.            obj.car = world.spawn_actor(car_bp, start_point);
47.            pause(1)
48.
49.            car2_transform = py.carla.Transform(py.carla.Location(7.97, 72.75,
0.5),obj.car.get_transform().rotation);
50.            obj.car2 = world.spawn_actor(car_bp, car2_transform);
51.
```

```

52.         %% Move spectator behind the car
53.         spectator = world.get_spectator();
54.         vehicle_transform = obj.car.get_transform();
55.         location_offset = py.carla.Location(-10, 0, 2.5);
56.         transform =
    py.carla.Transform(vehicle_transform.transform(location_offset),
    vehicle_transform.rotation);
57.         spectator.set_transform(transform);
58.
59.         % Lidar
60.         blueprint =
    world.get_blueprint_library().find('sensor.lidar.ray_cast');
61.         blueprint.set_attribute('points_per_second', '25000');
62.         blueprint.set_attribute('range', '50');
63.         blueprint.set_attribute('upper_fov', '45.0');
64.         blueprint.set_attribute('lower_fov', '-30.0');
65.         blueprint.set_attribute('rotation_frequency', '20')
66.         blueprint.set_attribute('channels', '64')
67.
68.         transform = py.carla.Transform(py.carla.Location(pyargs('x',0.8,
    'z',1.7)));
69.         obj.sensor = world.spawn_actor(blueprint, transform,
    pyargs('attach_to',obj.car));
70.         pause(0.5)
71.         obj.moduleLidar = sensorBind(obj.sensor, 'sensor', 'lidar',
    'array');
72.
73.         obj.car_det = 0;
74.     end
75.
76.     function [velocity, car_det, curr_pos] = stepImpl(obj, throttle_input,
    brake_input, steering_angle_input, lidar_search)
77.         pause(0.001);
78.         x_vel = obj.car.get_velocity.x;
79.         y_vel = obj.car.get_velocity.y;
80.         z_vel = obj.car.get_velocity.z;
81.         velocity = 3.6*sqrt((x_vel)^2 + (y_vel)^2 + (z_vel)^2);
82.
83.         v_vec = obj.car.get_transform().get_forward_vector();
84.         ego_trans = obj.car.get_transform().location;
85.         x_pos = ego_trans.x;
86.         y_pos = ego_trans.y;
87.         curr_pos = [x_pos y_pos 0 v_vec.x v_vec.y 0];
88.
89.         control = obj.car.get_control();
90.         control.steer = 0;
91.         control.throttle = throttle_input;
92.         control.brake = brake_input;
93.         control.steer = steering_angle_input;
94.         obj.car.apply_control(control);
95.
96.         lidarData = single(py.getattr(obj.moduleLidar,'array'));
97.         lidarData = double(lidarData(:, 1:3)); % Convert to double and
    extract x, y, z coordinates
98.         point_cloud = lidarData;
99.         [m,n]=size(point_cloud);
100.         within_range_mask = (point_cloud >= obj.lower_thresholds) &
    (point_cloud <= obj.upper_thresholds);
101.

```

```

102.         % Filter out elements not meeting the threshold
103.         valid_rows = all(within_range_mask, 2);
104.         filtered_rows = point_cloud(valid_rows, :);
105.         distances = sqrt(sum(filtered_rows.^2, 2));
106.
107.         if (distances < 25) & (obj.car_det == 0) & lidar_search == 1
108.             obj.car_det = 1;
109.         elseif lidar_search == 0 && obj.car_det == 1
110.             obj.car_det = 0;
111.         end
112.
113.         car_det = obj.car_det;
114.     end
115.
116.     function [velocity, car_det, curr_pos] = getOutputSizeImpl(~)
117.         velocity = [1 1];
118.         car_det = [1 1];
119.         curr_pos = [1 6];
120.     end
121.
122.     function [velocity, car_det, curr_pos] = getOutputDataTypeImpl(~)
123.         velocity = 'double';
124.         car_det = 'double';
125.         curr_pos = 'double';
126.     end
127.
128.     function [velocity, car_det, curr_pos] = isOutputComplexImpl(~)
129.         velocity = false;
130.         car_det = false;
131.         curr_pos = false;
132.     end
133.
134.     function [velocity, car_det, curr_pos] = isOutputFixedSizeImpl(~)
135.         velocity = true;
136.         car_det = true;
137.         curr_pos = true;
138.     end
139.
140.     function resetImpl(~)
141.     end
142. end
143.
144. methods(Access= public)
145.     function delete(obj)
146.         % Delete the car from the Carla world
147.         if ~isempty(obj.car)
148.             obj.car.destroy();
149.         end
150.         if ~isempty(obj.car2)
151.             obj.car2.destroy();
152.         end
153.         if ~isempty(obj.sensor)
154.             obj.sensor.destroy();
155.         end
156.     end
157. end
158. end

```

## Appendix B: SCXML code generated with Qt state machine

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" binding="early"
   xmlns:qt="http://www.qt.io/2015/02/scxml-ext" name="testcarla"
   qt:editorversion="15.0.1" datamodel="ecmascript" initial="Run">
3.   <qt:editorinfo initialGeometry="459.12;107;-20;-20;40;40"/>
4.   <datamodel>
5.     <data id="current_speed" expr="0"/>
6.     <data id="error" expr="0"/>
7.     <data id="target_speed" expr="20"/>
8.     <data id="cmd_throttle" expr="0"/>
9.     <data id="k_p" expr="0.4"/>
10.    <data id="ddt_flag" expr="0"/>
11.    <data id="cmd_brake" expr="0"/>
12.    <data id="WpIndex" expr="0"/>
13.    <data id="car_det" expr="0"/>
14.  </datamodel>
15.  <state id="Run">
16.    <qt:editorinfo geometry="174.82;323.56;-178;-50;924.59;505.54"
   scenegeometry="174.82;323.56;-3.18;273.56;924.59;505.54"/>
17.    <state id="Control">
18.      <qt:editorinfo geometry="86.54;70.77;-60;-50;259.03;368.51"
   scenegeometry="261.36;394.33;201.36;344.33;259.03;368.51"/>
19.      <onentry>
20.        <send event="GetSpeed"/>
21.        <assign location="error" expr="target_speed-current_speed"/>
22.        <assign expr="(k_p*error)" location="cmd_throttle"/>
23.        <send event="GetSteer"/>
24.        <send event="ddt_flag" delay="20s"/>
25.        <send event="GetRouteAndLidar"/>
26.        <send event="GoToStop"/>
27.        <send event="Update" delay="10ms"/>
28.      </onentry>
29.      <onexit>
30.        <if cond="cmd_throttle > 1">
31.          <assign location="cmd_throttle" expr="1"/>
32.          <elseif cond="cmd_throttle<0"/>
33.            <assign location="cmd_throttle" expr="0"/>
34.          </if>
35.          <send event="sendCommand">
36.            <param name="cmd_throttle" expr="cmd_throttle"/>
37.            <param name="cmd_brake" expr="cmd_brake"/>
38.            <param name="ddt_flag" expr="ddt_flag"/>
39.          </send>
40.          <if cond="ddt_flag == 1">
41.            <send event="GoToParkState"/>
42.          </if>
43.        </onexit>
44.        <transition type="external" event="ddt_flag" target="ddt_flag"
   cond="ddt_flag != 1">
45.          <qt:editorinfo startTargetFactors="93.43;37.12"/>
46.        </transition>
47.        <transition type="external" event="GoToStop" target="Stop"
   cond="WpIndex > 38 && WpIndex < 50 && car_det == 0
   && ddt_flag == 1">
48.          <qt:editorinfo endTargetFactors="20.99;7.01"/>
49.        </transition>
```

```

50.     </state>
51.     <transition type="internal" event="Update" target="Run"/>
52.     <transition type="internal" event="SendSpeed">
53.         <assign location="current_speed" expr="_event.data['var']"/>
54.     </transition>
55.     <transition type="internal" event="Index">
56.         <assign location="WpIndex" expr="_event.data['var']"/>
57.     </transition>
58.     <state id="ddt_flag">
59.         <qt:editorinfo geometry="515.85;123.46;-60;-50;172;188"
scenegeometry="690.67;447.02;630.67;397.02;172;188"/>
60.         <onentry>
61.             <assign location="target_speed" expr="10"/>
62.             <assign location="ddt_flag" expr="1"/>
63.             <send event="RunControl"/>
64.             <log label="ddt flag" expr="ddt_flag"/>
65.             <log label="Target Speed:" expr="target_speed"/>
66.         </onentry>
67.         <transition type="external" event="RunControl" target="Control">
68.             <qt:editorinfo endTargetFactors="89.18;66.14"
startTargetFactors="44.77;86.13"/>
69.         </transition>
70.     </state>
71.     <transition type="internal" event="car_det">
72.         <assign expr="_event.data['var']" location="car_det"/>
73.     </transition>
74. </state>
75. <state id="Stop">
76.     <qt:editorinfo geometry="355.67;1010.62;-60;-50;189.03;142"
scenegeometry="355.67;1010.62;295.67;960.62;189.03;142"/>
77.     <onentry>
78.         <assign location="cmd_throttle" expr="0"/>
79.         <assign location="cmd_brake" expr="0.5"/>
80.         <send event="sendCommand">
81.             <param name="cmd_brake" expr="cmd_brake"/>
82.             <param name="cmd_throttle" expr="cmd_throttle"/>
83.             <param name="ddt_flag" expr="ddt_flag"/>
84.         </send>
85.     </onentry>
86. </state>
87. </scxml>

```

## Appendix C: Python code used to run CARLA with Qt

```
1. from PySide6.QtCore import QObject, QApplication, Slot, QTimer, SLOT
2. from PySide6.QtScxml import QScxmlStateMachine, QScxmlEvent
3. import time
4. import numpy as np
5. from collections import deque
6. import sys
7. import glob
8. import os
9. import math
10.
11. waypoint_index = 1
12.
13. try:
14.     sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
15.         sys.version_info.major,
16.         sys.version_info.minor,
17.         'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
18. except IndexError:
19.     pass
20.
21. import carla
22.
23. # Starting the simulation
24. client = carla.Client('localhost', 2000)
25. world = client.get_world()
26.
27. # Spawning the car
28. bp_lib = world.get_blueprint_library()
29. vehicle_bp = bp_lib.filter('*mini*')[0]
30. spawn_points = world.get_map().get_spawn_points()
31. #start_point = spawn_points[60]
32. start_point = carla.Transform(carla.Location(x=-27,y=69.7,
33.     z=0.5),carla.Rotation(yaw=0.073))
34. vehicle = world.spawn_actor(vehicle_bp, start_point)
35.
36. spectator = world.get_spectator()
37. transform =
38.     carla.Transform(vehicle.get_transform().transform(carla.Location(x=-10,
39.     z=2.5)),vehicle.get_transform().rotation)
40. spectator.set_transform(transform)
41.
42. #vehicle2_transform = carla.Transform(carla.Location(x=7.97,y=72.75,
43.     z=0.5),vehicle.get_transform().rotation)
44. #vehicle2 = world.spawn_actor(vehicle_bp, vehicle2_transform)
45.
46. route = np.loadtxt('route.txt')
47. route_ddt = np.loadtxt('route_ddt.txt')
48. current_route = route
49. e_buffer = deque(maxlen=10)
50. offset = 0
51. k_p_s = 0.4
52. k_d_s = 0
53. k_i_s = 0
54. car_det = 0
55. current_velocity = 0
56. target_velocity = 20
```

```

53. ddt_flag = 0
54. foo = 0
55. error_buffer = deque(maxlen=10)
56. k_p=0.4
57. k_d=0
58. k_i=0
59. dt=0.01
60. collisions_data = 0
61. car_det=0
62.
63.
64. def lateral_control(vehicle, waypoint, k_p_s, k_d_s, k_i_s):
65.     ego_loc = vehicle.get_transform().location
66.     ego_loc = np.array([ego_loc.x, ego_loc.y, 0])
67.
68.     v_vec = vehicle.get_transform().get_forward_vector()
69.     v_vec = np.array([v_vec.x, v_vec.y, 0.0])
70.
71.     # Get the vector vehicle-target_wp
72.     if offset != 0:
73.         # Displace the wp to the side
74.         w_tran = waypoint#.Location#transform
75.         r_vec = w_tran.get_right_vector()
76.         w_loc = w_tran.location + carla.Location(x=offset*r_vec.x,
77.
78.         y=offset*r_vec.y)
79.     else:
80.         w_loc = carla.Location(x=waypoint[0], y=waypoint[1])
81.         w_vec = np.array([w_loc.x - ego_loc[0], w_loc.y - ego_loc[1], 0.0])
82.
83.         wv_linalg = np.linalg.norm(w_vec) * np.linalg.norm(v_vec)
84.
85.         if wv_linalg == 0:
86.             _dot = 1
87.         else:
88.             _dot = math.acos(np.clip(np.dot(w_vec, v_vec) / (wv_linalg), -1.0,
89.             1.0))
90.
91.         cross = np.cross(v_vec, w_vec)
92.         if cross[2] < 0:
93.             _dot *= -1.0
94.
95.         e_buffer.append(_dot)
96.         if len(e_buffer) >= 2:
97.             de = (e_buffer[-1] - e_buffer[-2]) / dt
98.             ie = sum(e_buffer) * dt
99.         else:
100.            de = 0.0
101.            ie = 0.0
102.            steer_cm = np.clip((k_p_s * _dot) + (k_d_s * de) + (k_i_s * ie),
103.            -1.0, 1.0)
104.            return steer_cm
105.
106.     def compute_speed(vehicle):
107.
108.         current_speed = vehicle.get_velocity()

```

```

109.         current_speed = 3.6 * (current_speed.x**2 + current_speed.y**2 +
current_speed.z**2)**0.5
110.
111.         return current_speed
112.
113.     def dist_to_wp(vehicle, waypoint):
114.
115.         global waypoint_index
116.
117.         if waypoint_index == (len(waypoint) - 1):
118.             waypoint_index = 1
119.
120.         if
vehicle.get_location().distance(carla.Location(x=waypoint[waypoint_index,0]
, y=waypoint[waypoint_index,1])) < 2.0:
121.             waypoint_index += 1
122.
123.         return waypoint_index
124.
125.     def get_route(ddt_flag, route, route_ddt, car_det):
126.
127.         if ddt_flag == 1 and car_det == 0:
128.             current_route = route_ddt
129.
130.         elif ddt_flag == 1 and car_det == 1:
131.             current_route = route
132.
133.         else:
134.             current_route = route
135.
136.         return current_route
137.
138.     def distance_to_shoulder(wp, vehicle):
139.         global dist
140.         v_pos = vehicle.get_transform().location
141.         wp = wp[35]
142.         #print(wp)
143.         #wp = wp.Location
144.
145.         w_vec = np.array([wp[0] - v_pos.x,
146.                             wp[1] - v_pos.y,
147.                             0.0])
148.
149.         dist = math.sqrt(w_vec[0] **2 + w_vec[1] **2 + w_vec[2] ** 2)
150.
151.         #print(dist)
152.         return dist
153.
154.     def lidar_data(point_cloud_data):
155.         global car_det, dist
156.
157.         dist = distance_to_shoulder(route_ddt, vehicle)
158.
159.         #distance_name_data = {}
160.         for detection in point_cloud_data:
161.             x = detection.point.x
162.             y = detection.point.y
163.             z = detection.point.z # z can be used for elevation if
needed

```



```

164.
165.         # Calculate the distance (optional, for reference)
166.         distance = math.sqrt(x**2 + y**2 + z**2)
167.
168.         # Calculate the angle (in radians)
169.         angle_rad = math.atan2(y, x) # atan2(y, x) gives the angle
        from the x-axis
170.
171.         # Convert angle to degrees
172.         angle_deg = math.degrees(angle_rad)
173.
174.         if ((0 < angle_deg < 45) and (-2.2 < z < 0) and (0 < y <
        3.25) and (22 < dist < 25) and (car_det == 0)):
175.             car_det = 1
176.             print('Car detected at, Distance: ', distance, ' Angle:
        ', angle_deg, ' Car Det = 1')
177.             elif ((dist > 50) and (car_det == 1)):
178.                 car_det = 0
179.                 print('No car detected, (Car_det = 0) ')
180.
181.
182.         # Set Up sensors
183.         lidar_blueprint = bp_lib.find('sensor.lidar.ray_cast')
184.         lidar_blueprint.set_attribute('channels', str(64))
185.         lidar_blueprint.set_attribute("points_per_second",str(56000))
186.         lidar_blueprint.set_attribute("rotation_frequency",str(100))
187.         lidar_blueprint.set_attribute("range",str(50))
188.
189.         lidar_init_trans = carla.Transform(carla.Location(z=2.5))
190.         lidar = world.spawn_actor(lidar_blueprint, lidar_init_trans,
        attach_to=vehicle)
191.
192.         lidar.listen(lambda point_cloud_data: lidar_data(point_cloud_data))
193.
194.         class ScxmlHandler(QObject):
195.             def __init__(self, scxml_file, parent=None):
196.                 super().__init__(parent)
197.                 self.stateMachine = QScxmlStateMachine.fromFile(scxml_file)
198.
199.                 # Check if the state machine is initialized properly
200.                 if not self.stateMachine:
201.                     print("Failed to initialize the state machine. Check
        SCXML file and syntax.")
202.                     return # Exit initialization if state machine failed to
        Load
203.
204.                 # Connect event from SCXML to a slot in Python using the
        correct signature
205.                 self.stateMachine.connectToEvent("GetSpeed", self,
        SLOT("handleDataFromScxml(QScxmlEvent)"))
206.                 self.stateMachine.connectToEvent("sendCommand", self,
        SLOT("handleDataFromScxml(QScxmlEvent)"))
207.                 self.stateMachine.connectToEvent("GetSteer", self,
        SLOT("handleDataFromScxml(QScxmlEvent)"))
208.                 self.stateMachine.connectToEvent("GetRouteAndLidar", self,
        SLOT("handleDataFromScxml(QScxmlEvent)"))
209.                 # Connect the state machine's "Log" signal to the
        outputReceived slot
210.                 #self.stateMachine.Log.connect(self.outputReceived)

```

```

211.
212.         # Start the state machine before connecting slots to ensure
        all signals are caught
213.         self.stateMachine.start()
214.
215.         # Timer to check active states every second
216.         #self.timer = QTimer(self)
217.         #self.timer.timeout.connect(self.printActiveStates)
218.         #self.timer.start(2000) # Check every 1000 milliseconds (1
        second)
219.
220.     @Slot(QScxmlEvent)
221.     def handleDataFromScxml(self, event: QScxmlEvent):
222.         global foo, ddt_flag, get_route, current_route, car_det,
        dist
223.         #print("Received SCXML event:", event.name())
224.         data = event.data() # data() returns a QVariantMap,
        accessed like a dictionary in Python
225.
226.         #prt(data)
227.         #print(data["cmd_throttle"])
228.         if event.name() == 'GetSpeed':
229.             current_speed = compute_speed(vehicle)
230.             #print(f'Entered get speed, Speed is [{current_speed}]')
231.             self.send_event_with_data(current_speed, "SendSpeed")
232.
233.             #if event.name() == 'GetSteer':
234.             #    indx = dist_to_wp(vehicle, route)
235.             #    cmd_steer = lateral_control(vehicle, route[indx],
        k_p_s=0.4, k_d_s=0, k_i_s=0)
236.
237.             if event.name() == 'GetRouteAndLidar':
238.                 current_route = get_route(ddt_flag, route, route_ddt,
        car_det)
239.
240.             if event.name() == 'sendCommand':
241.                 #global route, ddt_flag # Declare global variables
242.
243.                 cmd_throttle = data["cmd_throttle"]
244.                 cmd_brake = data["cmd_brake"]
245.                 ddt_flag = data["ddt_flag"]
246.                 #print('recieved cmd throttle', cmd_throttle)
247.
248.                 """if ddt_flag == 1 and foo == 0:
249.                     foo = 1
250.                     route = route_ddt
251.                     print("route fallback!)" """
252.
253.
254.                 indx = dist_to_wp(vehicle, current_route)
255.                 self.send_event_with_data(indx, "Index")
256.                 self.send_event_with_data(car_det, "car_det")
257.                 cmd_steer = lateral_control(vehicle,
        current_route[indx], k_p_s=0.4, k_d_s=0, k_i_s=0)
258.                 vehicle.apply_control(carla.VehicleControl(throttle =
        cmd_throttle, steer = cmd_steer, brake = cmd_brake))
259.                 #print(cmd_brake)
260.
261.     @Slot(str, str)

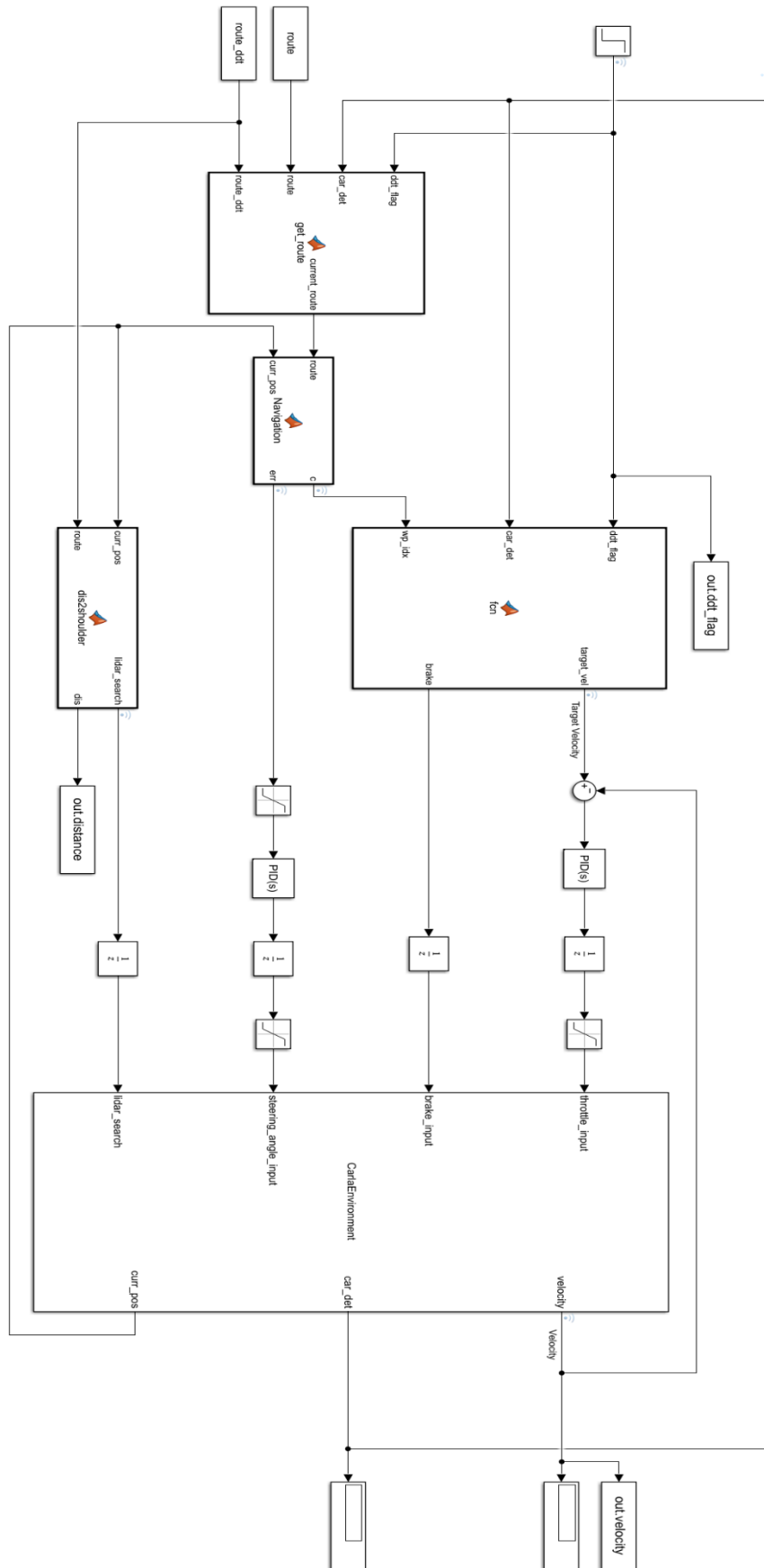
```

```

262.     def outputReceived(self, label, msg):
263.         #Slot that handles the 'Log' signal from the state machine
264.         print(f"Log event - Label: {label}, Message: {msg}")
265.
266.     def printActiveStates(self):
267.         #Prints the currently active states of the state machine
268.         active_states = self.stateMachine.activeStateNames()
269.         print("Current active states:", active_states)
270.
271.     def send_event_with_data(self, value, name):
272.         """Send a custom event with data."""
273.         event = QScxmlEvent()
274.         event.setName(name)
275.         event.setData({"var": value})
276.
277.         # Submit the event with data
278.         self.stateMachine.submitEvent(event)
279.         #print(f"Sent event with value: {value}")
280.
281.     def EventUpdate(self):
282.
283.         event = QScxmlEvent()
284.         event.setName("Update")
285.         # Submit the event with data
286.         self.stateMachine.submitEvent(event)
287.
288.
289. if __name__ == "__main__":
290.     import sys
291.
292.     app = QApplication(sys.argv)
293.     handler = ScxmlHandler("testcarla.scxml")
294.
295.
296.     #handler.send_event_with_data(19)
297.     #handler.EventUpdate()
298.
299.     if not handler.stateMachine:
300.         sys.exit(-1)
301.     sys.exit(app.exec())

```

# Appendix D: Model based design in MATLAB



## Appendix F: MATLAB function for callback of lidar sensor through a python script

```
1. function pyModule = sensorBind(sensor, fileName, sensorType, varName)
2.
3.     %% Function signature
4.     % -----
5.     %                               Inputs
6.     % -----
7.     % sensor           ---> The sensor
8.     % fileName        ---> Name of the python file that will be produced for
9.     %                  sensor binding
10.    %
11.    % varName          ---> Name of the variable that will store the sensor
    data
12.    %
13.    % sensorType      ---> Choose from sensors
14.    %                  1. rgb                      ---> Normal camera
15.    %                  2. grayScale                ---> grayScale image
16.    %                  3. depth                    ---> Gives distances in
    m
17.    %                  4. depthRGB                  ---> RGB
    coded distance
18.    %                  5. semantic_segmentation    ---> Classifies objects
19.    %                  and tags them
    with id
20.    %                  6. semantic_segmentation_rgb ---> Map the id
    to RGB for
21.    %
    visualization
22.    %                  7. lidar                      ---> Gives 3d
    points array
23.    %                  along with
    intensity values
24.    %
25.    % -----
26.    %                               Output
27.    % -----
28.    % pyModule ---> Returns a python module that is responsible for
29.    %                  acquiring the sensor data
30.
31.    %% Check if the sensor is valid
32.    if ~isa(sensor, 'py.carla.libcarla.ServerSideSensor')
33.        error("The provided sensor is not valid\n")
34.    end
35.
36.    %% Create sensor callback binder
37.    if strcmp(sensorType, "rgb") || strcmp(sensorType, "grayScale") ||
    strcmp(sensorType, "depthRGB")
38.        rgb(fileName, sensorType, varName);
39.    elseif strcmp(sensorType, "depth")
40.        depth(fileName, sensorType, varName);
41.    elseif strcmp(sensorType, "semantic_segmentation")
42.        semantic_segmentation(fileName, sensorType, varName);
43.    elseif strcmp(sensorType, "semantic_segmentation_rgb")
44.        semantic_segmentation_rgb(fileName, sensorType, varName);
```

```

45. elseif strcmp(sensorType, "lidar")
46.     lidar(fileName, sensorType, varName);
47. end
48.
49. pyModule = py.importlib.import_module(fileName);
50. eval(strcat("py.", fileName, ".bindSensor(sensor)"));
51.
52. % Takes time for the sensor to receive first data
53. pause(0.25);
54. end

```

Function to generate the python for the lidar sensor, script from MATLAB

```

1. function lidar(fileName, sensorType, varName)
2.
3.     %% Check if input is valid
4.     if ~isstring(fileName) && ~ischar(fileName)
5.         error("File name must a string or char array\n")
6.     end
7.
8.     if ~strcmp(sensorType, "lidar")
9.         error("Wrong sensor selected\n")
10.    end
11.
12.    if ~isvarname(varName)
13.        error("Invalid variable name\n")
14.    end
15.
16.    %% Generate the python file
17.    file = fopen(strcat(fileName, '.py'), 'w');
18.
19.    % Automatically generates a python file containing the sensor call back
20.    % bindings
21.    fprintf(file, 'import numpy as np\n');
22.    fprintf(file, '\n');
23.    fprintf(file, 'def bindSensor(sensor):\n');
24.    fprintf(file, '    sensor.listen(lambda _image:
25.        do_something(_image))\n');
26.    fprintf(file, '\n');
27.    fprintf(file, 'def do_something(_image):\n');
28.    fprintf(file, '    global %s\n', varName);
29.    fprintf(file, '    data = np.frombuffer(_image.raw_data,
30.        dtype="float32")\n');
31.    fprintf(file, '\n');
32.    fprintf(file, '    # Pair up in [x,y,z] format\n');
33.    fprintf(file, '    data = np.reshape(data, (-1,4))\n');
34.    fprintf(file, '\n');
35.    fprintf(file, '    # Convert the data into MATLAB cast compatible
36.        type\n');
37.    fprintf(file, '    %s = np.ascontiguousarray(data)\n', varName);
38.    fclose(file);
39. end

```

## Result python script

```
1. import numpy as np
2.
3. def bindSensor(sensor):
4.     sensor.listen(lambda _image: do\_something(_image))
5.
6. def do\_something(_image):
7.     global array
8.     data = np.frombuffer(_image.raw_data, dtype="float32")
9.
10.    # Pair up in [x,y,z] format
11.    data = np.reshape(data, (-1,4))
12.
13.    # Convert the data into MATLAB cast compatible type
14.    array = np.ascontiguousarray(data)
```

## **Bibliography**

- [1] Reuters, "Americans still don't trust self-driving cars, Reuters/Ipsos poll finds," [Online]. Available: <https://www.reuters.com/article/idUSKCN1RD2QV/>.
- [2] int, SAE, "Taxonomy and Definitions for Terms Related to Driving automation systems for on road motor vehicles," 2021.
- [3] BMW, <https://www.bmw.com/en/automotive-life/autonomous-driving.html>, 2024.
- [4] A. D. a. G. R. a. F. C. a. A. L. a. V. Koltun, "{CARLA}: {An} Open Urban Driving Simulator," *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1-16, 2017.
- [5] QT, "Qt SCXML," 2024. [Online]. Available: <https://doc.qt.io/qt-6/qtscxml-index.html>.
- [6] J. I.-G. P. M. O. H. Yrvann Emzivat, "Dynamic Driving Task Fallback for an Automated Driving System whose Ability to Monitor the Driving Environment has been Compromised," 2018.
- [7] "Proportional–integral–derivative controller," [Online]. Available: [https://es.wikipedia.org/wiki/Controlador\\_PID](https://es.wikipedia.org/wiki/Controlador_PID).
- [8] PROJ, "PROJ coordinate transformation software library," [Online]. Available: <https://proj.org>.
- [9] GPX, 2022. [Online]. Available: <https://gpx.studio//it/> .
- [10] G. W. G. S. a. C. G. F. Group, "Its Definition and Relationships with Local Geodetic Systems. National Center for Geospatial Intelligence Standards," 2014.
- [11] soumyadip, "Building Autonomous Vehicle in Carla: Path Following with PID Control & ROS 2," 2024. [Online]. Available: <https://learnopencv.com/pid-controller-ros-2-carla/#Introduction-to-Lateral-Control>.
- [12] "W3C SCXML".
- [13] Qt. [Online]. Available: <https://doc.qt.io/qtforpython-6/PySide6/QtStateMachine/index.html>.
- [14] C. F. F. Karney, "Algorithms for geodesics," [Online]. Available: <https://doi.org/10.1007/s00190-012-0578-z>.



- [15] K. L. / A. V. Stock, "Alamy," [Online]. Available: <https://www.alamy.it/schema-della-metodologia-di-sviluppo-software-del-modello-v-infografiche-del-processo-del-ciclo-di-vita-fase-di-verifica-convalida-raccolta-dei-requisiti-sistema-image396522569.html>.
- [16] G. S. N. B. C. H. Maxim Serebreni, "Reliability Physics Approach for High-Density Ball Grid Arrays in Autonomous Vehicle Applications," 2019.
- [17] U. D. o. transportation, "A Framework for Automated Driving System Testable Cases and Scenarios," 2018.
- [18] J. P. a. A. Z. Jose Angel Matute-Peaspan, "A Fail-Operational Control Architecture Approach and Dead-Reckoning Strategy in Case of Positioning Failures," 2020.