Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Test Environment for Automated Lane Keeping System Verification

Oskar Krejčí

Supervisor: Ing. Michal Sojka, Ph.D.
May 2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krejčí** Jméno: **Oskar** Osobní číslo: **483579**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Testovací prostředí pro verifikaci systému automatizovaného udržování vozidla v jízdním pruhu (ALKS)

Název bakalářské práce anglicky:

Test Environment for Automated Lane Keeping System Verification

Pokyny pro vypracování:

1) Seznamte se se simulátory pro autonomní řízení automobilů. Např. CARLA a SVL Simulator a s požadavky na systémy ALKS dané předpisem UNECE R157.
2) Vyberte jeden ze simulátorů a připravte v něm scénáře co nejpodobnější těm, popisovaným v UNCE R157.
3) Vytvořte v simulátoru jednoduchou implementaci ALKS, která využívá přesné informace o okolí vozidla poskytované simulátorem (mapa, pozice ostatních vozidel, …) a ukažte, že tato implementace splní požadavky všech testů.
4) Vytvořte druhou implementaci ALKS, která bude místo některých přesných informací používat informace ze simulovaných senzorů, kde bude simulován např. šum nebo jiné nedokonalosti skutečných senzorů. Vyhodnoťte, jak tato implementace splňuje požadavky testu.
5) Výsledky pečlivě zdokumentujte.

Seznam doporučené literatury:

[1] Předpis UNECE R157 http://data.europa.eu/eli/reg/2021/389/oj
[2] CARLA Documentation: https://carla.readthedocs.io/en/0.9.13/
[3] SVL Simulator Documentation: https://www.svlsimulator.com/docs/
[4] A. Tenbrock, A. König, T. Keutgens, and H. Weber, "The ConScenD Dataset: Concrete Scenarios from the highD Dataset According to ALKS Regulation UNECE R157 in OpenX," in 2021 IEEE Intelligent Vehicles Symposium Workshops (IV Workshops), Jul. 2021, pp. 174–181. doi: 10.1109/IVWorkshops54471.2021.9669219.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michal Sojka, Ph.D. vestavěné systémy CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.01.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce:
do konce letního semestru 2022/2023

Ing. Michal Sojka, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Acknowledgements

Hereby I would like to thank my supervisor Ing. Michal Sojka, Ph.D. and my family. Mr. Ing. Michal Sojka, Ph.D. has provided helpful advice and observations during consultations of the thesis. My family has been patient and supportive during the time.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 20. May 2022

Abstract

With the approval of the regulation UNECE R157 there is now a minimal set of scenario based tests, that an ALKS has to pass in order to be allowed into production. However, there are no specific parameters, hence they shall be reasonably selected. To vary them as much as possible in the shortest time, a simulator is used. Using Python 3 and Carla, I implemented an ALKS as well as 3 of the tests to assess it. The tests can be parameterized, providing the total of 90 versions. The ALKS has used a radar as the only sensor and has not passed all the tests, therefore further work is required.

Keywords: autonomous driving, OpenDRIVE, ALKS, Python, Carla, Unreal Engine 4

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Se schválením předpisu UNECE R157 nyní existuje minimální soubor testů založených na scénáři, kterými musí ALKS projít, aby mohl být uveden do výroby. Neexistují však žádné konkrétní parametry, ty proto musí být rozumně vybrány. Pro jejich co největší obměnu v co nejkratším čase se používá simulátor. Pomocí Pythonu 3 a Carla jsem implementoval ALKS a také 3 testy, abych mohl ALKS zhodnotit. Testy lze parametrizovat a poskytují celkem 90 verzí. ALKS používá radar jako jediný senzor a neprošel všemi testy, proto je zapotřebí další práce.

Klíčová slova: autonomní řízení, OpenDRIVE, ALKS, Python, Carla, Unreal Engine 4

Překlad názvu: Testovací prostředí pro verifikaci systému automatizovaného udržování vozidla v jízdním pruhu (ALKS)

Contents

Figures

Tables

Chapter 1

Introduction

In recent decades, automation development intervenes more and more domains of our lives, ranging from factory pipelines through self service cash registers to many smart devices. Some of these are fail-safe, but those which are fail-deadly require intensive testing and verification before being manufactured and sold. This is the case within the automotive area, where especially cars are implementing new systems and functions to make drivers' journey easier and more comfortable. One of the newest functions is so called ALKS, standing for automated lane keeping systems. Such a function, when active, is responsible for the dynamic driving task (DDT), so the driver can focus on secondary tasks. Put in another words, it actively controls the throttle, brake pedal as well as the steering wheel in response to the environmental conditions and traffic around. ALKS is primarily intended to use on straight roads or highways, where the vehicles' path is not curved too much and can not be crossed by any other road user.

With the recent approval of the regulation UNECE R157 [1] (further referred to as the regulation) it is possible to include ALKS into equipment of vehicles sold on the market in over 60 countries, if they pass at least the set of tests mentioned therein. The specific parameters of the tests are not specified, however, so a technical service, conducting the tests, should verify safety and reliability by selecting parameters for them reasonably.

While all the tests are done on real vehicles, of course, they are very well complemented by using simulation tools, where many of possible errors can be identified. This is cheaper, faster and has the advantage of allowing the system in question to be easily tested in most of any conceivable traffic scenarios, common ones as well as edge cases, without modifying an already produced piece.

The goal of this thesis is to present the reader with 3 open source simulation tools and describe a way how to implement an ALKS as well as 3 tests from the regulation in one of them, including the parameterization.

Chapter 2
Background

2.1 ALKS and 6 levels of driving automation

The ALKS is a system designed to take over the dynamic driving task (DDT) and is fully responsible for controlling your vehicle once activated. However, the conditions to fulfill before activation are strict. They are stated in paragraphs 5-9 of the regulation [1] and include the maximum operational speed of 60 km/h, no intrusion on any neighbouring lane, prohibition of pedestrians and cyclists on the road or physical separation of traffic moving in opposite direction, that is to prevent any other road users from cutting across the path of the vehicle. With that being said, ALKS shall not be activated for example in a city with dense road network where a lot of junctions happen to be.

ALKS belongs to level 3 out of the 6 levels of driving automation, with level 0 being no automation and level 5 fully automated[1]. They are defined by the standard J3016_201806 from the Society of automation engineers (SAE)[2] and were adopted by U.S. Department of Transportation. Level 3 is the conditional driving automation, so not only are there conditions to fulfill before activation, but as soon as ALKS evaluates it will not able to control the vehicle for any reason, it shall hand the control over to the driver. How such a transition shall be done is described in the regulation as well. Nevertheless, it shall be possible for the human driver to override ALKS at any instant time.

2.2 ASAM OpenX standards

ASAM stands for Association for standardization of Automation and Measuring systems [2]. It is located in Höhenkirchen, Germany and was founded in 1998 on an initiative of German car manufactures AUDI, BMW, Daimler and Porsche VW. They develop and maintain standards in different areas of automotive industry. In 2018 they added new area to their expertise – highly automated driving – and adopted 3 new standards into it. These are

[1] https://www.synopsys.com/automotive/autonomous-driving-levels.html
[2] https://www.sae.org/standards/content/j3016_201806/

OpenDRIVE, OpenSCENARIO and OpenCRG, together called the OpenX standards. The three standards are complementary to each other and cover both the static and dynamic content of a simulation.

The OpenDRIVE[3] deals with the static one and description of an environment is done using an xml schema, usually all in one file. Each road has its' own reference line along which features of the road are defined, including lanes, traffic signs or objects as parked car or traffic cones. A road can be chopped into sections, which is done for example when the number of lanes changes, e.g. merge lane on a highway, or linked together with others either directly with a link tag or via junction tag if the connection is ambiguous.

OpenSCENARIO[4], however, handles the dynamic one. Scenario files use xml format and have an initial section to set up the scenario (set the weather, put actors on the right spot, etc.) and a main container for all that happens called storyboard. Usually there is only one per scenario and can be divided into 6 more elements: ⟨**Story**⟩, ⟨**Act**⟩, ⟨**ManeuverGroup**⟩, ⟨**Maneuver**⟩, ⟨**Event**⟩ and ⟨**Action**⟩. An action is the simplest thing that can be performed by any actor in the scenario, e.g. acceleration or lane change. They differ depending on which actor they are used with, because you can not execute a lane change with a pedestrian. Actions are grouped into events and so are events into maneuvers etc. To determine if an element should be triggered, conditions of that element are evaluated. They must be wrapped into a ⟨**ConditionGroup**⟩ tag. Logical AND is applied to conditions contained therein and logical OR between ⟨**ConditionGroup**⟩s.

Finally, OpenCRG[5] describes a file format for the road surface description and a method to store them in a layout called curved regular grid, hence CRG. Primary use of this standard lies in tire, vibration or endurance tests. The only possible usage of this standard while simulating ALKS is the option to have precise and realistic rendering of the road texture. Besides that, OpenCRG presents C API and MATLAB API that are equipped sufficiently to handle data in the OpenCRG format. Nevertheless, only OpenDRIVE in version 1.4 was used in this work.

2.3 Roadrunner

Roadrunner is a commercial application with a graphical user interface (GUI) for creating static simulation environments, although it is free for those who have access to Matlab [6]. It has wide range of customization capabilities regarding traffic signals controllers and their timing at junctions, road lanes and their marking. It comes with a library consisting of hundreds of predefined models from many categories, namely traffic signs, traffic light styles, barriers or vegetation.

Besides the predefined assets, user can import custom models or textures by simple drag-and-drop technique into the asset library. OpenDRIVE files

[3] `https://www.asam.net/standards/detail/opendrive/`
[4] `https://www.asam.net/standards/detail/openscenario/`
[5] `https://www.asam.net/standards/detail/opencrg/`

are importable, too. When satisfied with the work, the environment can be exported in two ways: either as a single file, namely OpenDRIVE (.xodr), filmbox (.fbx) or AutoCad (.dxf), or directly bundled for a simulation software or a game engine. Following are available: Carla, Unity engine, Unreal engine, Metamoto, VIRES Virtual Test Drive (VTD).

2.4 Game engines

Game engines are tools primarily used by programmers for fast and easy game development. They could be categorized into orientation, i.e. 2D or 3D oriented, or into target platform, i.e. Windows, Linux etc. Games are often computationally demanding, so the code of a game engine is usually speed-optimized to maintain high frame rate. In the scope of this thesis, I have come across 2 out of tens game engines, which are described in greater detail.

2.4.1 Unreal engine

The Unreal engine is proprietary, cross-platform, 3D game engine written in C++ and made by Epic Games, Inc. It includes an integrated editor, the Unreal Editor, that eases the creation of simulation environment. Related features to highlight are:

- **World partitions** – In case of a large map, rendering it all at once may be difficult and unnecessary. The solution is chopping the map into a grid of sectors and rendering those needed.
- **World layers** – Having screen overcrowded with objects makes it impossible to easily edit any particular one when needed. The concept of layers eases objects editing by dividing them into groups, whose visibility can be toggled by one click.
- **Physics system** – Real physics is of great importance to some vehicle related simulations. The Unreal engine is equipped with a high-performance physics system called Chaos.

2.4.2 Unity

The Unity engine is capable of dealing with both 2D and 3D content, written in C++ as well. It is newer engine than the Unreal engine and automated driving simulation is not its' focus. Instead it revolves around mobile games development. In comparison to the Unreal engine, the support for automated driving simulation is lacking, e.g. real physics simulation is missing. It is more suitable for precise modeling, virtual reality (VR) or computer vision.

2.5 Simulation tools

Simulation tools come either as standalone applications or projects built upon an engine, which eases some computations, so the tool is able to focus on others. Their aim is to identify as many errors as possible before a product is tested in the real world.

2.5.1 Carla

The Carla simulator[6] is an open source software that is being developed to test, train and validate autonomous driving systems, built upon the Unreal engine 4. It is built on the client-server architecture and exposes Python API to control the simulation on the server. It comes with preset maps and models for vehicles, sensors as well as pedestrian figures. Moreover, it is possible to define your own sensors, run synchronous as well as asynchronous simulations or exploit built-in module called the Traffic Manager to create realistic urban traffic in the scene.

2.5.2 SVL Simulator

The SVL Simulator is an open source simulator developed by LG Electronics America R&D Lab, located in California[7]. It supports features similar to Carla, such as a Python API or ROS 2 bridge as well as some specific to it [4] – Apollo 5.0 (open source software for autonomous vehicles), Visual Scenario Editor (VSE), wider range of sensor in comparison to Carla.

2.5.3 Esmini

The name comes from the abbreviation of Environment Simulator Minimalistic[8], used in [5]. It was initially developed to familiarize with the OpenSCENARIO standard. It lacks sensor or physics simulations. However it provides an interface to scenes described in OpenDRIVE format, called RoadManager. It utilizes OpenSceneGraph to visualize the scenarios.

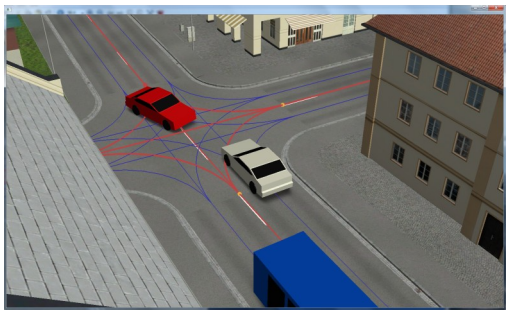


Figure 2.1: Visualization of scenarios in Esmini

[6] https://carla.org/
[7] https://www.svlsimulator.com/about/
[8] https://github.com/esmini/esmini

Chapter 3

Design

3.1 Setup

The first step was to choose a suitable simulator. Esmini was discarded early, because it does not support sensor related simulation and only exploits scenarios in OpenSCENARIO format. Usability is further limited by the simulator, since it does not support all features of the standard. That reduced the options to SVL Simulator and Carla. SVL Simulator did not allow user to import custom maps at the time and has no developer mode in Linux. Realizing developer mode's usefulness, I discarded SVL Simulator, leaving Carla as the winner.

The second step was to install Carla, which could be done in two ways: installing the package version or building from source. With the package version you lose the ability to customize the simulation environment. Although being much more time-consuming, error-prone and less straightforward, the source build was the obvious option here. Note, that I encountered errors during building Carla, that I was not able to solve by using its' documentation alone.

The third step was to create the scene, where the tests from the regulation could be run. Again, the user has more options. Generally, you must have 2 files prepared to be imported into Carla - geometry in filmbox format (.fbx) and road information in OpenDRIVE (.xodr). To obtain them, 2 ways are described in the Carla documentation [3] – using OpenStreetMap or RoadRunner. OpenStreetMap is free, real world map database maintained by diverse community of volunteers. If you need real world-like scenery, OpenStreetMap would be the better choice for you. RoadRunner is recommended though and was therefore, and for reasons mentioned in 2.3, used in this thesis.

Next, setting up the environment is needed Carla provides multiple ways to do it, but they differ depending on whether you have a package version or a source build. Since I use the source build, 3 options were available: typing `make import` in command line, using RoadRunner plugin and manual import through the Unreal Editor. I used to exploit the manual way, back when the one-command procedure had not been implemented yet, but, at the time of writing, it works well. Moreover, it is possible to import multiple maps at once this way.

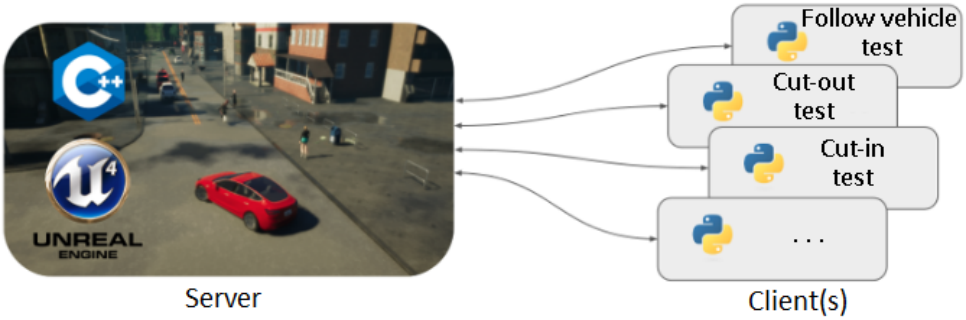


Figure 3.1: Server-client architecture of Carla.

Finally, the user has to decide how to implement the ALKS. To my knowledge, Carla enables this through ROS bridge, CarSim or its Python API. I used the Python API purely because of my familiarity and experience with Python.

3.2 Carla and tests

I built Carla on a computer with Unix based operational system. Carla is based on server-client architecture, see picture 3.1, and uses two ports for communication. I have run both the client and the server locally, so no change was required in the default configuration. Then I exploited Carla's Python API to construct ALKS tests from the regulation. A controller already implemented by Carla developers was used in the test vehicle and my ALKS controlling the ego vehicle, i.e. the vehicle under test. Each test for ALKS is a Python script built up from 3 main parts, that imports 4 common modules, depicted in picture 3.2. Firstly, the setup involves connecting to a server and spawning all actors, i.e. the ego vehicle, its' sensors and the test vehicle. Afterwards, sensor callbacks are assigned as are controllers to the vehicles. Secondly, there is a main loop, which runs until enough time has been simulated. ALKS is called and useful computed data are saved every iteration therein. Each test can be parameterized and was run multiple times in different variations. Finally, saved data are processed and graphs created.

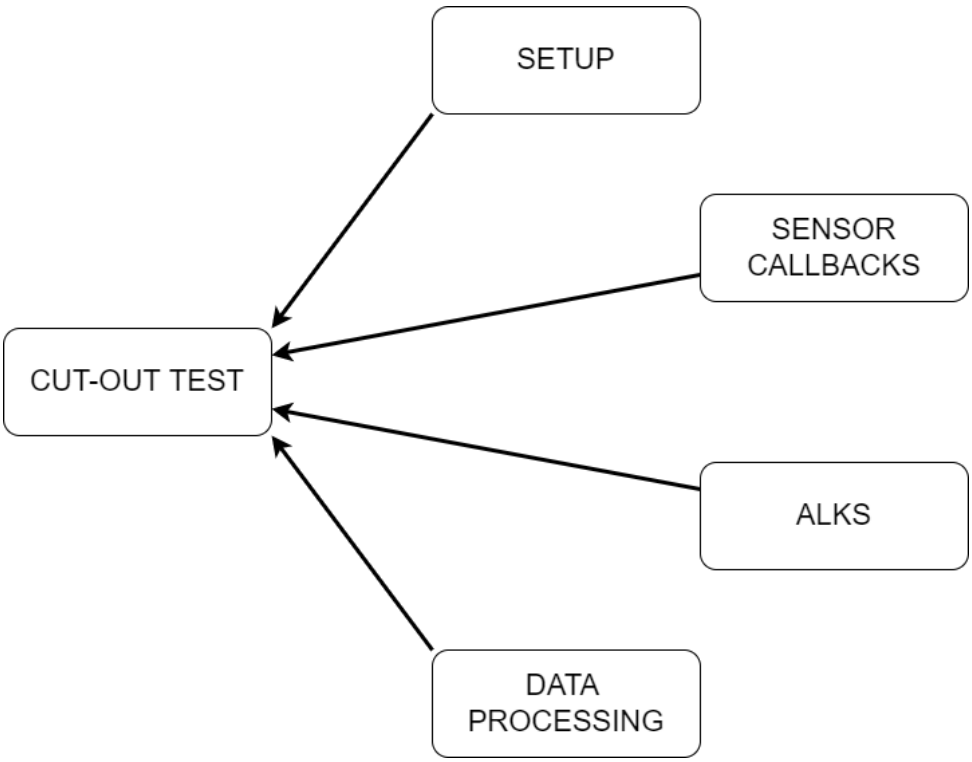


Figure 3.2: Inclusion of common modules into a test (the cut-out test as an example here), all written in Python 3.

Chapter 4

Implementation

In this chapter I describe the installation of necessary software, then the structure of the ALKS tests implemented in Python 3 and finally the way of processing generated data. Throughout the last section issues that occurs are mentioned, but have not been solved yet and require further work.

4.1 Software installation

Carla in version 0.9.12. was used in this work, but newer versions are available at the time of writing. Generally, there are two sections – requirements and Carla build. The prerequisites[1] has to be fulfilled before installation. Then, create a GitHub and Epic Games account and link them together[2]. This is needed, because Carla uses a fork of Unreal engine 4.26, that contains specific patches. Next, clone the repository via `git clone` and build the engine with `make` in its' directory.

The second part is building Carla. Run all following commands in the Carla root directory. First, `git clone` the repository. Downloading assets[3] and exporting the environment variable **UE4_ROOT** is necessary. Finally, run `make PythonAPI` to prepare the client side, `make launch` to run the server and `make libCarla`, which makes it possible to import Carla library into a Python script. Note, that `make launch` is only needed the first time after you either switch to another version of Python or change the source code to compile everything. Afterwards, using `make launch-only` suffice.

For RoadRunner, MathWorks account is required and versions 2021a and 2021b were used. Once you have it, follow the steps from their Help Center[4].

[1] `https://carla.readthedocs.io/en/0.9.12/build_linux/#part-one-prerequisites`

[2] `https://www.unrealengine.com/en-US/ue4-on-github`

[3] `https://carla.readthedocs.io/en/0.9.12/build_linux/#get-assets`

[4] `https://www.mathworks.com/help/roadrunner/ug/install-and-activate-roadrunner.html?s_tid=srchtitle`

4.2 Map creation in RoadRunner

After opening RoadRunner for the first time, you will be prompted to create a new project, to choose a root folder on your drive and name for it. Right after, there will be another prompt to include the asset library. Select "yes". Each project is divided into scenes. After creating the project, an empty scene opens. It takes a few clicks to make a map similar to the one on picture 4.1. Having their documentation[5] at hand, I made such a map ensuring following points:

- In the asset library, freeway is selected in RoadStyles category. This is 4-lane road that suffice to run all ALKS tests. To my knowledge, the road style can not be changed afterwards.
- There are no dead-end roads. When importing the map to into Carla, an error may occur, which makes it impossible for Carla to process OpenDRIVE related information.
- Road bends have the radius of cca. 250 meters, at least on one side. This eases radar points filtration in a way that only simple functions instead of complex algorithms can be deployed.

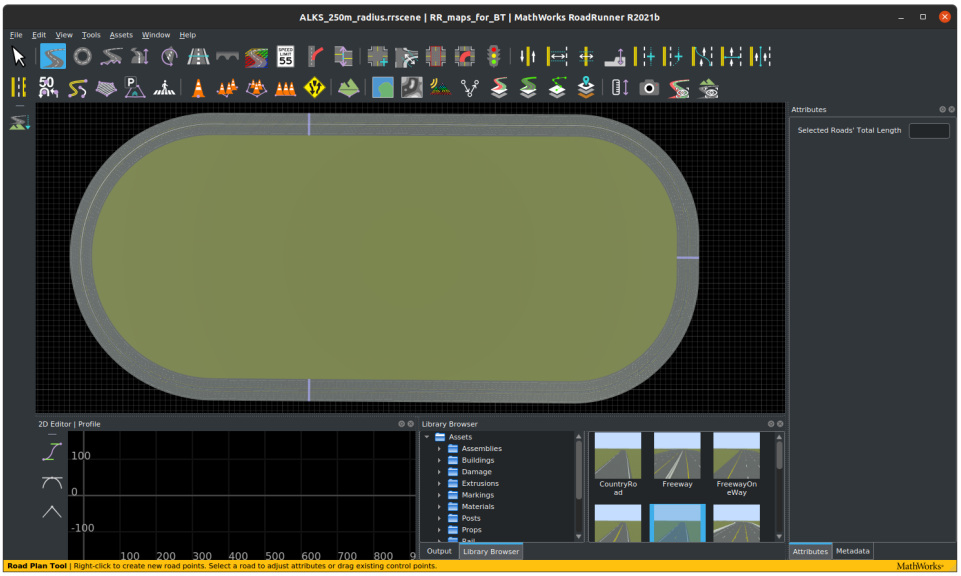


Figure 4.1: RoadRunner GUI with a finished scene, exploited in the Carla simulator to test ALKS.

When finished with the scene, particular files have to be exported. In the section File from the menu above the toolbar, select Export -> Carla. The window that appears has additional options. In the filmbox export subwindow, check only Power Of Two Texture Dimensions and uncheck any other. Export

[5] `https://www.mathworks.com/help/roadrunner/`

to tiles is only useful for huge maps. Under OpenDRIVE options, fulfill following list:

- **Database Version** – Set it to version 1.4. Carla does not support any other.
- **Driving Side** – Right, but if you have a symmetrical map such as in picture 4.1, then left can be chosen as well.
- **Export signals** – Check
- **Export objects** – Check
- **Export scene origin reference** – Check
- **Clamp distances** – Check

Leave options not mentioned in the list unchecked or as they are.

4.3 Import into Carla

Shut down Carla if it is running. Then move or copy all files, obtained after following 4.2, to Import folder located in Carla root folder. Then run `make import AGRS="-package=<name>"` and replace `<name>` with desired name for your map. `<name>` must be different each time you do this import. After the process finishes, launch Carla again. There is a content manager in the bottom part of the Unreal editor. Find the imported map and double click it to load it into the main window. If the import succeeded, then there will be red, dotted lines inside each lane.

Now, depending on whether your scene in RoadRunner contains road separators, denoted as purple lines across a road in picture 4.1, or not, you will have some spawn points generated. Road sections between these lines are recognized as different roads, therefore they ID will differ in the OpenDRIVE file. Check for spawn points can be done in two different ways. Either navigate the map by right clicking and holding the right mouse button and W, A, S, D keys in the main view port until you can see a spawn point. They are denoted as oriented points with the orientation being indicated by a little arrow. Keep the view port close to the road, because spawn points are not rendered if far enough. The second way is to go through the list of all object in the world where your map is. It is located on the upper right side of the Unreal editor. Spawn points contain the string "VehicleSpawnPoint" and some number at the end to differ between them.

In case there are no spawn points, you have to add them manually, because any road user can be spawned only in these points. In the object class database in the upper left side of the Unreal Editor, filter VehicleSpawnPoint and drag-and-drop it into the view port. Place them cca. 2 meters above the road, in the center of a lane and set the orientation accordingly. This prevents many undesired situations such as a vehicle spawned with its' wheels under the terrain, not spawned on a road while its' wheels are or moving in the opposite direction.

4.4 ALKS tests in Python 3

I programmed 3 ALKS tests and 4 common modules in Python 3, exploiting Carla's Python API[6] and Python libraries `numpy` (denoted as `np` in code listings), `matplotlib` (`plt`), class Lock from `threading` and class deque from `collections`. Each contains two vehicles, the ego vehicle with a radar attached and the test vehicle, that perform it. Initially, functionality of the modules, depicted in 3.2, is described. Then the tests themselves are presented.

4.4.1 Setup

Setup prepares the test. It spawns both vehicles and the radar attached to the ego. The radar can be parameterized[7]. How this is done and what numbers I set can be seen in listing 4.1. On the line 1, the radar blueprint is retrieved, then attributes are set and on the line 8, Carla tries to spawn it. In order to simulate a realistic radar, parameters were taken from Bosch car front radar[8]. To spawn the radar, the customized blueprint, the parent actor, i.e. the ego, and transform relative to it are passed to the world's method[9]. Contrary to the documentation, numbers have to be converted to strings, otherwise an error is raised.

Spawning vehicles is in principle the same process, except they are not attached to any other actor, see 4.2. Parameter `ego_spawn` says where it should be spawned (4.3).

```
1 radar_bp = bp_library.find('sensor.other.radar')
2 radar_bp.set_attribute('sensor_tick',str(delta_seconds))#
      delta_seconds==0.02
3 radar_bp.set_attribute('range',str(180))# Info 10 seconds
      ahead at 60 km/h
4 radar_bp.set_attribute('points_per_second',str(246/
    delta_seconds))# (120/3+1)*(30/6+1)==246
5 radar_bp.set_attribute('vertical_fov',str(2 * 15))# +-15
6 radar_bp.set_attribute('horizontal_fov',str(2 * 60))#
    +-60
7 radar_transform = carla.Transform(carla.Location(x=
    ego_vehicle.bounding_box.extent.x, z=ego_vehicle.
    bounding_box.location.z - 0.4))
8 radar = world.try_spawn_actor(radar_bp, radar_transform,
    attach_to=ego_vehicle)
```

Listing 4.1: Radar settings

[6] https://carla.readthedocs.io/en/latest/python_api/

[7] https://carla.readthedocs.io/en/latest/bp_library/#sensor

[8] https://www.bosch-mobility-solutions.com/media/global/products-and-services/passenger-cars-and-light-commercial-vehicles/driver-assistance-systems/multi-camera-system/front-radar-plus/onepager_front-radar_en_200608.pdf

[9] https://carla.readthedocs.io/en/latest/python_api/#carlaworld

```
ego_bp = bp_library.find("vehicle.audi.a2")
sp = world.get_map().get_spawn_points()[ego_spawn]
ego_vehicle = world.try_spawn_actor(ego_bp, sp)
```

Listing 4.2: Vehicle spawning

4.4.2 Sensor callbacks

Radar generates data each sensor tick. Radar callback is a function applied on that data. Filtration of the radar data is shown in listing 4.3. The radar is situated at the front and low above the ground, so it scans points on the road surface. Such points and any other below the radar's height are excluded from ALKS computation, see line 12. Next, all points beyond both adjacent lines are discarded. To compute these, vector cross product and OpenDRIVE information is used, see lines 28, 29. Finally, the callback appends the points into a GlobalData object, see listing 4.4, and sets bool flag `new_r` to `True`. **Note, that the argument map is not Python built-in function. It is another object from the API.**

```
def radar_callback(radar_data, debugger, map, ego_vehicle
    ):
  points = np.frombuffer(radar_data.raw_data, dtype=np.
    dtype('f4'))
  points = np.reshape(points, (len(radar_data), 4))# [[
    velocity, azimuth, altitude, depth],...[,,,]]

  radar_rot = radar_data.transform.rotation
  radar_loc = radar_data.transform.location
  ego_loc = ego_vehicle.get_transform().location
  ego_rot = ego_vehicle.get_transform().rotation
  wp = map.get_waypoint(ego_loc, project_to_road=True)

  # Ignore points below the sensor's height
  points = [p for p in points if p[2] + np.radians(
    ego_rot.pitch) >= 0]

  # Determine where the adjacent lanes end
  wp_loc = wp.transform.location
  wp_loc.z = ego_loc.z
  wp_vec = wp.transform.rotation.get_forward_vector()
  fw_vec = np.array([wp_vec.x, wp_vec.y, 0])
  of_vec = np.array([ego_loc.x - wp_loc.x, ego_loc.y -
    wp_loc.y, 0])# wp_loc.z - ego_loc.z == 0
  result = np.cross(fw_vec, of_vec)
  offset = wp_loc.distance(ego_loc)
  if result[2] < 0:
    offset = -offset
  left_distance = wp.get_left_lane().lane_width + wp.
    lane_width / 2 - offset
  right_distance = wp.get_right_lane().lane_width + wp.
    lane_width / 2 + offset
```

```
26
27    # Ignore points behind adjacent lanes
28    points = [p for p in points if (p[1] < 0 and abs(np.sin
       (p[1]) * p[3]) < left_distance)
29              or (p[1] >= 0 and np.sin(p[1]) * p[3] <
       right_distance)]
30     with MUTEX:
31     GD.rd.append(points)
32     GD.new_r = True
```

Listing 4.3: Radar data filtration

4.4.3 ALKS

The whole ALKS function is shown in listing 4.4. GLobalData class makes up an interface that both the radar and the ALKS use. Moreover, constants of the controllers are stored here (lines 11-17). The steer is calculated as a cross product of the vector from the ego location to the point in the middle of its' lane 3 meters ahead and the ego's forward vector, respectively (lines 24-40). It does not rely on data from the radar, therefore it is computed every time. Throttle and brake computation requires radar data. In case there are none, the default throttle 0.7 is returned (lines 55-58). It has to be in range 0-1 and greater then 0, otherwise the ego would stop as soon as there is nothing detected by the radar. Longitudinal controller's set point is distance that depends on the ego's speed. This is also addressed by the regulation [1], paragraph 5.2.3.3.

```
1  class GlobalData:
2    """Class in which data from sensors are stored and
      accessed by ALKS algorithm."""
3    def __init__(self, lon_p=.15, lon_i=.2, lon_d=.3, lat_p
      =.15, lat_i=.2, lat_d=.3):
4      self.rd = deque(maxlen=10)  # Radar data
5      self.past_controls = deque(maxlen=10)  # My computed
     controls
6      self.new_r = False  # New radar data
7      self.lon_p = lon_p  # Longitudinal proportional
     constant
8      self.lon_i = lon_i  # Longitudinal integral constant
9      self.lon_d = lon_d  # Longitudinal derivative
     constant
10     self.lat_p = lat_p  # Lateral proportional constant
11     self.lat_i = lat_i  # Lateral integral constant
12     self.lat_d = lat_d  # Lateral derivative constant
13
14 # Global variables
15 MUTEX = Lock()
16 GD = GlobalData(lon_p=1.5, lon_i=0, lon_d=0, lat_p=1.,
     lat_i=0, lat_d=0)
17
18 def ALKS(dt, ego_vehicle, road_vec, target_speed=40):
```

```
19  # STEER
20  ego_vec = ego_vehicle.get_transform().rotation.
     get_forward_vector()
21  ego_vec = np.array([ego_vec.x, ego_vec.y, 0])  # We
     only care about xy plane
22  ego_vec /= np.linalg.norm(ego_vec)
23
24  cross = np.cross(road_vec, ego_vec)  # Road vector has
     unit length
25  cross = np.arcsin(cross)
26  angle = np.linalg.norm(cross)
27  if cross[2] > 0:
28    angle = -angle
29
30  # Lateral PID terms
31  lat_ie = sum([pc[2] for pc in GD.past_controls]) * dt
32  lat_de = (GD.past_controls[-1][2] - GD.past_controls
     [-2][2]) / dt
33  lat_pe = angle
34
35  steer = lat_pe * GD.lat_p + lat_ie * GD.lat_i + lat_de
     * GD.lat_d
36
37  if len(GD.past_controls) < 2:
38    GD.past_controls.append([0.7, 0.0, steer])
39
40  if not GD.new_r:
41    last_control = GD.past_controls[-1]
42    last_control[2] = steer
43    return last_control
44
45  # THROTTLE / BRAKE
46  with MUTEX:
47    min_dists = [min(y, key=lambda x: x[3])[3] for y in
     GD.rd if y]  # Array of scalars
48    GD.new_r = False
49
50  # Nothing measured
51  if len(min_dists) <= 1:
52    GD.past_controls.append([0.7, 0.0, steer])
53    return GD.past_controls[-1]
54
55  ego_speed = ego_vehicle.get_velocity()  # ego_speed = [
     x_speed, y_speed, z_speed] [m/s]
56  ego_speed = np.sqrt(ego_speed.x ** 2 + ego_speed.y ** 2
     + ego_speed.z ** 2)
57  time_gap = 1.0 + ego_speed / 27.8  # 0.1 for each 2.78
     m/s
58  distance = max(2.0, time_gap * ego_speed)
59
60  # Longitudinal PID terms
```

```
61    lon_ie = (sum(min_dists) - distance * len(min_dists)) *
        dt
62    lon_de = (min_dists[-1] - min_dists[-2]) / dt
63    lon_pe = min_dists[-1] - distance
64
65    throttle = lon_pe * GD.lon_p + lon_ie * GD.lon_i +
       lon_de * GD.lon_d
66    brake = -(lon_pe * GD.lon_p + lon_ie * GD.lon_i +
       lon_de * GD.lon_d)
67
68    GD.past_controls.append([throttle, brake, steer])
69    return GD.past_controls[-1]
```

Listing 4.4: ALKS implementation

4.4.4 Data processing

Data processing consists of two functions. The first one writes all generated data to memory and the other reads them to draw several graphs, with help of `matplotlib` library. Both vehicles are spawned few meters above the ground and have to fall onto it before moving. This is irrelevant to the tests, so this fraction of the data is not considered when constructing the graphs.

```
1  def write_data(folder, *args):
2    """
3    :param args: name of the file, ego_locations,
       ego_velocities, test_locations, test_velocities,
       ego_controls
4    :return: name of the file
5    """
6    args = args[0]  # Reduce dimensions for better
       indexation
7
8    i = len(args[1])
9    for j in range(2, len(args)-1):
10     assert i == len(args[j])  # Arrays have to be of the
       same length
11
12   with open("records/" + folder + "/" + args[0], 'w') as
       f:
13     for j in range(i):
14       f.write(str(args[1][j][0]) + " " +  # x_ego
15               str(args[1][j][1]) + " " +  # y_ego
16               str(args[2][j]) + " " +  # [m/s]
17               str(args[3][j][0]) + " " +  # x_test
18               str(args[3][j][1]) + " " +  # y_test
19               str(args[4][j]) + " " +  # [m/s]
20               str(args[5][j][0]) + " " +  # throttle
21               str(args[5][j][1]) + " " +  # brake
22               str(args[5][j][2]) + " " +  # steer
23               str(args[6][j]) + " " +  # distance
24               "\n")
```

```
25    return args[0]
```

Listing 4.5: Function that saves data

```
1  def create_graphs(folder, *args, save=True, show=False):
2    """
3    :param args: names of the files where data is saved
4    :param save: bool indicating whether to save figures or
       not
5    :param show: bool indicating whether to show figures on
       the screen or not
6    :return:
7    """
8    import matplotlib.pyplot as plt
9    num_ignore_first = 200
10   for arg in args:
11     with open("records/" + folder + "/" + arg, 'r') as f:
12       data = f.readlines()  # data = [str1, str2, ...]
13       orig_len = len(data)
14       data = data[num_ignore_first:]  # data = [str1000,
     str1001, str1002, ...]
15       data = [d.splitlines() for d in data]  # data = [
     str1000, str1001, ...], only removes '\n' from the end
16       data = [d[0].split() for d in data]  # data = [['a
     ', 'b', 'd', ...], ['c', 'b', 'x', ...], ...]
17       data = [[float(m) for m in n] for n in data]  #
     data = [[a, b, d, ...], [c, b, x, ...], ...]
18
19     def display():
20       plt.show(block=False)
21       plt.pause(5)
22       plt.close()
23
24     plt.figure(num=1)
25     plt.plot(np.arange(num_ignore_first, orig_len), 3.6 *
     np.array([d[5] for d in data]), "r-")
26     plt.xlabel("Cycle [-]")
27     plt.ylabel("Velocity [km/h]")
28     plt.ylim(0, 70)
29     plt.title("Test vehicle's velocity in scenario " +
     arg)
30     if save:
31       plt.savefig("graphs/" + folder + "/" + "TV_" + arg)
```

Listing 4.6: Part of the function that creates graphs from saved data

4.4.5 Tests

Each test accepts 7 arguments – `test_spawn`, `ego_spawn`, `trigger_dist`, `cut_dist`, `ego_speed`, `test_speed`, `change_right` – and begins with creation of empty arrays where data are appended each simulation frame and

additional variables, connection to the server and adjustment of the settings, shown in 4.7 and explained in Carla documentation[10].

```
ego_locations = []
test_locations = []
ego_velocities = []
test_velocities = []
ego_controls = []
distances = []
client = None
world = None
radar = None
settings = None
i = 0
try:
  client = carla.Client('127.0.0.1', 2000)
  client.set_timeout(10)
  world = client.get_world()
  settings = world.get_settings()
  settings.fixed_delta_seconds = 0.02
  settings.synchronous_mode = True
  settings.max_substeps = 10
  world.apply_settings(settings)
```

Listing 4.7: Connection and settings

Then both the test vehicle and ego with the radar are spawned, employing `test_spawn` and `ego_spawn` parameters. Carla controllers are assigned to both vehicles, as in 4.8.

```
from agents.navigation.controller import
    VehiclePIDController
LONGITUDINAL_TERMS = {"K_P": 0.5, "K_I": 0.2, "K_D": 0.3,
     "dt": 0.02}
LATERAL_TERMS = {"K_P": 0.5, "K_I": 0.2, "K_D": 0.3, "dt"
    : 0.02}
ego_CARLA = VehiclePIDController(ego_vehicle,
    LATERAL_TERMS, LONGITUDINAL_TERMS)
test_CARLA = VehiclePIDController(test_vehicle,
    LATERAL_TERMS, LONGITUDINAL_TERMS)
```

Listing 4.8: Controller assignment

Afterwards, the main loop (4.9) executes until 80 seconds has been simulated, which is enough even with the slowest speeds. The `lane_change` helper variable prevents lane change from happening multiple times. The check is based on the vehicles' lane ID from OpenDRIVE file, therefore the safest option to run the test successfully is to ensure that both vehicles stay on the same road segment during the execution of the test. If the lane ID changes when a vehicle enters new road segment depends on how the roads were originally defined. Check on line 22 is valid for the cut-in test only. In the cut-out test, this condition is negated and completely omitted in the follow test, because no lane change happens therein.

[10] `https://carla.readthedocs.io/en/0.9.12/adv_synchrony_timestep/`

Lines 28-34 dictate where the test vehicle moves. Here, `trigger_dist`, `cut_dist`, `change_right` parameters play their role. `change_right` switches between adjacent lanes. When the vehicles reach the distance of `trigger_dist`, the test vehicle starts performing the lane change. Next waypoint to reach for the test vehicle is selected `cut_dist` meters ahead, i.e. how fast will the test vehicle turn its' steering wheel. This logic is the same in the cut-out test and completely omitted in the follow test, since it is irrelevant therein.

```
1  lane_changed = False
2  # Main loop
3  while i * settings.fixed_delta_seconds < 80:
4    # Compute next simulation frame
5    world.tick()
6
7    # Variables needed for each cycle
8    ego_loc = ego_vehicle.get_location()
9    test_loc = test_vehicle.get_location()
10   dist = ego_loc.distance(test_loc)
11   curr_ego_wp = map.get_waypoint(ego_loc, project_to_road
      =True)
12   curr_test_wp = map.get_waypoint(test_loc,
      project_to_road=True)
13
14   # Get all waypoints in distance X meters where the
      vehicles can go, end if there is none
15   next_test_wps = curr_test_wp.next(3) #next_test_wps = [
      wp1, wp2, ...]
16   next_ego_wps = curr_ego_wp.next(3) #next_ego_wps = [wp3
      , wp4, ...]
17
18   # Check if lane has already been changed
19   tv_lid = map.get_waypoint(test_vehicle.get_location(),
      True).lane_id
20   ev_lid = map.get_waypoint(ego_vehicle.get_location(),
      True).lane_id
21
22   if tv_lid == ev_lid:
23     lane_changed = True
24
25   # Choice of where to go next for both vehicles
26   next_ego_wp = random.choice(next_ego_wps)
27
28   if dist < trigger_dist and not lane_changed:
29     if change_right:
30       next_test_wp = random.choice(curr_test_wp.
     get_right_lane().next(cut_dist))
31     else:
32       next_test_wp = random.choice(curr_test_wp.
     get_left_lane().next(cut_dist))
33   else:
34     next_test_wp = random.choice(next_test_wps)
35
```

```
36    # Road vector calculation for steer
37    next_ego_wp_loc = next_ego_wp.transform.location
38    road_vec = np.array([next_ego_wp_loc.x - ego_loc.x,
       next_ego_wp_loc.y - ego_loc.y, 0])
39    road_vec /= np.linalg.norm(road_vec)
40
41    # Control calculated by Carla
42    ego_control = ego_CARLA.run_step(ego_speed, next_ego_wp
       )
43    test_control = test_CARLA.run_step(test_speed,
       next_test_wp)
44    my_control = ALKS(settings.fixed_delta_seconds,
       ego_vehicle, road_vec, ego_speed)
45
46    # Adjust my_control to Carla format
47    ego_control.throttle = np.clip(my_control[0], 0., 1.)
48    ego_control.brake = np.clip(my_control[1], 0., 1.)
49    ego_control.steer = np.clip(my_control[2], -1., 1.)
50
51    # Apply control to the ego vehicle
52    ego_vehicle.apply_control(ego_control)
53    test_vehicle.apply_control(test_control)
54
55    # Save data
56    ego_locations.append([ego_loc.x, ego_loc.y])
57    ego_vel = ego_vehicle.get_velocity()
58    ego_velocities.append(np.linalg.norm([ego_vel.x,
       ego_vel.y, ego_vel.z]))
59    test_locations.append([test_loc.x, test_loc.y])
60    test_vel = test_vehicle.get_velocity()
61    test_velocities.append(np.linalg.norm([test_vel.x,
       test_vel.y, test_vel.z]))
62    ego_controls.append(my_control)
63    distances.append(dist)
64
65    i += 1
```

Listing 4.9: Main loop

The tests was executed while varying their arguments as shown in 4.10. However, this variation was different for each of them. In the cut-in test, the last 2 `for` cycles were merged into one, because the `test_spawn` differs with each value of `change_right` and `ego_spawn` parameters. In the follow test, the last `for` cycle is omitted. `trigger_dist` remained constant. This variation provides 90 possibilities altogether – 36 versions of the cut-out and the cut-in test and 18 versions of the follow test. 5 graphs were created from each possible test.

```
1  if __name__ == "__main__":
2    files = []
3    for i in range(1, 4):  # i = 1, 2, 3
4      for j in range(i):  # j = 0..i
5        for spawns in zip([15, 26, 27], [30, 22, 13]):  #
```

```
    spawns = (15, 30), (26, 22), (27, 13)
6         for right in [True, False]:  # right = True,
    False
7             files.append(write_data("cut-out", cut_out(
    test_spawn=spawns[0], ego_spawn=spawns[1],
    trigger_dist=20, cut_dist=7+3*j, ego_speed=20*i,
    test_speed=10*(j+i), change_right=right)))
8
9   create_graphs("cut-out", *files)
```

Listing 4.10: Test variation

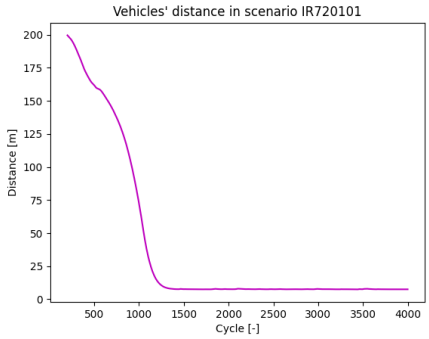
Chapter 5

Evaluation

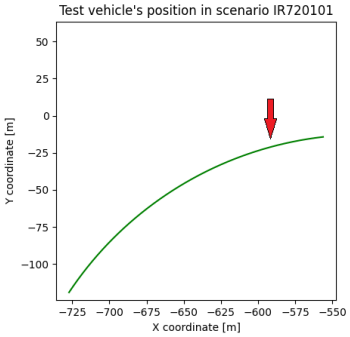
90 tests including all versions have been run to assess behaviour of the ALKS, which consists of 2 P controller, one for each motion domain. Along the X-axis, cycles are labeled. Each test lasted 80 seconds with the time-step of 0.02 second, therefore there are 4000 cycles. First 200 of them are not shown in the graphs, due to reasons in 4.4.4 or 4.6. On the next pages, there are pictures from 1 cut-in test on a right bend at slower speed of both vehicles and from 1 cut-out test on a left bend at medium speed, demonstrating the results. On pictures 5.2c and 5.1c can be seen that the ego vehicle started at certain distance behind the test vehicle, approached it fast and kept cca. 10 meters distance. This indicates no crash of the two vehicles.

In figure 5.1e there is a peak in the ego vehicle's speed around cycle 1000, that exceeds the limit of 60 km/h. In this case the ALKS did not succeed in the test. The reason for it is that the speed is not limited by the algorithm so far. There is no such a peak in 5.2e, but that is due to much shorter initial distance.

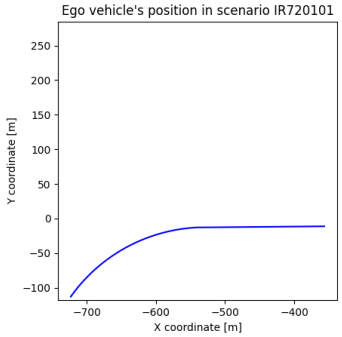
Due to non-realistic method to calculate the steer, the graphs 5.2a, 5.2b, 5.1a and 5.1b indicate perfection in the ego vehicle staying in its' lane. However, in 5.2b, the moment when the lane change happened is recognizable around coordinates **X=-600** and **Y=-500**, indicated by the red arrow. In 5.1d and 5.2d is the speed of the test vehicle driven by Carla controller, whose set point is given speed and shows a only tiny error when performing the lane change.



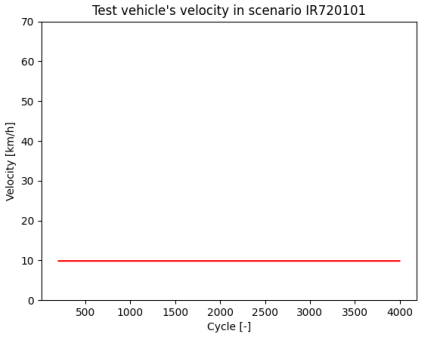
(a) : Distances between the vehicles



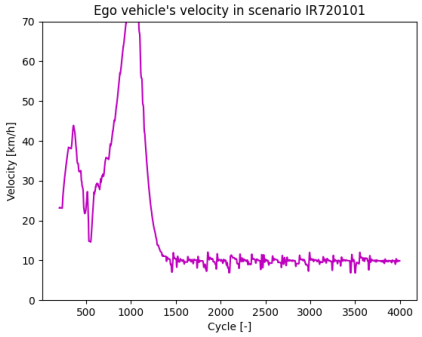
(b) : Test vehicle's position



(c) : Ego vehicle's position

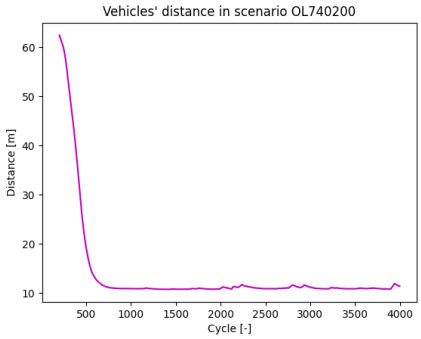


(d) : Test vehicle's speed

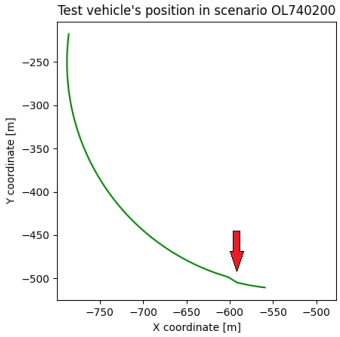


(e) : Ego vehicle's speed

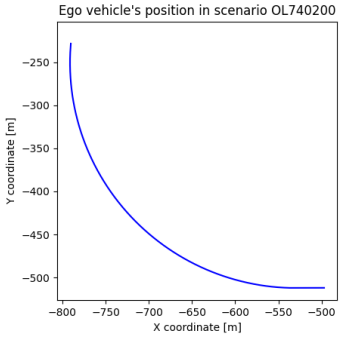
Figure 5.1: Graphs from the cut-in test on a right bend, where the ego was supposed to go by 20 km/h and the test vehicle by 10 km/h cutting into ego's lane from the left. Red arrow shows the beginning of the maneuver.



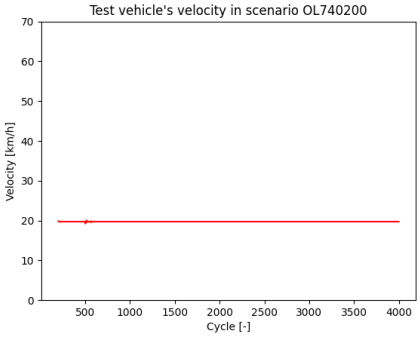
(a) : Distances between the vehicles



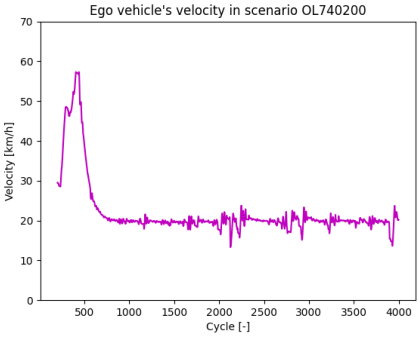
(b) : Test vehicle's position



(c) : Ego vehicle's position



(d) : Test vehicle's speed



(e) : Ego vehicle's speed

Figure 5.2: Graphs from the cut-out test, where the ego was supposed to go by 40 km/h and the test vehicle by 20 km/h and cutting out from ego's lane to the left. Red arrow shows the beginning of the maneuver.

Chapter 6

Conclusion

In this thesis, the reader is presented with 3 open source tools suitable for autonomous driving simulations – Carla, the SVL Simulator and Esmini. Carla was chosen to implement an ALKS according to the regulation UNECE R157, 3 tests (cut-out, cut-in, follow vehicle) from the regulation and a way how to run them in the Python API. Parameterization of the tests allowed for 90 versions altogether.

The ALKS consists of two PID regulators, one controlling the lateral motion, the other the longitudinal one, with their constants set to 1 and 1.5, respectively. The results were shown in 2 sets of graphs. The ALKS passed some tests and failed the other. This and improvements of the ALKS will be the subject of future work.

Bibliography

[1] *UN Regulation No 157 – Uniform provisions concerning the approval of vehicles with regards to Automated Lane Keeping Systems [2021/389]*, 1 2021. Accessed 11.5.2022. URL: http://data.europa.eu/eli/reg/2021/389/oj.

[2] ASAM e.V., 5 2022. Accessed 11.5.2022. URL: https://www.asam.net/.

[3] Antonio M. Lopez German Ros, Vladlen Koltun et al., 5 2022. Accessed 11.5.2022. URL: https://carla.readthedocs.io/en/0.9.12/.

[4] LG Electronics America R&D Lab, 5 2022. Accessed 11.5.2022. URL: https://www.svlsimulator.com/docs/.

[5] Alexander Tenbrock, Alexander König, Thomas Keutgens, and Hendrik Weber. The conscend dataset: Concrete scenarios from the highd dataset according to alks regulation unece r157 in openx. In *2021 IEEE Intelligent Vehicles Symposium Workshops (IV Workshops)*, pages 174–181, 2021. doi:10.1109/IVWorkshops54471.2021.9669219.

[6] The MathWorks, Inc. Roadrunner, 5 2022. Accessed 11.5.2022. URL: https://nl.mathworks.com/products/roadrunner.html.