Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science

Simulation environment for testing of planning algorithms

Diploma thesis

Author:
Bc. Šimon Maňour

Supervisor:
Ing. Jaroslav Klapálek

Master program:
Open Informatics

Specialization:
Software Engineering

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Maňour Šimon**	Personal ID number:	**465818**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Simulation environment for testing of planning algorithms

Master’s thesis title in Czech:

Simulační prostředí pro testování plánovacích algoritmů

Guidelines:

1. Get familiar with Robot Operating System (ROS).
2. Look up videos showing traffic on the intersections. These videos should be appropriate for identification (recognition of vehicles using artificial intelligence). Restrict the selection on a perpendicular view.
3. Using Goodvision AI identify vehicles on the videos, including their trajectories and minimal bounding boxes.
4. Perform an analysis of existing approaches for scene description (traffic situations with an emphasis on intersections). Prefer open-source and widely supported formats.
5. Design and implement an application for scene annotation on the videos. For data representation, use the format selected from the analysis. If it is suitable, use an already existing software, which you extent with missing features.
6. Design and implement a ROS simulation environment that allows rerunning the identified scenario (vehicles and their trajectories). This environment should also support the ignoring of selected vehicles, add custom vehicles with their trajectories, and show additional vehicles controlled by an external application. Check for collisions between individual vehicles and between vehicles and the environment.
7. Perform an analysis of algorithms for path planning (eventually trajectory planning) to navigate through the intersection.
8. Implement the selected algorithm with ROS support and support of your custom simulation environment.
9. Verify the application cascade on several videos using the implemented planner. At least one of the videos should be complex.
10. Evaluate your results and document everything thoroughly.

Bibliography / sources:

[1]K. Tong, Z. Ajanović, and G. Stettinger, ‘Overview of Tools Supporting Planning for Automated Driving’, Sep. 2020, pp. 1–8, doi: 10.1109/ITSC45102.2020.9294512.
[2]M. Althoff, M. Koschi, and S. Manzinger, ‘CommonRoad: Composable benchmarks for motion planning on roads’, in 2017 IEEE Intelligent Vehicles Symposium (IV), Los Angeles, CA, USA, Jun. 2017, pp. 719–726, doi: 10.1109/IVS.2017.7995802.
[3]E. Tsardoulias, K. Iliakopoulou, A. Kargakos, and L. Petrou, ‘A Review of Global Path Planning Methods for Occupancy Grid Maps Regardless of Obstacle Density’, Journal of Intelligent &amp; Robotic Systems, vol. 84, Dec. 2016, doi: 10.1007/s10846-016-0362-z.
[4]S. Pothan, J. L. Nandagopal, and G. Selvaraj, ‘Path planning using state lattice for autonomous vehicle’, in 2017 International Conference on Technological Advancements in Power and Energy (TAP Energy), Dec. 2017, pp. 1–5, doi: 10.1109/TAPENERGY.2017.8397363.

Name and workplace of master’s thesis supervisor:

Ing. Jaroslav Klapálek, Department of Control Engineering, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **10.02.2021** Deadline for master's thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Jaroslav Klapálek Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

ACKNOWLEDGEMENTS

I want to thank Ing. Jaroslav Klapálek for providing advice and guidance regarding the analysis, design, and implementation of the simulation environment.

ABSTRACT

This diploma thesis seeks to create an environment for benchmarking of path planning algorithms for autonomous driving in *ROS*. The aim is to give researchers the means to validate the performance of their algorithms. We analyze different sources of data to determine, whether they are suitable for the benchmarking. Using this knowledge, we create a set of benchmarks containing vehicle trajectories, road descriptions, and goals for the planning algorithm, all extracted from real traffic. We also provide a simulator able to run multiple planning algorithms, evaluate them and visualize their performance. Furthermore, we provide a planning algorithm to demonstrate the benchmarking process and its outputs, and to provide insight on how to work with the benchmark data set.

Key Words: Planning algorithms, Autonomous driving, Benchmark, Simulation, *ROS*

ABSTRAKT

Tato diplomová práce si klade za cíl vytvořit simulační prostředí pro testování plánovacích algoritmů v systému *ROS*, které umožňuje výzkumníkům ověřit své plánovací algoritmy pro autonomní vozidla. Práce se věnuje rozboru různých zdrojů dat a hodnotí, zda jsou data vhodná pro testování algoritmů. Součástí práce je sada scénářů obsahujících anotace silnic, trajektorie vozidel, a cílů pro autonomní vozidlo. Tyto scénáře jsou vytvořeny na základě záznamů skutečné dopravy. Součástí práce je rovněž simulátor, který je schopen spouštět, hodnotit a vykreslit chování plánovacích algoritmů v různých dopravních situacích. K jeho ověření byl implementován vlastní plánovací algoritmus, s jehož pomocí lze doložit nejen funkčnost simulačního programu, ale i přiblížit čtenáři práci s testovací sadou dat a příslušnými knihovnami.

Klíčová Slova: Plánovací algoritmy, Autonomní řízení, Testování, Simulace, *ROS*

Název práce: Simulační prostředí pro testování plánovacích algoritmů

Table of Contents

1 Introduction

Many companies, research centers, and universities acknowledge that automation technology is an important driver for Industry 4.0. Modern autonomous production takes advantage of devices that can complete tasks efficiently, with a focus on safety and collaboration [1]. This automation also includes the possibility of self-driving personal vehicles.

We understand the term ‘Highly autonomous vehicle‘ (HAV) as a vehicle equipped with several sensors, commonly including devices like GPS, cameras, and lidars. This vehicle also needs to be equipped with an embedded computational unit, which is able to respond to the several stimuli produced by the sensors in real-time, reacting by steering the vehicle or adjusting the velocity.

However, for this vehicle to be usable in the real world, it needs to be able to navigate efficiently and, which is the most important, safely in as many scenarios as possible (or at least comparably to human drivers).

This thesis aims to design a set of benchmarks and an environment for the testing of different planning algorithms for autonomous vehicles.

At first, it is necessary to gather data from different traffic scenarios, which the planning algorithm needs to be able to navigate. We mainly focus on roundabouts and intersections, but we also include more straightforward scenes. Moreover, the system is designed to work with data from real traffic (mainly but not limited to trajectories extracted from video footage) rather than fabricated data.

We use AI to extract trajectories from video footage and provide a toolkit to transform the data from a format commonly used by MOT (Multiple object tracking) datasets and detection frameworks to a format suitable for autonomous driving.

Then, we replace one of the vehicles in the original data with a so-called ‘ego-vehicle’ controlled by an external planning algorithm. The format of the data and different ways of pre-processing it is described in Chapter 3.

However, being able to describe the trajectories of the vehicles is not enough for autonomous driving, as we also need information about the scene itself. This includes the location of traffic lanes, surface markings, and obstacles together with their orientation and borders. The aim of Chapter 4 is to explore different formats of scene description and to analyze their benefits and pitfalls.

As we aim to test planning algorithms using the dataset, we need a simulator. This program is able to benchmark algorithms by serving as a substitute for the sensors of the vehicle. The simulator monitors the ego-vehicle's position to check for any errors. The simulator is designed to allow for automated testing of multiple algorithms in multiple situations. In Chapter 5, we describe the overall design, the used technologies, and the implementation of the simulator.

In order to verify the simulator, we have implemented an algorithm, which can communicate with the simulator and plan a trajectory for the autonomous vehicle using the provided data. This algorithm serves as a proof of concept for the parts mentioned above, as well as a practical example of how to communicate with the simulator and how to use the chosen libraries. The design and the implementation of the algorithm are discussed in Chapter 6.

Together, all of the separate parts serve as a toolkit for the verification of autonomous driving algorithms. This includes creating datasets, describing different scenarios, and implementing the algorithms. The performance of the algorithms can then be evaluated and visualized in batch using the simulator.

2 Related work

In this chapter, we explore some of the existing solutions and describe how they are related to this thesis. Mentioned solutions can be related either to the dataset or to the simulation platform itself.

2.1 Robot Operating System

The Robot Operating System (*ROS*) is a flexible framework consisting of tools, libraries, and conventions. The framework aims to simplify the task of creating complex and robust programs for a wide variety of robotic platforms [2].

Given that using *ROS* was a part of the thesis assignment, all of the implemented software was written in the form of *ROS* packages. The thesis includes three packages devoted to working with data, development of autonomous driving algorithms, and benchmarking.

ROS promotes such a modular approach, as it expects a system to consist of several decoupled nodes. To enable communication between the nodes, *ROS* provides a communication protocol using the publish/subscribe pattern [3]. We took advantage of this when designing the interface between the simulator and the autonomous driving algorithm.

2.2 CommonRoad

CommonRoad is a benchmark collection for motion planning of road vehicles [4]. This collection contains a substantial amount of scenarios, consisting of a map in *lanelet* format and positions of vehicles.

The user can then benchmark their algorithms by downloading the scenario as an *XML* file. After processing the scenario, the trajectory of the ego-vehicle controlled by the algorithm has to be serialized to another *XML*. The evaluation is then performed by creating a new submission on the benchmark website [5].

However, this concept of benchmarking expects us to use existing scenarios. We are not allowed to alter the data, create a new scenario, or introduce a custom physical vehicle model.

2.3 NVIDIA DeepStream

Even though we were allowed by *GoodVision* to use their AI [6] to extract trajectories from the selected scenes, our system was designed to be able to work with data gathered from various sources. This could include existing datasets, hand-crafted annotation, or other systems capable of multiple object tracking (*MOT*).

One such system is *NVIDIA DeepStream*. *DeepStream* is a streaming analytic toolkit for building AI-powered applications. It takes video frames as input and uses computer vision to gather information from the footage [7]. This system provides an end-to-end solution for extracting trajectories from a video stream. Using one of the pre-trained models for detection, classification, and tracking, the user can easily extract trajectories from any video footage.

2.4 Autoware

Autoware.AI is the world's first "All-in-One" open-source software for autonomous driving technology [8]. It is built on ROS and contains several modules for localization, object detection, position prediction, and trajectory planning.

This solution comprises tools for processing the data gathered from sensors such as cameras and LIDARs and uses AI to determine the position of the ego-vehicle and the surrounding objects. Coupled with the modules for trajectory planning and position prediction, *Autoware* controls the vehicle by changes in velocity and steering angle.

2.5 Mapping formats

We explored several formats used for mapping. One of such formats is the one used by *OpenStreetMap* (*OSM*) – a free map of the whole world build by volunteers [9] and commonly used for GPS devices [10]. This format describes roads using an imaginary center line, while other attributes, such as the number of lanes or lane size, are added via a key-value system.

The second format we analyzed is *OpenDRIVE* [11], a format developed as a standardized way of describing traffic scenes for driving simulators. *OpenDRIVE* describes the road geometry along with the surface markings and logical properties such as lane types and directions [12].

The last format, *lanelet2*, was introduced in 2018 by Poggenhans et al. [13]. This format was created specifically for autonomous driving. *Lanelet2* describes the road using a network of *lanelets* – small segments of traffic lanes described by their left and right borders. The format's authors also provide a library, which can be used to work with the mapping data.

2.6 Datasets

To create traffic scenarios, we can take advantage of existing datasets containing fully annotated images of road users. *VisDrone* is a dataset consisting of 400 aerial video clips taken in 14 different cities in China [14]. Another example is *AU-AIR* dataset, which focuses on images from low altitude traffic [15]. *InD* dataset consists of footage from four different intersections. The authors use drone cameras to overcome the limitations of established traffic collection methods such as occlusions [16]. With a positional error typically less than 10 cm, the dataset is usable for the development of autonomous driving systems. The useability of the datasets is discussed more in Section 3.1.

3 Benchmark dataset

Within the context of this thesis, we define benchmark as a complete description of a specific traffic scenario, together with a task for the ego-vehicle. In order for the description to be considered complete, we require knowledge about the trajectories of every vehicle and the description of the environment itself. The task of the ego-vehicle is then denoted by the starting position and the destination it has to reach.

We generate the tasks by artificially removing one of the vehicles from the scenario and substituting it with a vehicle controlled by an algorithm. This agent has to reach the same destination as the original vehicle.

3.1 Analysis

To create a complete benchmark, two types of data are required. At first, we require information about the trajectories of all vehicles on the scene. Each road user is described by a unique name, a position in time and space, size, and direction. Using this, we can describe the trajectory of any vehicle that was present on the scene.

The second type of data is about the scene itself. We are looking for a way to describe the location of the surface markings and road edges, which are together creating different traffic lanes. Moreover, we want to annotate the intersections, so the algorithm can deduce which lanes can be reached in order to plan the path correctly.

Using this data, we can select vehicles that can be substituted with the ego-vehicle. Such a benchmark can be used to test the performance of different algorithms for autonomous driving.

3.1.1 Data source

At first, when creating the benchmarks, we had to determine the source for extracting the data. We chose to gather the information from traffic videos that are publicly available on the internet. We were looking for footage that minimizes occlusions and where the objects do not change in size while traveling across the scene. This set of criteria matches drone footage the most, as the camera is located directly above the road.

We also explored the possibility of using existing datasets, as mentioned in Section 2.6. However, most of the data is not suitable for our use case since the view is not top-down, or the camera is moving, which is the case of both *VisDrone* [14] and *AU-Air* [15] datasets. In the case of *inD* [16], the data are not available to the public.

Because of this, we decided to extract the data directly from drone footage, which is publicly available and which fulfills our criteria. The source of the video is always mentioned inside the benchmark configuration file. One such source of free video materials is the website *Pexels* [17], which provides free photographs and videos. Then, we used *GoodVision AI* to extract the trajectories from said footage.

Apart from this, it is also possible to extract the trajectories via human annotators using a specialized toolkit such as *Vatic* [18].

3.2 Trajectories format

Further work with the extracted data requires a standardized and more manageable format. For this reason, we chose *MOTChallenge* [19] format as the baseline for our annotations. The format specifies that every object is on an individual line using comma-separated values. The object's location is described by a bounding rectangle (also called a bounding box), a minimal unoriented rectangle containing the whole object. The names of the fields used in this format are shown below.

```
<frame>, <id>, <bb_left>, <bb_top>, <bb_width>, <bb_height>, <conf>, <x>, <y>,
                                   <z>
```

A brief description of relevant column names is listed below.

- *frame* – number of the frame on which is the object present
- *id* – identifier of the trajectory that the object belongs to
- *bb_ left* – *x*-coordinate of the top-left corner of the object bounding rectangle
- *bb_ top* – *y*-coordinate of the top-left corner of the object bounding rectangle
- *bb_ width* – the width of the bounding rectangle in pixels
- *bb_ height* – the height of the bounding rectangle in pixels

- *conf* – confidence of the detection given by the prediction model
- *x* – *x* location in world coordinates
- *y* – *y* location in world coordinates
- *z* – *z* location in world coordinates

An example of such data for 2D tracking, which is our use case, can be:

```
1, 3, 794.27, 247.59, 71.245, 174.88, -1, -1, -1, -1
1, 6, 1648.1, 119.61, 66.504, 163.24, -1, -1, -1, -1
1, 8, 875.49, 399.98, 95.303, 233.93, -1, -1, -1, -1
```

As we can see in the example, the last four columns are not holding any value and can be neglected. We consider the origin of the coordinate system used by the bounding rectangles to be placed in the top-left corner of the image. The *x*-axis is along the width of the image from left to right, and the *y*-axis describes the height from top to bottom. This is illustrated in Figure 1.

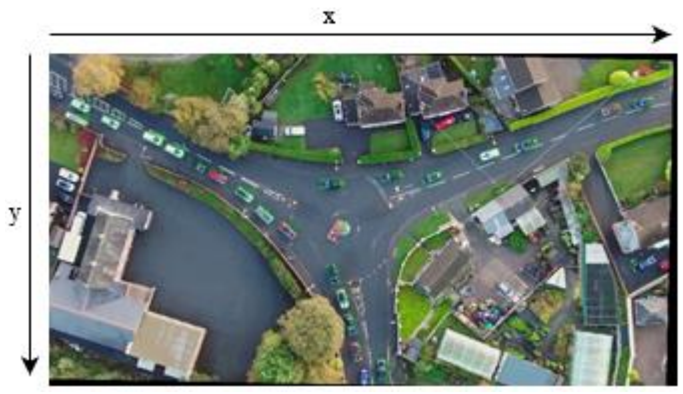


Figure 1 - An image with axes directions

Additionally, we introduce other values, that are necessary for our use case:

```
<class>, <dx>, <dy>, <cx>, <cy>, <w>, <h>
```

The data have the following meaning:

- *class* – type of the object (e.g., ‘car’, ‘van’, ...)
- *dx* – x part of the normalized direction vector
- *dy* – y part of the normalized direction vector
- *cx* – x part of the object center point
- *cy* – y part of the object center point

- w – object width in pixels
- h – object height in pixels

In this setting, we consider width and height in the context of the image coordinate system. This means that width represents the distance from the center of the vehicle to the right edge and height represents the distance to the bottom edge when the direction vector is aligned with the x-axis. This is illustrated in Figure 2.

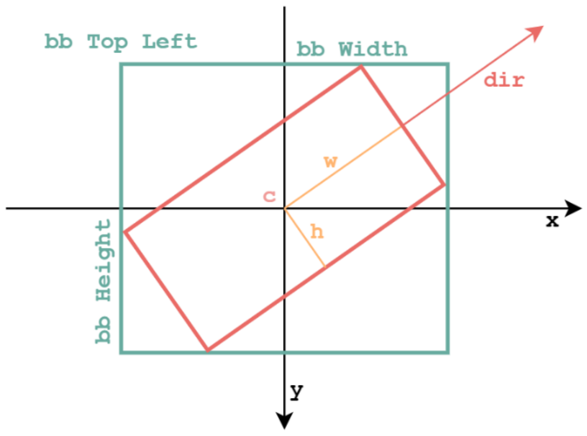


Figure 2 - Describing a rotated rectangle

We use a semicolon instead of a comma as the separator for language localization compatibility reasons. Lastly, we expect the entries to be sorted by their frame numbers in ascending order.

3.3 Data pre-processing package

In contrast to the *MOTChallenge* format and its adaptations, the additional information we require (oriented bounding rectangles) is not common, and none of the aforementioned datasets provided us with this information. Therefore, it was necessary to create a package to process the data as a part of this thesis.

The created package, named '*hav_sim_utitls*', is a standard *ROS* package containing several utility python scripts. Its purpose is to filter trajectories, calculate the orientation of different vehicles, calibrate the map scale, and visualize the data on a video. Using those scripts, we can calculate the missing values, which are needed by the simulator. A summary of script names and their purposes is as follows.

- *insert.py* –adds a custom trajectory to an existing set
- *measure.py* – estimates the calibration when drawing maps (more in Section 4.2)
- *orientate.py* – converts unoriented bounding rectangles to oriented
- *reduce.py* – removes trajectories from an existing set
- *test.py* – contains unit tests for the package
- *utils.py* – contains functions mostly related to I/O and conversions between data structures

We use *Python 3* as the interpreter for this project because, since 2020, *Python 2* is considered deprecated [20].

3.4 Parsing the format

The system needs to be robust, even when working with incomplete data. Therefore all the created tools are designed to work in two different scenarios. The input *CSV* can either contain (i) all the data or (ii) only the trajectory id, frame, and the bounding box. An example of such a situation is processing the raw data for the first time.

A set of functions that handle parsing and writing the data can be found in the Python script *utils.py*. This file is not runnable but contains functions used by other scripts.

3.5 Filtering trajectories

When preparing the dataset, it is necessary to be able to filter out unwanted trajectories, such as trajectories created by false positives or trajectories that are fragmented. Both of these trajectories could cause our system to return erroneous results. An example of a false positive collision is shown in Figure 3. It is possible to perform this filtering by passing a list of trajectory ids and a file containing trajectories to the script *reduce.py*, which produces a new *CSV* without any of the specified trajectories.

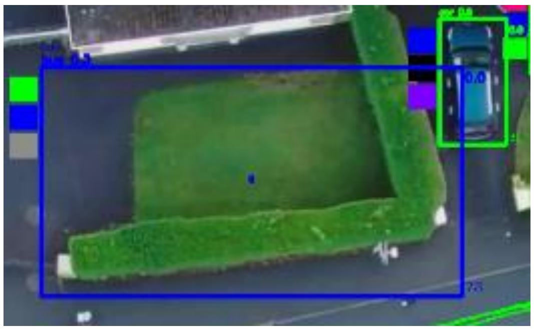


Figure 3 - False positive intersecting with a real object

3.6 Calculating rotated rectangles

Estimation of the other values such as the orientation of the vehicle, its size, and its center is done by the script *orientate.py*.

It is necessary to account for a noise in the position of the bounding rectangle center point. The noise can either be caused by the prediction model or by human error in the case of hand-crafted data. Therefore, we decided to filter out buses and trucks, as it was difficult to calculate their orientation reliably. It was also necessary to filter out any pedestrians, as they were detected only scarcely due to the high altitude of the recording drone.

Afterward, the program proceeds with estimating the orientation of the bounding rectangles. First of all, the program calculates the center point of the object. When there is enough distance between two center points, the orientation is obtained as the difference between these two points. This is repeated for each point of the whole trajectory, while this distance in pixels is set as a hyperparameter, and the optimal value may differ for different scenes. If there is not enough movement (e.g., the object is stationary), the trajectory is removed altogether, as the orientation cannot be calculated.

Estimating the vehicle orientation splits the trajectories into several segments with the same orientation. The values are then interpolated to make the change in the direction more smooth and natural.

As the next step, the script estimates the width and height of the object using the ratio of typical dimensions of road vehicles. The length of 4-5 m and width of

1.5-2 m [21] gives an estimated ratio of 0.3-0.5. We use the middle value of 0.4. Also, the size can be multiplied by an arbitrary constant to compensate for artificially inflated or deflated bounding boxes.

When the orientation and size of all the objects are calculated, the script saves the resulting values in a new *CSV* file. While Figure 4 shows the data before this transformation, Figure 5 contains orientated rectangle with adjusted sizes.

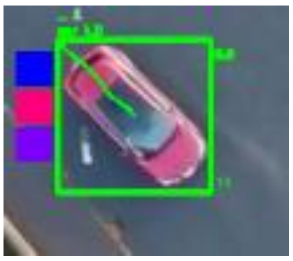


Figure 4 - Non orientated bounding box

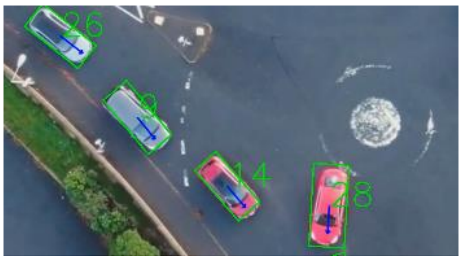


Figure 5 - Bounding boxes with orientation

3.7 Visualization

The script *plotter.py* provides a convenient way of visualizing the processed data. By specifying a path to a *CSV* with trajectories and a path to the corresponding source video, the script plots the bounding rectangles into this video. By default, the system plots orientated rectangles. In case that the required data are missing, it falls back to default bounding boxes.

Using this script, we have a simple way of performing a sanity check of our data. The script can also be used to manually verify the results of operations like *reduce* (Section 3.5) or *orientate* (Section 3.6).

3.8 Adding custom trajectory

As a part of the assignment, we need to be able to insert an artificial trajectory into the existing dataset. Later on, it was specified that the artificial trajectory should be formatted as a *CSV* file containing a serialized ROS message of the type *Path* [22]. This operation is handled by the script *insert.py*.

This script parses the serialized message and then transforms the poses to fit the target scene. This means that both the center point of the vehicle and its size are adjusted, and the direction vector is calculated from the provided quaternion. Also, the timestamps are converted from real-time (seconds) to logical time (frames). For any frame that is not associated with any pose, the values are estimated using linear interpolation.

When the calculations are finished, the artificial trajectory is added to the existing data and saved to the disk. The positions of this trajectory have the fields `<bb_left>`, `<bb_top>`, `<bb_width>`, `<bb_height>` set to -1. This way, other components, such as *orientate* (Section 3.6), can identify them as artificial and not modify their data. On the other hand, the *simulator* or the *plotter* treats them as legitimate trajectories and works with them as if they were in the original set. An example is shown in Figure 6.

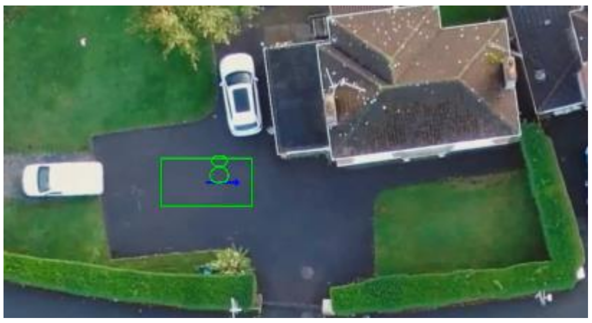


Figure 6 - An object artificially inserted into the scene

3.9 Testing

The scripts used to pre-process data are covered by unit tests from the *Python unittest* [23] framework. The script *test.py* contains several classes used to test the different operations with the data. Also, the *resources* directory contains input files for the tests to work with. Therefore, the tests must be run with the folder *scripts* as the working directory or have access to *resources* via a link.

The tests, when ran, perform operations on sample data. These tests consist of reading and writing the trajectories, trajectory filtering, and inserting artificial trajectories into a dataset. The output is then compared to the expected result.

For *orientate.py*, we only test the functionality but not the quality of the results, as we do not know the ground truth for the results, which vary depending on the hyperparameters (such as the *pixel limit* or the *width and height ratio*). Because of this, the main aim of the test is to verify that the script does not modify any artificial trajectory. To test the quality of the results, we plot the objects onto a video. By watching the video, we can manually inspect the boxes and their rotation, and determine, how close the estimated bounding rectangle fits the object in the video.

4 Scene annotations

Apart from having the serialized trajectories, it is also necessary to describe the scene itself. An autonomous vehicle needs to be aware of its surroundings to navigate. Therefore, we need to devise a way of annotating road borders, lane borders, and intersections to be used with the trajectories as an input for the algorithm.

It is required to find a standardized way of describing the scene with all the necessary information for the planning algorithm. Moreover, as the scenes are annotated manually, the availability of a comprehensible and user-friendly editor is paramount.

4.1 Analysis

We described three different mapping formats in Section 2.5. The main reasons behind not using *OSM* are (i) its counter-intuitive way of describing the scene and (ii) it hinders our ability to create precise scene annotations. Moreover, creating a comprehensible description of an intersection using this format has proven to be very hard since there is no single center line.

The major shortcoming of *OpenDRIVE* was that even after a thorough search, we did not manage to find any free editing software that was also functional.

Issues similar to that we encountered are also described by Poggenhans et al. [13].

4.1.1 Lanelet2 format

The framework *Lanelet2* [24] provides a solution for the stated problems. It is built with autonomous driving in mind and therefore provides enough primitives for the developers to create custom maps. Furthermore, the format is essentially an extension of the *OSM* format, and as such, there is an open-source editor called *JOSM* [25]. *Lanelet2* provides a set of extensions for this editor to enable highlighting and labeling of the *lanelet* primitives. A tutorial on how to enable the extension can be found on the framework's *GitHub* [26]. Apart from that, we are also provided with a *C++* library in the form of a *ROS* package to work with *lanelet2* maps.

The mentioned format introduces a type of object called *lanelet*. A lanelet consists of exactly two *linestrings* – ordered sequences of points and the line segments connecting them. One of the *linestrings* denotes the left edge of the *lanelet*; the other denotes the right. Together they form a driveable area called *lanelet*. The orientation of the *linestrings* does not matter since the framework uses the *left* and *right* annotations to determine the direction of the traffic lane.

Apart from entities that are physically present on the scene (e.g., curbstones, road surface markings), the linestring may be purely virtual. A virtual lanelet border does not physically exist but serves to connect lanelets on different sides of an intersection.

More on the different types of primitives specified by *lanelet2* can be found in the documentation [24].

4.2 Calibrations with OSM

When creating a map for the dataset, we use the *JOSM* editor with *lanelet2* extensions. To create a map, we need a canvas that we can draw onto. The addon *PicLayer* [27] lets us import a screenshot into the editor and use it as a reference for the map.

Unfortunately, the screenshot has most likely a wrong scale, and depending on the resolution, the objects can be severely larger or smaller than in reality. Therefore, we have to create a calibration file for the image to make the scale more accurate. This file contains the following field:

```
INITIAL_SCALE=<...>
```

The value on the right-hand side represents the equivalence of 100 pixels in meters (e.g., a value of 2.5 would mean that every 100 pixels in the image are equal to 2.5 meters in reality).

We can either guess this value ourselves, but since this process can be tedious, we use the script *measure.py* (mentioned in Section 3.3). Passing a path to the screenshot and the estimated size of one of the vehicles on the scene opens a window displaying the screenshot. The user should then click on the front and rear sides of the vehicle. Afterward, the program computes the value of calibration for the picture. After loading this new calibration value, we may validate the sizes of objects in the scene (e.g., using the ruler tool) to ensure that the distances are credible.

4.3 Annotations using Lanelet2

When the image is loaded and its size is calibrated, we can start drawing the maps. This process consists of drawing lines to match the shapes of curbstones, road surface markings, or other objects present on the scene. Each of the lines should be marked with its corresponding type. To create a *lanelet*, we create a relation between exactly two lines – one assigned as *left*, the other as *right*.

4.3.1 Using virtual lines

When creating the map, we should keep in mind that for a connected path between two points to exist, there must be an uninterrupted sequence of *lanelets* in between. For this reason, we should create virtual lines. This is especially useful when describing an intersection, as shown in Figure 7. This way, we can describe different paths that the vehicle can take depending on its destination.

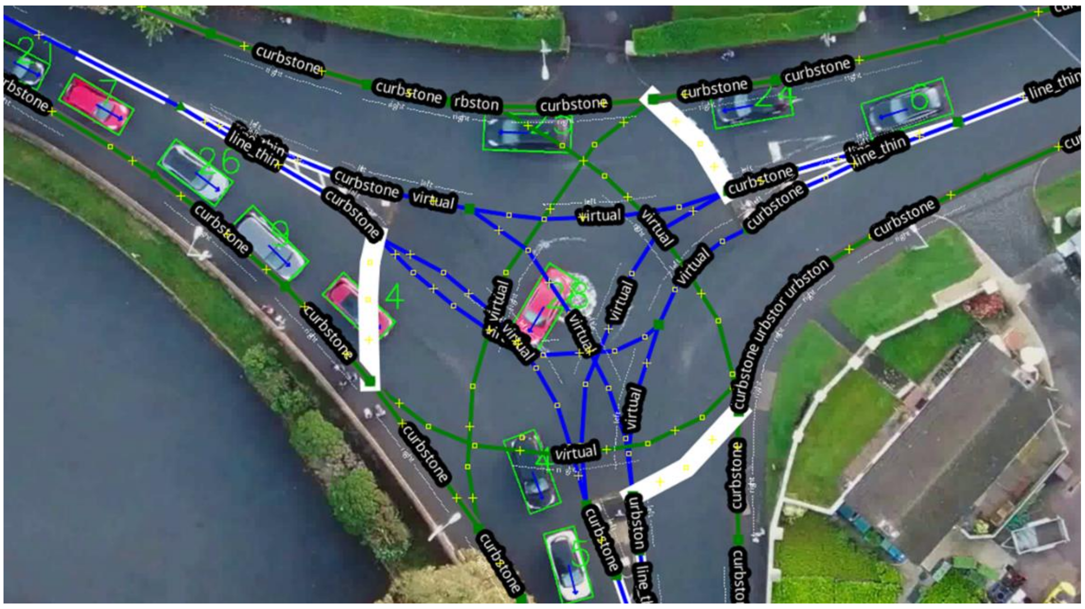


Figure 7 - Intersection described using lanelets

4.3.2 Connecting lanelets

One inconvenience can arise from the ‘atomicity‘ of the *lanelets*. If we want to connect two *lanelets* (or split one in two), we should do that only at the end of the *lanelet*. If we do not connect the *lanelets* in this way, the library does not treat such *lanelets* as connected and cannot find any path between them. An example of the wrong way of drawing *lanelets* is illustrated in Figure 8. On the left, we see a *lanelet* connected to the middle of another *lanelet* (highlighted in red). On the right side, we should notice that the resulting routing graph is disconnected.

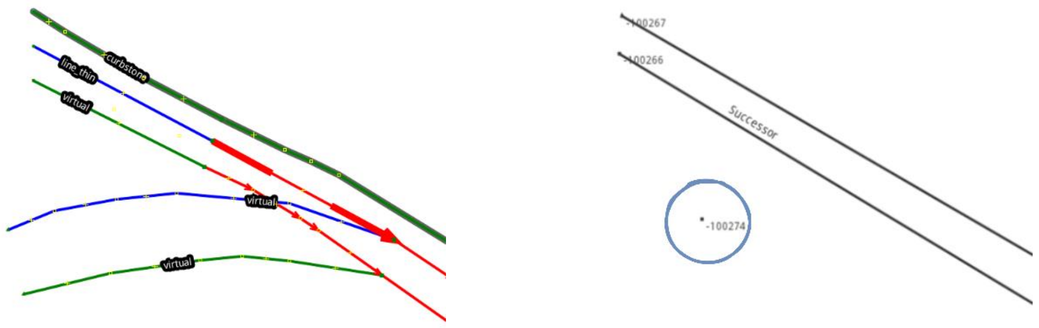


Figure 8 - Incorrectly connected lanelet and the resulting routing graph with the circled node disconnected

A correct approach is illustrated in Figure 9, where the *lanelet* is connected to the end of the highlighted one. As we can see on the right, the routing graph is connected, and a *Successor* relation is formed.

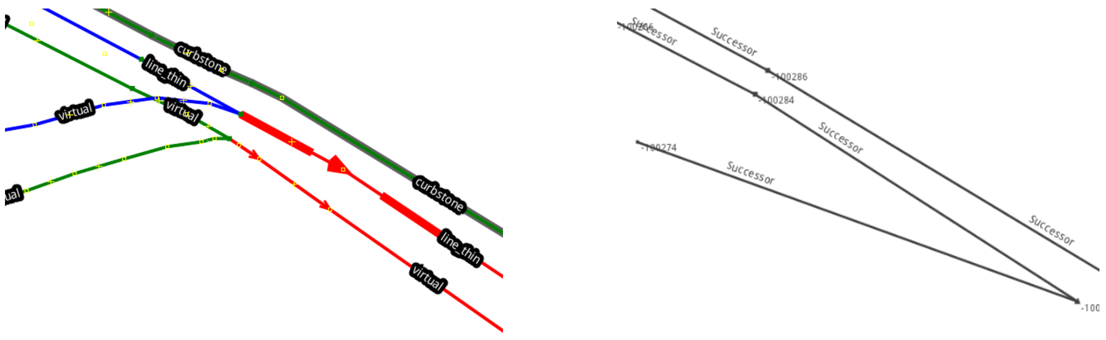


Figure 9 - Correct way of connecting lanelets and the resulting routing graph

4.4 Custom tags

When we are finished with the mapping, we need to add two more artificial nodes to our map. These nodes are required since our trajectories are using pixels as their coordinates, ranging from *(0,0)* to *(width, height)*, while *lanelet2* uses metric coordinates. Placing one node to the top-left corner and the second one to the bottom-right enables us to project the trajectories on the created map.

To distinguish the two artificial nodes, we introduce a new keyword *custom_ coords*. The top-left node has the value set to *tl*, while the other is defined by the value of *br*.

4.5 Validation

The *lanelet2 ROS* library contains a tool to validate the created map. This tool can be run in the following way:

```
$ rosrun lanelet2_validation lanelet2_validate <path_to_map>
```

By running this program, we can make sure that we did not violate the format. If the program prints an error message or does not finish successfully (e.g., due to a segmentation fault), there are most likely some issues with our map. The different errors we encountered and their solutions are described in greater detail in Section 4.6.

The validator can also output some warnings. Those are not necessarily harmful, but resolving them can improve the accuracy of programs working with the map. In contrast, some warnings are safe to ignore (e.g., our artificial top-left and bottom-right nodes also produce a warning, as the library does not recognize them as a part of the *lanelet2* format).

4.6 Fixing the errors

In this section, we describe all of the errors that were encountered during the development and explain a way of solving them. When using *JOSM* in the default configuration, we were not able to search for an object with a negative ID. *JOSM* assigns a negative ID to any object, which was not uploaded to *OSM* servers. Given that we are drawing custom maps, all the annotations we create have a negative id. This has proven to be problematic since we need to be able to locate the erroneous objects on the map to fix them. A solution for this is to switch *JOSM* to 'Expert Mode' and then use the *MapCSS* search syntax. For example, to look for an object with id -11668 we could use the following *MapCSS* selector:

```
*[osm_id()=-111668]
```

Let us note that the described issues were encountered using version *1.0.1-1bionic.20201017*. Other versions may behave differently.

4.6.1 Segmentation fault

Segmentation of the validator occurs when the process is stopped due to attempted access to unallocated memory. This behavior can be caused by a *lanelet* relation with less than two members. Such relation may be caused by a user deleting a linestring that is a part of a *lanelet*. Even though both linestrings of a *lanelet* are deleted, a leftover relation is still present and must be manually cleaned, as illustrated in Figure 10.

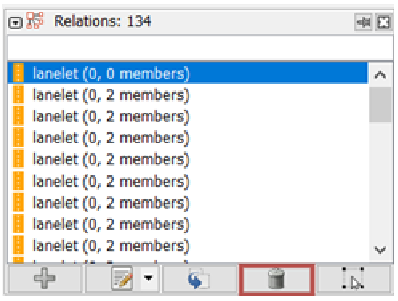

Relations: 134
lanelet (0, 0 members)
lanelet (0, 2 members)
lanelet (0, 2 members)
lanelet (0, 2 members)
lanelet (0, 2 members)
lanelet (0, 2 members)
lanelet (0, 2 members)
lanelet (0, 2 members)
lanelet (0, 2 members)

Figure 10 – Lanelets with 0 members has to be deleted manually

4.6.2 Not the closest lanelet

Some of the *lanelets* might be reported as erroneous in the following way:

```
Error: There is a 'left' relation from -100318 to -100319, but -100318 isn't
        the closest lanelet the other way round [routing.graph_is_valid]
```

This error is introduced whenever one linestring is a left (or right) border for two different *lanelets*. This is shown in Figure 11, where the upper linestring is in relation to the bottom two as their left border. The solution to this is to introduce the fourth *lanelet* and use it as a left border for one of the bottom linestrings.

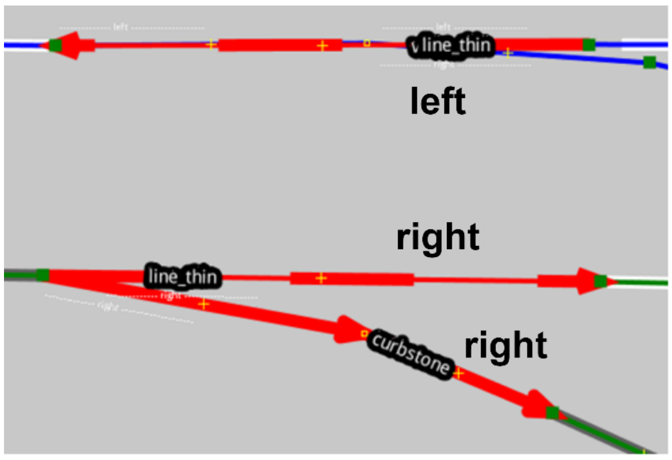


Figure 11 - A 'left' lanelet in relation to two 'right' lanelets is considered an error.

5 The simulator

Apart from assembling a dataset of traffic scenarios, we were required to design and implement a simulator to work with the contained data to evaluate an external autonomous driving algorithm.

This chapter is dedicated to the requirements of the simulator package, its architectural design, and the implementation of its components.

5.1 Requirements

According to the assignment, a simulator has to be a program that can load benchmark data and run an externally provided autonomous driving algorithm. The simulator passes the data describing the scene and the trajectories to the algorithm while gathering its responses.

The simulator should be able to detect any errors made by the algorithm, such as collisions with other objects or steering away from the road. Furthermore, the simulator should support a means of visualizing the scene and the performance of an autonomous vehicle.

As the simulator serves to test different planning algorithms, there should be a possibility for batch evaluation of various algorithms on multiple scenes. Besides, the simulator and the algorithm should be coupled as loose as possible, as we do not want to constrain the developers of the algorithm to a single toolkit or programming language.

5.2 Architectural design

One of the first designs was based on implementing the algorithms as shared libraries to be imported by the simulator. Although this would provide an efficient way with a low amount of overhead, it would also mean the simulator and the algorithm would be parts of the same process. Such an approach would introduce many constraints on the algorithm, e.g., limiting developers to use only the `C/C++` programming language.

The chosen approach is to keep both the simulator and the algorithm as separate executables while using the *ROS* messaging interface for communication [28]. Using this interface, both of these nodes serve as a publisher and a subscriber. This way, both components do not need any knowledge about the other, as the only connection between them is the communication bus. This is illustrated by Figure 12, where the simulator and the autonomous driving algorithm are depicted as two separate nodes. Both advertise and subscribe to different topics using the *ROS* master, a process used for communicating between them.

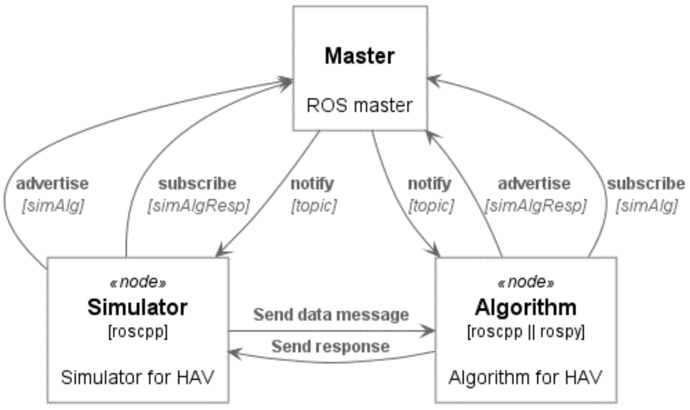


Figure 12 - Communication between ROS nodes

Thus, the only constraint that we impose on the algorithm other than using specific messages to communicate is to be able to understand the *lanelet2* format. Apart from this, there is complete freedom in the selection of programming language or used libraries.

5.3 The communication protocol

Inter-node communication uses six different types of messages: *Init*, *Frame*, *Fin*, and a response for each of the three types. The simulator always initiates the communication while the algorithm responds to the messages. Communication consists of three phases, each using different types of messages. The messages are described in greater detail in Section 5.4.

In the beginning, the simulator sends an *Init* message to the algorithm to initiate the communication. After receiving this message, the algorithm sends a response to signal that the algorithm is ready.

After the initialization phase, the simulator starts to send *Frame* messages, describing the position of all the objects on the scene in the given frame. This means that the position of the vehicle substituted by the ego-vehicle (as discussed in Chapter 3) is also being sent. We do not want to force the algorithm to respond right from the start, as we realize that some algorithms need several frames to calibrate. Thus, the algorithm can reply with the position of the original vehicle. This also reflects the reality better, as the algorithm should have some knowledge about the vehicle's surroundings prior to entering the intersection.

After there are no more frames to send, the simulator waits for all the pending *Frame* responses and then sends a *Fin* message. After receiving a response, the communication stops. The algorithm may also request the simulation to end in the *Frame* response messages if, for example, the ego-vehicle has reached its destination.

The communication between the simulator and an algorithm is illustrated in Figure 13.

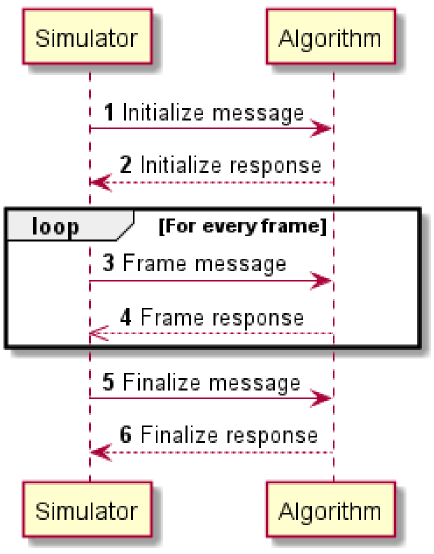


Figure 13 - Communication between the simulator and the algorithm

5.4 Messages

In this section, we will go over the different messages that are being used for the communication of the simulator and the autonomous driving algorithms. All the definitions can be found on the attached DVD as described in Appendix.

As stated in Section 5.3, the simulator and the algorithm communicate using six different message types.

5.4.1 Point2

Point2 represents a point or a vector in 2-D space.

- *float x* – *x*-part of the point or vector
- *float y* – *y*-part of the point or vector

5.4.2 Position

The *Position* is used to describe the shape of one object in the scene. The message uses *Point2* as the type for some of its fields.

- *Point2 center* – location of the center of the object
- *Point2 size* – *x* (width) and *y* (height) size (described in Section 3.2) of the object
- *Point2 orientation* – the direction vector of the object
- *int32 id* – identifier of the trajectory that the position is part of

5.4.3 Init

The simulator sends the *Init* message to initiate the communication with the algorithm. It provides data, which might be useful to the algorithm before the simulation starts. The map is passed as a path to the file because the *lanelet2 ROS* package does not, to this date, provide a way of serializing its map as a message.

- *string osm_path* – path to the map in *lanelet2* format
- *string name* – the name of the benchmark
- *int32 id* – identifier of the vehicle that is replaced by the algorithm
- *float fps* – fps of the source video, used to convert from frames to seconds
- *float maxSpeed* – the maximum traveled distance between frames
- *float initialSpeed* – speed of the vehicle when entering the scene
- *Point2 size* – *x* (width) and *y* (height) size (described in Section 3.2) of the object
- *Point2 start* – location of the vehicle when entering the scene
- *Point2 end* – a destination that needs to be reached by the algorithm

Multiplying the values of fields *maxSpeed* or *initialSpeed* by the *fps* yields the speed in meters per second.

The algorithm responds with the *InitResponse* message, which currently has no fields and is used only to notify the simulator that the algorithm is ready.

5.4.4 Frame

The simulator sends a *Frame* message for each frame in the dataset in a sequence with ascending frame id. This message describes the position of every object in the scene in a given time, using the *Position* message.

- *int32 frameId* – identifier of the frame
- *Position[ ] positions* – an array of position messages for each object in the scene

The algorithm responds with *FrameResponse* message:

- *int32 frameId* – identifier of the frame, matches the one in *Frame* message
- *Position position* – position of the simulated vehicle on the frame
- *string status* – the status of the algorithm

Currently, the status can have three different values. `WAITING` means that the algorithm is still waiting for the simulated vehicle to appear in the scene. `RUNNING` means that the algorithm is running, and `FIN` means that the algorithm has completed its objective (the vehicle has reached its destination). For any other status than `RUNNING`, the position may have an arbitrary value, as it is not taken into account.

5.4.5 Fin

Both *Fin* and *FinResponse* messages are empty and are only used for synchronization between the simulator and the algorithm.

5.5 Error types

During the simulation, the simulator detects several errors. Their names and descriptions are listed below:

- *Object collision* - arises when the ego-vehicle collides with another vehicle

- *Line collision* - introduced when the ego-vehicle collides with a *linestring* object in the map
- *Timeout* - used when simulating a real-time situation (described in Subsection 5.7.1) and is detected whenever the time before a message is processed exceeds the specified limit
- *Not in destination* - signals that the ego-vehicle did not arrive at the target destination
- *No error* – used as a placeholder when there is no error detected

5.6 High-level design

In this section, we describe the flow of the simulator process from reading the input to serializing the results. We also discuss the libraries we use in Subsection 5.6.1.

The simulator takes a path to a *JSON* file as an input argument. This file describes the whole test scenario and specifies the data on which the algorithms will be benchmarked. Afterward, the program loads the trajectories and map specified in the *JSON* file and validates that they are in the correct format (explained in Section 5.7).

When the data are loaded and validated, the program starts an algorithm on a new thread and communicates with it (as described in Section 5.3) while checking for any errors produced by the algorithm. The simulator also provides the frames for any type of visualization method specified in the input configuration. This is repeated for every algorithm, scene, and ego-vehicle id.

When the simulation completes, the results are written to a *CSV* file and the program exits successfully. However, in case that an exception is encountered, the program writes the exception message into the log and then exits with a non-zero code.

The diagram in Figure 14 illustrates the flow of the simulator process from its start to the end.

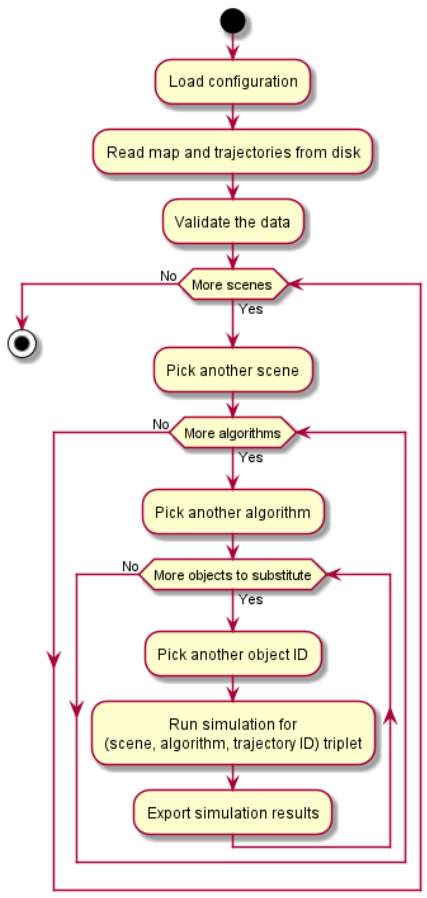

Load configuration
Read map and trajectories from disk
Validate the data
No
More scenes
Yes
Pick another scene
No
More algorithms
Yes
Pick another algorithm
No
More objects to substitute
Yes
Pick another object ID
Run simulation for (scene, algorithm, trajectory ID) triplet
Export simulation results

Figure 14 - The simulator process from start to end

5.6.1 Used libraries

We use the *Eigen 3* library to represent different algebraic structures. This library provides an elegant and templated implementation of different algebraic objects and operations involving them [29]. For image processing, we use *OpenCV* [30].

To work with map data in *lanelet2* format, we use the library provided as a *ROS* package by the format's authors [24]. This library comes together with *Boost* [31], which we use for threading, and structures not supported in C++11.

The message passing framework is provided directly by *ROS*, namely the library *roscpp* [32]. Using this library, we create publishers, subscribers, and messages used for communication with the algorithm or for a visualization environment (e.g., *rViz* [33]).

We use the open-source, header-only library *JSON for Modern C++* by N. Lohmann to handle [34] *JSON* format.

5.7 Input Configuration

To parse the input arguments, we use a parser from the *OpenCV* library so that we do not introduce new dependencies. The path to the configuration *JSON* file is the only argument, which needs to be provided to the program. This way is more practical, as the *JSON* contains several nested objects, and passing all the information as input parameters might not be as comprehensible.

The simulator loads this file and parses it. If the loading fails for any reason (e.g., syntax error, an erroneous path to the file), an appropriate exception is thrown, and the application terminates. After loading the *JSON* object to memory, the program validates this object by checking for all the required input fields. If any field is missing or has a wrong type, an exception is thrown.

An example *JSON* is shown in Figure 15. More examples are provided on the enclosed DVD as described in Appendix.

```
{
  "opencv": "OFF",
  "rviz": "ON",
  "timeLimit": -1,
  "out": "results",
  "algorithms": [{
      "name": "SimpleAlgorithm",
      "cmd": "rosrun sim_alg sim_alg FOLLOW"
    }],
  "benchmarks": [{
      "data": {
        "name": "ThreeWay",
        "source": "https://www.pexels.com/some_video",
        "osm": "/simulator_data/lanelets/threeWay.osm",
        "csv": "/simulator_data/csvs/threeWay.csv",
        "width": 4096,
        "height": 2160,
        "fps": 40
      },
      "simulationIds": [1,6,23]
    }]
}
```

Figure 15 - An example of an input JSON

5.7.1 Top-level data format

The fields of the top-level *JSON* object are described below.

- *opencv* – If set to ‘‘ON’’, the program plots the scene into a video file.

- *rviz* – Setting to ``ON'' enables sending the objects on the scene as *markers* to *rViz*.
- *timeLimit* – Setting *timeLimit* to a positive value causes the program to run in a 'real-time mode'. This means that the rate of sending frames to the algorithm is limited by the fps of the data source. If the algorithm does not send a response in the required time, it is treated as an error. It is advised to use this mode without any visualization as it may hinder the speed of the simulator. If timeLimit is negative, the simulator sends the frames as fast as possible and does not check the response time.
- *results* – The results of the benchmarks are written to files named according to this field. For example, setting the value ``results'' creates two files named `results.csv` and `results.json`.
- *benchmarks* – An array of objects holding the information about the benchmarks on which to test the algorithms. This array is described in greater detail in Subsection 5.7.2.
- *algorithms* – An array of objects describing which algorithms to benchmark. More information is provided in Subsection 5.7.3.

5.7.2 Benchmark object

The field *benchmarks* is, as stated above, an array of objects. A brief description of the contained fields follows.

- *name* – Name of the scene is used by the program to name the videos and images produced during the benchmarking.
- *source* – This field contains a path to the source video from which the trajectories were extracted.
- *osm* – The field *osm* describes the path to a map in *lanelet2* format.
- *csv* – A string containing the path to trajectories file.
- *width* – The width of the source video in pixels.
- *height* – The height of the source video in pixels.
- *fps* – frame rate of the source video
- *simulationIds* – A list of trajectory identifiers describing which vehicles to substitute with the ego-vehicle.

5.7.3 Algorithm object

The field *algorithms* in the configuration *JSON* specifies which algorithms will the simulator benchmark. This field has the following properties:

- *name* – Describes the name of the algorithm. This field is used by the simulator to name produced videos and images.
- *cmd* – Specifies a command to run the algorithm.

5.8 Data loading

When the configuration is parsed, the program loads the trajectories and the map from the disk. Afterward, the program projects the positions of the objects onto the map. Using the coordinates of the top-left and bottom-right corner of the map (as mentioned in Section 4.4), we compute the size ratio as shown in (1),

$$\begin{aligned} m &= [w, h]\,, \\ r &= m \oslash (t - b)\,, \end{aligned} \tag{1}$$

where w and h denote the width and height of the source, t the coordinates of the top-left corner and b of the bottom-right, and r is called the *calibration ratio.* This ratio describes the value by which the object sizes are multiplied to fit the map scale.

Using the values from (1), we transform the position d of each object into d', a position projected onto the map. A position has three properties, which need to be transformed – the center point d_c, the size d_s, and the direction d_d. This is shown in (2).

$$\begin{aligned} d'_c &= (d_c \oslash r) + t \\ d'_d &= d_d \odot r \\ d'_s &= abs(d'_s \oslash r) \end{aligned} \tag{2}$$

As the values of r might not always be positive, the direction has to be multiplied by the calibration as well. We use absolute value for the size for a similar reason, as size must always be positive.

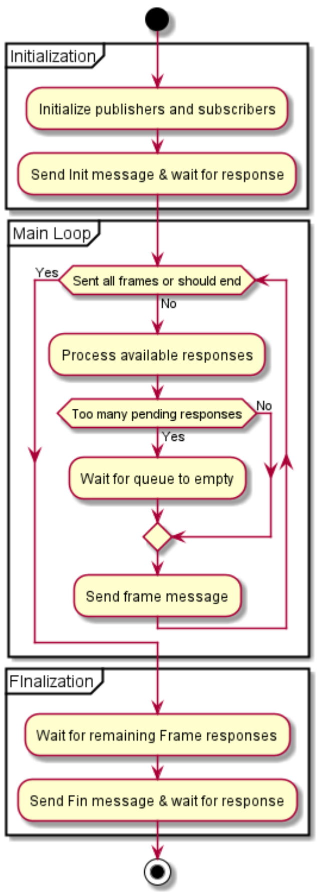
When the trajectories and the map are correctly loaded, and the object positions are transformed, the program then performs a check, whether the map is suitable for routing. The program determines the starting and destination *lanelet* for each ego-vehicle and attempts to find a (shortest) path between them. If a path is not found, an exception is thrown. In such a case, we should check our map for errors as described in Subsection 4.3.2.

In any case, a routing graph is plotted to a file called '`<name>_routing.osm`', containing the name of the scenario, as described in Subsection 5.7.2. We can inspect the file using an editor such as *JOSM* to check whether the graph is connected correctly.

5.9 Simulation

After the data are loaded and validated, a simulation routine is executed for every combination of algorithm, scene, and ego-vehicle. A simulation consists of four phases: initialization, main loop, finalization, and serialization of results.

The first three phases are the implementation of the more abstract communication model discussed in Section 5.3. The flow of the simulation procedure is described in Figure 16 from beginning to end, where every action is assigned to an appropriate phase.



Initialization
Initialize publishers and subscribers
Send Init message & wait for response
Main Loop
Yes
Sent all frames or should end
No
Process available responses
Too many pending responses
No
Yes
Wait for queue to empty
Send frame message
FInalization
Wait for remaining Frame responses
Send Fin message & wait for response

Figure 16 - Simulation procedure

5.9.1 Initialization

In the initialization phase, the simulator instantiates three publishers and subscribers for all the types of messages (Section 5.4) sent during the communication. The topic names of these channels are specified in the class *Constants,* a class holding several constants to be used by the simulator components.

The simulator runs the algorithm on a new thread and prepares an *Init* (Subsection 5.4.3) message. This includes filling the fields, which are already set in the configuration (*osm_path, name, id, fps*), while others are extracted from the data (*start, end, size, maxSpeed, initialSpeed*). The *size* is calculated as the median of all the observed sizes of the object, while the *maxSpeed* and *initialSpeed* are consequently set as the maximum and initial velocity of the original vehicle. The values *start* and *end* are set as the first and the last observed location of the vehicle.

This *Init* message is sent via the simulator's publisher, and then the simulator waits for a response from the algorithm. The main simulation loop starts after receiving the response.

5.9.2 The main loop

During the main loop, the simulator reads the trajectories frame by frame and creates a *Frame* message (Subsection 5.4.4), containing the positions of every object that is present on said frame, repeating for each one of the frames. If possible, it starts publishing frames four seconds before the vehicle enters the scene. This way, the algorithm is provided data about the scene without the need to make decisions for the ego-vehicle. However, this is not always possible, as the object may be present in the scene from the beginning. The message is then sent via the simulator's publisher.

If the simulation is running with a time limit set (as described in Subsection 5.7.1), the simulator limits the rate of sending the messages according to the *fps* parameter. Otherwise, the simulation is executed as fast as possible.

When receiving a *FrameResponse* (Subsection 5.4.4), the simulator performs several actions depending on the *status* contained in the response:

- *RUNNING* - The simulator checks for any errors, as described in Section 5.5and stores them in memory.
- *FIN* - The simulator stops publishing new frames and ends the main loop.

In any case, the simulator increments the counter and publishes the frame for visualization. The simulation also ends if there are no more frames available to publish.

5.9.3 Pipelining

When sending the *Frame* messages, the first approach was to simply wait to receive a response (as described in Subsection 5.9.1). An illustration is shown in Figure 17.

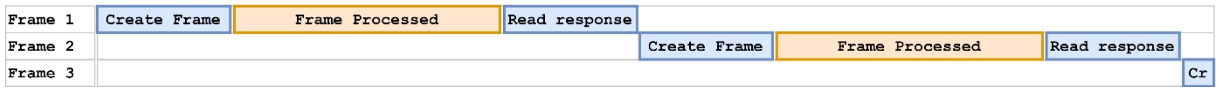


Figure 17 - Non-pipelined communication

The fields in blue color (*Create frame* and *Read response*) denote operations made by the program, while orange (*Frame processed)* denote that the message is either in a queue or is being processed by an external algorithm. This idling of the simulator created a significant overhead.

By introducing a pipelining scheme, the simulator is able to create or respond to messages while other messages are being processed (Figure 18). Therefore the simulator does not need to wait for every response actively. We keep track of two values, the id of the last published frame and the id of the last received frame. If the difference between the two values exceeds the preset queue size, the simulator stops sending the messages and wait for the queue to clear.

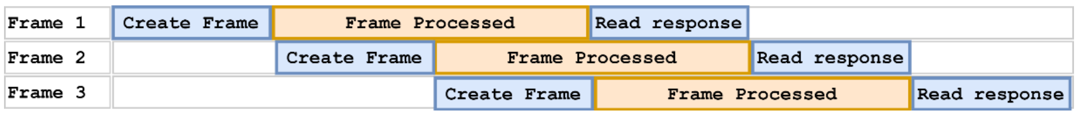


Figure 18 - Pipelined communication

To get more insight on how much the pipelining affected the speed of the simulator, we conducted an experiment by running implementations on four forks, each consisting of 10 iterations. We made use of equations proposed by Kalibera and Jones [35]. We measured the time taken by one simulation and calculated the average running time for each fork $\bar{Y}_{j_n}$, as well as the total average $\bar{Y}$. Using (3, we can calculate the half-width h of the confidence interval for 95% confidence:

$$I_c = t_{1-\frac{\alpha}{2},\nu}\sqrt{\frac{S_n^2}{r_n}}, \tag{3}$$

where $t_{1-\frac{\alpha}{2},\nu}$ is the $\left(1-\frac{\alpha}{2}\right)$ – quantile of *t*-distribution with $\alpha = 0.05$ (given by 95% confidence) and $\nu = r_n - 1$ degrees of freedom, n represents the number of levels (in this case $n = 2$), and r_n stands for the number of repetitions at the highest level ($r_n =$ 4).

S_n^2 denotes the sample variance of execution means and can be calculated using the following equation (4):

$$S_n^2 = \frac{1}{r_n - 1} \sum_{j_n=1}^{r_n} (\bar{Y}_{j_n} - \bar{Y})^2 \,. \tag{4}$$

We show the results in Table 1.

	Time [seconds]					
Algorithm	$\bar{\boldsymbol{Y}}_1$	$\bar{\boldsymbol{Y}}_2$	$\bar{\boldsymbol{Y}}_3$	$\bar{\boldsymbol{Y}}_4$	$\bar{\boldsymbol{Y}}$	$\boldsymbol{h}$
Pipelined	3.68	3.68	3.65	3.67	3.669	$\pm$ 0.020
Non-pipelined	45.41	45.51	45.58	45.53	45.509	$\pm$ 0.112

Table 1 - Execution times for different implementations of the main loop

We can then estimate the speedup R between the implementation with pipelining and without. Using (5), we calculate the speedup as

$$\begin{aligned} R &= \frac{\bar{Y} \cdot \bar{Y}' \pm \sqrt{(\bar{Y} \cdot \bar{Y}')^2 - (\bar{Y}^2 - h^2)(\bar{Y}'^2 - h'^2)}}{\bar{Y}^2 - h^2} \,. \\ &= 12.403 \pm 0.076 \,. \end{aligned} \tag{5}$$

The experiment was conducted on a machine equipped with Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz and 16 GB of RAM with installed *ROS Melodic 1.4.1*.

5.9.4 The finalization

When the main loop finishes, the simulation enters a finalization phase. During this phase, the simulator waits for all the pending messages to be processed. When all the remaining responses have arrived, the simulator sends a *Fin* message (Subsection 5.4.5) and waits for the response to arrive.

After reading the response, the simulator evaluates whether the vehicle has arrived at its destination and adds an error if not. Finally, it shuts down all publishers and subscribers and waits for the algorithm subprocess to terminate.

5.9.5 Writing the results

After the simulation has finished, all encountered errors are extracted and serialized to a *CSV* file specified in the configuration (Subsection 5.7.1). To decrease the size of the resulting table, the errors are written in the form of intervals. An example is shown in Table 2.

LARGEJUNCTION_SIMPLEALGORITHM_59

SEVERITY	Status Type	Frame Start	Frame End	Obj Id
OK	NO ERROR	0	376	-
HARD_ERROR	OBJ_COLLISION	377	432	10
OK	NO ERROR	433	446	-

Table 2 - Example of the results in CSV format

For automated benchmarking, the results are also serialized to a *JSON* file, which should be easier to parse than the *CSV*.

5.9.6 No messages lost

As stated in Subsection 5.9.2, we use a queue to send the *Frame* messages (Subsection 5.4.4). However, the queue has a limited size and works in a *FIFO* manner. If we push a new message while the queue is full, the oldest message gets deleted. An algorithm we use to ensure that no messages get lost is described in the following pseudocode (Figure 19).

```
S := "the size of the queue"
o := "id of the last sent frame"
i := "id of the last received frame"

ros::spinOnce()

if o - i == S then
    while o - i > S / 2 do
        ros::spinOnce()
        sleep()
    end
end

publishFrame()
```

Figure 19 - Pseudocode of waiting for the message queue to empty

The *spinOnce* is a function provided by *roscpp* which executes all of the callbacks waiting to be processed (this includes waiting on any *FrameResponse* (Subsection 5.4.4) messages).

The condition causes the program to wait until the queue is at least half empty when the amount of pending messages is equal exactly to the size of the queue. However, the pending messages can be in three different states:

1. In the queue waiting to be picked up by the algorithm.
2. Being processed by the algorithm.
3. In the response queue waiting to be read by the simulator.

In the worst case, all of the messages are stored in one of the queues. But since the number of messages cannot be larger than the queue size, they all fit. Therefore, no unprocessed messages are lost.

5.9.7 Deadlock prevention

As we are dealing with multi-process communication with synchronization, we are at risk of encountering a loop of infinite waiting.

One of the potential threats arises from the different interleaving of publishers and subscribers being created. If a publisher sends a message while no subscriber is listening, the message gets lost, and the process, waiting for the message, gets stuck forever.

This issue is solved by using 'latched' queues. These queues cause the last message to be saved for any subscriber in the future. Hence, if any subscriber connects after the message has been sent, they are still able to receive the message.

However, even with using latched queues, we still can encounter a deadlock (e.g., algorithm crashing, networking error). To avoid deadlocks, we can use a configurable timeout in the *Simulation* class. If any waiting exceeds this limit, an exception is thrown, and the simulation is terminated.

5.10 Visualization

As shown in the configuration (Subsection 5.7.1), the simulator supports two different ways of visualizing the scene. This includes either a real-time method using *rViz* or plotting into a video file using *OpenCV*.

To export data in an arbitrary format, we introduce the *AbstractPublisher* class. An implementation of this class can be registered to the *Simulation* class, which in turn handles calling the virtual methods. Both *OpenCVPublisher* and *RVizPublisher* are such implementations and are illustrated by the diagram in Figure 20. Both of the classes implement the virtual methods while also introducing their own fields and methods. Some private members are not included in the diagram to improve readability. A description of the virtual methods follows.

- *initPublisher* – Called once before the simulation starts and initializes the object.
- *publishFrameMsg* – Called for every *FrameResponse* received by the simulator.
- *tearDown* – Called once after the simulation ends.

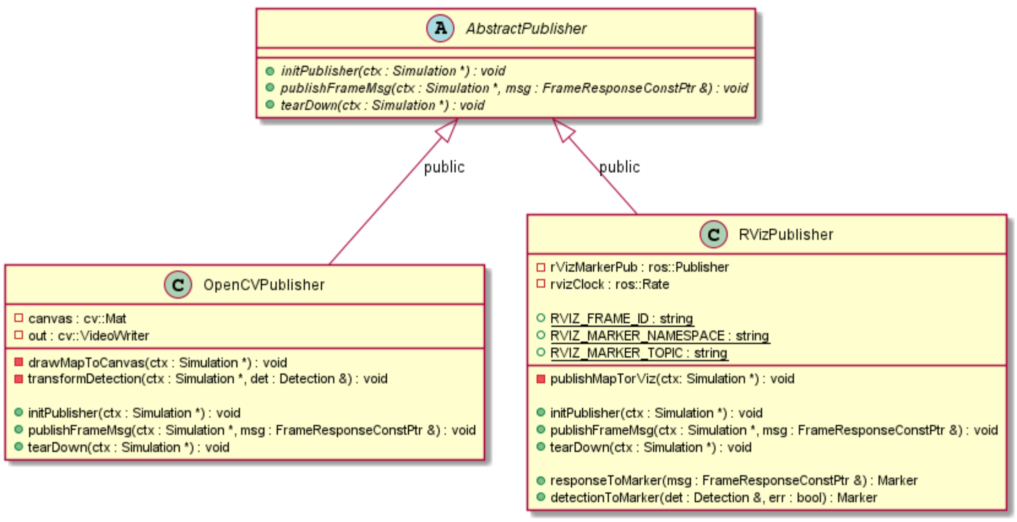

AbstractPublisher
initPublisher(ctx : Simulation *) : void
publishFrameMsg(ctx : Simulation *, msg : FrameResponseConstPtr &) : void
tearDown(ctx : Simulation *) : void
public
public
OpenCVPublisher
canvas : cv::Mat
out : cv::VideoWriter
drawMapToCanvas(ctx : Simulation *) : void
transformDetection(ctx : Simulation *, det : Detection &) : void
initPublisher(ctx : Simulation *) : void
publishFrameMsg(ctx : Simulation *, msg : FrameResponseConstPtr &) : void
tearDown(ctx : Simulation *) : void
RVizPublisher
rVizMarkerPub : ros::Publisher
rvizClock : ros::Rate
RVIZ_FRAME_ID : string
RVIZ_MARKER_NAMESPACE : string
RVIZ_MARKER_TOPIC : string
publishMapTorViz(ctx: Simulation *) : void
initPublisher(ctx : Simulation *) : void
publishFrameMsg(ctx : Simulation *, msg : FrameResponseConstPtr &) : void
tearDown(ctx : Simulation *) : void
responseToMarker(msg : FrameResponseConstPtr &) : Marker
detectionToMarker(det : Detection &, err : bool) : Marker

Figure 20 - Deriving from AbstractPublisher

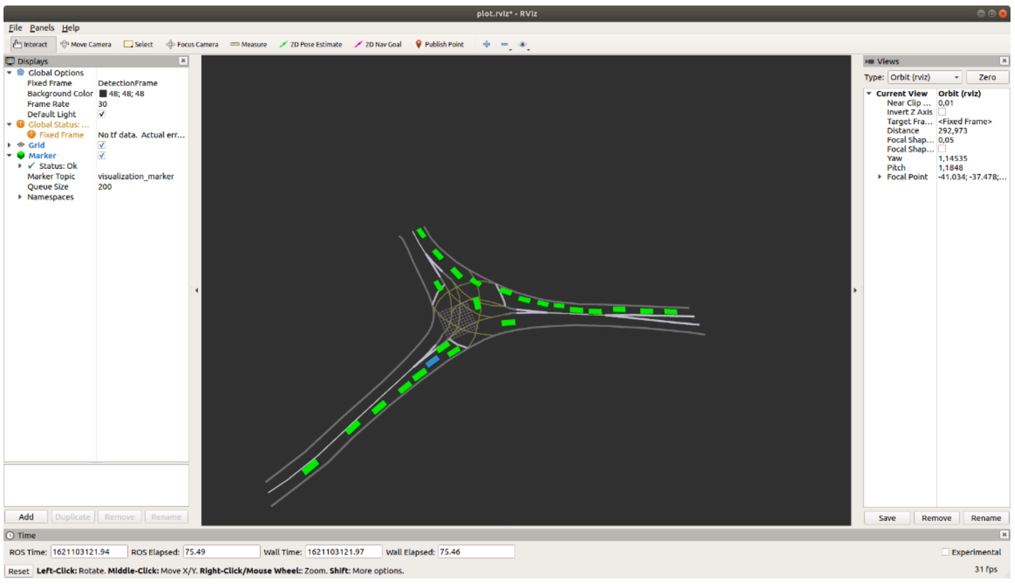
5.10.1 Publishing to rViz

When initialized, the class instance creates a publisher, which sends *Marker* messages over a designated topic. The *topic name*, *marker namespace*, and *frame id* are all defined by constants. In this phase, the map is also converted into a set of markers and sent to *rViz*.

When publishing frames, all objects are converted to *Marker* messages [36] and assigned an appropriate color using the *Palette* class. Objects associated with an error are displayed with red color. All the messages are published via the *rViz* publisher created during the initialization. The rate of sending the messages is also limited to the source video *fps*.

During the tear-down phase, one more message with the action `DELETEALL` is sent, which ensures the removal of any leftover markers in the scene.

An example of a scene visualized via *rViz* is shown in Figure 21, where the ego-vehicle is denoted by a blue cuboid marker, while other vehicles by a green one.



Global Options	
Fixed Frame	DetectionFrame
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: ...	
Fixed Frame	No tf data. Actual err...
Grid	✓
Marker	✓
Status: Ok	
Marker Topic	visualization_marker
Queue Size	200
Namespaces	

Type: Orbit (rviz)

Current View	Orbit (rviz)
Near Clip ...	0,01
Invert Z Axis	☐
Target Fra...	<Fixed Frame>
Distance	292,973
Focal Shap...	0,05
Focal Shap...	☐
Yaw	1,14535
Pitch	1,1848
Focal Point	-41.034; -37.478;...

ROS Time: 1621103121.94 ROS Elapsed: 75.49 Wall Time: 1621103121.97 Wall Elapsed: 75.46 ☐ Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options. 31 fps

Figure 21 - Using rViz to visualize a scene

5.10.2 Plotting via OpenCV

Using *OpenCV*, the simulator is able to plot the scene into a video file with the same parameters (width, height, and fps) as the source video. The program instantiates a *VideoWriter* object during the initialization. This object is able to serialize frames into a video file. The frames are represented as matrices of BGR values, where each element corresponds to one pixel (essentially a bitmap). We are using matrices with the map drawn as a canvas.

For each frame, the publisher creates a copy of the canvas and draws all the objects onto it. Any object with an error associated with it is drawn using red color. After that, the matrix is appended to the video via the *VideoWriter*.

During the tear-down phase, the resources are released, and the video becomes a playable *mp4* file. An example is shown in Figure 22, displaying a single frame from such video. The ego-vehicle is displayed as a blue rectangle, while the other objects are green, with yellow identifiers. The target destination of the agent is denoted by the blue circle. The road borders, road surface markings, and virtual lines are also displayed in the video.

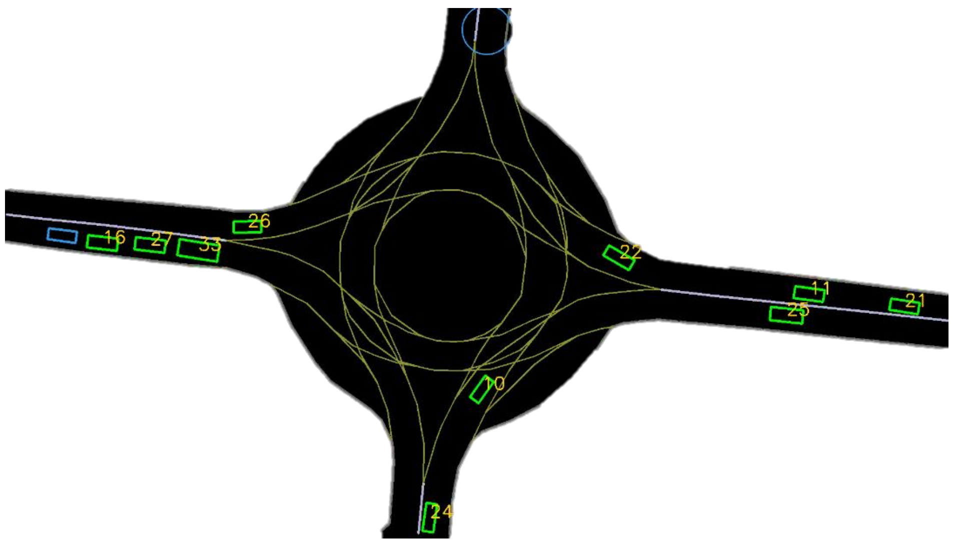


Figure 22 - Roundabout scenario visualized with OpenCV

5.11 Shared library

Apart from the executable binary, we also create a shared library from the source code, allowing other projects to use the features of the *hav_simulator* package. The library includes functions for geometry or visualization and also different constants which need to be set correctly for communication.

To avoid any future naming conflicts, we are using the namespace *hav_simulator* for all symbols in the package.

5.11.1 SFINAE pattern

One of the challenges when writing different geometry functions is that there are several different implementations for data types, such as *Vector2f* (an x, y vector of the type *float*). The implementations often come from different libraries (e.g., *Eigen*, *lanelet2, OpenCV*, *ROS*, ...), and functions from each library expect their own implementation.

Moreover, almost in every case, there are no implicit conversions available between the types.

One approach is to use overloading to specify the implementation for every type. However, this is very time-consuming, as there are multiple implementations for several structures, and we ought to manually write an overload for each one of them. Besides this, any implementation introduced in the future would be lacking its overload.

Since we use only the properties x, y of a *Point* (or *Vector*) in most of the functions, we can use templates to achieve the result. This means that we need to provide only one implementation, and the compiler generates all the required overloads. However, a significant shortcoming is that this requires all of the properties x, y to be accessible in the same manner. An example of a template can be seen in Figure 23. This way, an overload is created if the fields are accessible as a property. In contrast, if the fields are accessible via a method, this results in a compilation error.

```
template<class T>
float maximum(T a, T b){
    return max(a.x, b.x, a.y, b.y);
}
```

Figure 23 - A template example

As stated earlier, it is not desirable to use standard overloading or templating, as we want to maximize compatibility. We cannot use polymorphism, as the classes from different libraries rarely extend the same base. Instead, we need to provide different overloads based on the properties (traits) of the different classes.

A solution to this is to use the *SFINAE* (*substitution failure is not an error*) technique. This allows ignoring certain templates during the compile-time under specific conditions [37]. An example is shown in Figure 24, where the function is enabled or ignored, depending on the characteristics of the type *T*.

```
template<typename V, typename T>
typename std::enable_if<!hasXY<T>::value, V>::type
extractVec(T t) {...}
```

Figure 24 - Using enable_if SFINAE

Using this technique, we have to provide only two overloads for certain functions - one for the implementation of *Point* with x, y as fields, and a second one which uses functions. It is not necessary to know anything about the implementation, only how to access the required properties.

Thus, we can use the same implementation for multiple types sharing certain traits. This approach makes it possible to use the functions without casting to a specific type and does not restrict the users of the library from using types of their own.

5.12 Exceptions

When dealing with exceptions, we want them to be easily identifiable. If an exception is thrown by a library method, it is often descriptive enough, as the libraries implement their exception structures.

We use a similar approach and implement different exception types depending on the reason they were raised. This way, we can use a specific exception type in the *catch* clause (or a unit test) rather than raising a pre-defined exception (such as *runtime_ error)* and deducing the type from the error message.

5.13 Testing

For unit testing, we provide a separate CMake target using the *gtest* framework [38]. The tests are spread across multiple headers according to the tested functionality and can be run together or separately. The tests are using files stored in the folder *data* as resources. This folder needs to be set as the working directory when the tests are run, so the program can find the required files. This is handled automatically using *catkin* in the project *CMakeLists* file [39]. However, when using another tool (such as *CLion*), the path may need to be set manually.

5.13.1 Unit testing

Apart from testing our own code for bugs and unintended behavior, we also want to ensure that library functions behave as we expect them to. For example, we may expect a method to return an object by value, so it is safe to modify it. If in another version of the library the object is returned by reference (e.g., to lessen the memory impact), mutating this object can lead to modifications of unwanted data. This way, we diminish the chance of any future update introducing unwanted behavior. For all of this, we are using parameter files saved in the *data* folder.

In the unit tests, we focus on testing whether the program detects wrong inputs and throws an appropriate exception. We also test our geometry functions, the correctness of the projections, and the loading of the map data. One header is focused on the work with the map, e.g., correct detection in which *lanelet* is the object located and that it can detect the existence of a path (or lack thereof) between two *lanelets*.

A special class of tests is dedicated to the correctness of the SFINAE overloads. These tests are done by using a large variety of types when calling our generic functions. The existence of a valid overload is checked during the compile-time and its correctness by running the tests.

Moreover, we are also testing the full simulation routine. For this, a reduced dataset is used on a simplified map. After that, we create an algorithm stub on a separate thread and run the simulation. Afterward, the results are checked, whether a correct number of errors was detected in the correct order. We are also simulating a deadlock and checking whether the deadlock is detected, and the simulation is terminated in time.

5.13.2 Checking for memory errors

We tested the code for erroneous memory accesses to ensure that the program does not end with a segmentation fault or act with undefined behavior.

To test for memory errors, we used *Memcheck*; a tool implemented using the *Valgrind* instrumentation framework [40]. This tool checks the addressability of every byte, the definedness of every bit, and tracks all allocated heap blocks during the runtime of the program.

We ran the whole simulation using the *Memcheck* tool with different configurations several times to check for memory errors. It was necessary to modify some parameters like the *Frame* response timeout duration, as this tool hinders the performance of the profiled application (in our case, the program was more than 100 times slower). The tool did not find any memory errors but reported a potential leak, as 1352 bytes were ‘possibly lost’.

After closer inspection, as hinted by the trace containing the symbols `ld-2.27.so` and `libglib-2.0.so`, we deduced that the leak is introduced by the external library *glib* [41]. Given that the size of the leak did not increase across multiple executions and that the library developers confirmed that the ‘possible’ leaks are not suppressed [42], it is safe to assume that the potential leak does not pose a threat.

Other than *Memcheck*, we also used *Address Sanitizer* to check for memory errors. *Address Sanitizer* checks for out-of-bounds memory accesses at the cost of a 73% slowdown [43] in contrast to a 10000% slowdown of *Memcheck*. This lesser slowdown allows us to use the tool even when running a program with such a configuration that would take *Memcheck* several hours to analyze. Using *Address Sanitizer*, we did not find any error in our application.

5.14 Docker

The environment for the package to compile and run is fairly complex and contains a lot of different libraries, modules, and other dependencies. To ensure the runnability of our system and the reproducibility of the results, we have to describe in detail the parameters of the used environment.

We could potentially provide an installation script, which would set up the environment on a given machine. However, we realize that some users may not want to install all the necessary dependencies to their machine just to be able to use this package. Moreover, some of the dependencies might be in direct conflict with the packages already installed on the system and could potentially break existing programs.

Boettiger [44] described issues with reproducibility concerning scientific research. In this article, the author mentions *Docker*, a containerization technology [45], as one of the solutions. Following this example, we provide a *Dockerfile* with the instructions to assemble an image with all the dependencies pre-configured. This image can then be used to develop, compile, and execute the simulator or its utility modules.

Another benefit of the *Dockerfile* is that it can also serve as a recipe for setting up the environment locally. If someone wants to run the project directly on their machine, they can follow the instructions specified in the *Dockerfile* to set up their machine accordingly.

6 Self-driving algorithm

The last objective is to design and implement a path planning algorithm with the ability to communicate with the simulator. There are several reasons why this is necessary. First of all, an algorithm is required to test the simulator end to end. Secondly, the algorithm serves as a practical example of how communication works and how to use the chosen libraries. Lastly, we want to provide a foundation on which more complex algorithms can be built.

We are using several symbols defined in the simulator package, which we include as a dynamic library. This way, we can benefit from the already implemented utilities for geometry and plotting.

Thus, we implemented three algorithms derived from a base class which is described in Section 6.2. Two of these algorithms should be considered stubs, as they do not perform any computations and are trivial. Although simple, the third one can predict the movement of other vehicles and navigate the ego-vehicle to reach its destination.

It should be noted that the algorithm, in the context of this work, should serve as a way to validate the simulator. Thus, it is not considered wrong if the algorithm causes the vehicle to collide with other objects or does not reach its destination. Such events should be present, as we want to ensure that they are picked up by the simulator.

6.1 Analysis

A main requirement for the algorithm is to solve the planning problem by proposing a viable path, or eventually a trajectory (a path with a schedule) between two points. There are several algorithms and libraries dealing with this task, as reviewed by Tsardoulias et al. [46].

Based on the research conducted by Tong et al. [47] we discovered, that the necessary basic path planning functionality is already included in the *lanelet2* library. Thus we decided to build our solution on top of this module to demonstrate the end-to-end usability of our system. The routing module converts the map in *lanelet2* format to a 'routing graph', where every *lanelet* is assigned to a node and constructs the edges based on the reachability between adjacent *lanelets.*

Using this routing graph, we can perform several operations, such as determining the optimal path from the start to the destination, including possible lane changes. It is also possible to predict routes and points of conflict for other vehicles [13]. An example of a routing graph on a roundabout is shown in Figure 25.

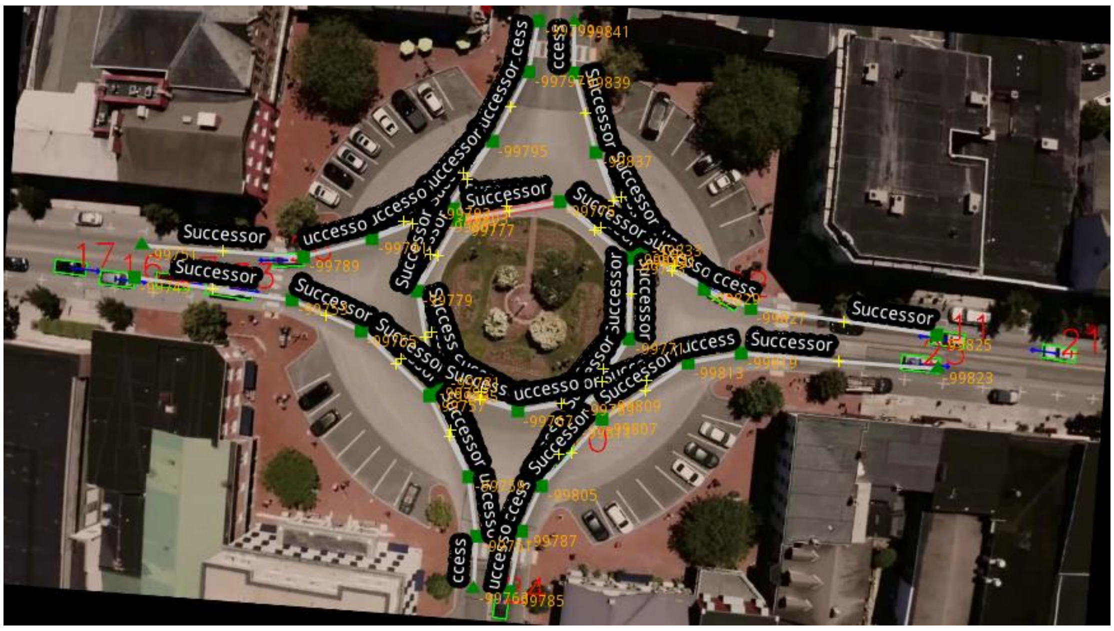


Figure 25 - A routing graph calculated for a roundabout

Moreover, we also need a physical model for our vehicle. For our algorithm, we use a *Point-Mass Model*, one of the models proposed by Althoff et al. [4]. This model sets a limit on the maximum absolute acceleration. However, as this model does not differentiate between acceleration and deceleration (braking), we limit the value by the maximum braking speed. As written in [48], most of the modern vehicles are equipped with an anti-lock braking system, putting their maximum deceleration close to the value of $g = 9.8\frac{m}{s^2}$ [48]. We use this value for our theoretical model.

6.2 Algorithm interface

As a base class for the algorithms to inherit, we provide class *AbstractAlgorithm.* We use the *Template Method* pattern to deal with the communication with the simulator.

Inside this template method, the algorithm handles the creation of the necessary publishers and subscribers and calls the hooks, defined as virtual methods.

We require the algorithm to implement three different hooks, specifically the callbacks for the *Init*, *Frame*, and *Fin* messages (Section 5.4). The algorithm should process the contained information and assemble responses for the *Frame* messages. These responses have to contain the position of the ego-vehicle for the frame, which the algorithm had calculated using the gathered information.

It is not required to call the hooks or create any publishers or subscribers for the messages, as this is already handled by the template method.

The template method, together with the virtual hooks, is illustrated by a diagram shown in Figure 26. This diagram provides an example of two different algorithms extending the base class. The callbacks are all virtual functions that the algorithm has to implement, while *run()* is the public template method invoking the callbacks.

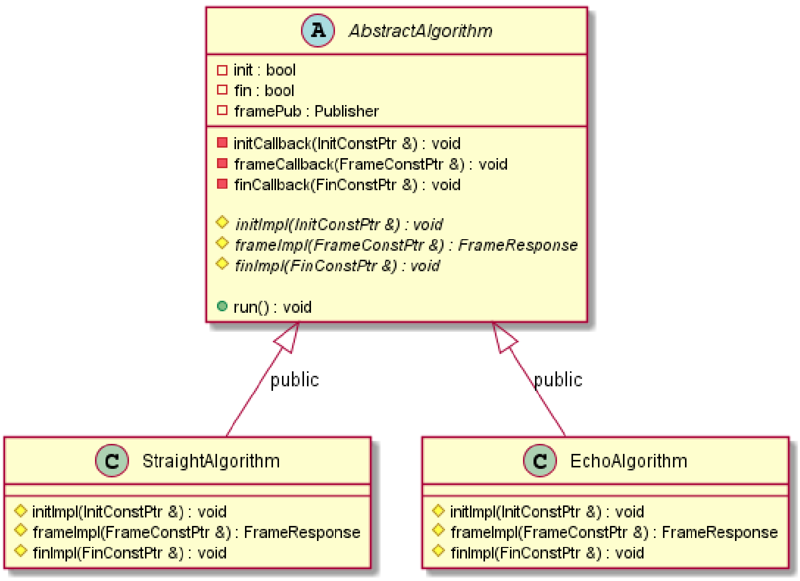

A AbstractAlgorithm
init : bool
fin : bool
framePub : Publisher
initCallback(InitConstPtr &) : void
frameCallback(FrameConstPtr &) : void
finCallback(FinConstPtr &) : void
initImpl(InitConstPtr &) : void
frameImpl(FrameConstPtr &) : FrameResponse
finImpl(FinConstPtr &) : void
run() : void
public
public
C StraightAlgorithm
initImpl(InitConstPtr &) : void
frameImpl(FrameConstPtr &) : FrameResponse
finImpl(FinConstPtr &) : void
C EchoAlgorithm
initImpl(InitConstPtr &) : void
frameImpl(FrameConstPtr &) : FrameResponse
finImpl(FinConstPtr &) : void

Figure 26 - Using the template method pattern with two different implementations

6.2.1 The template method

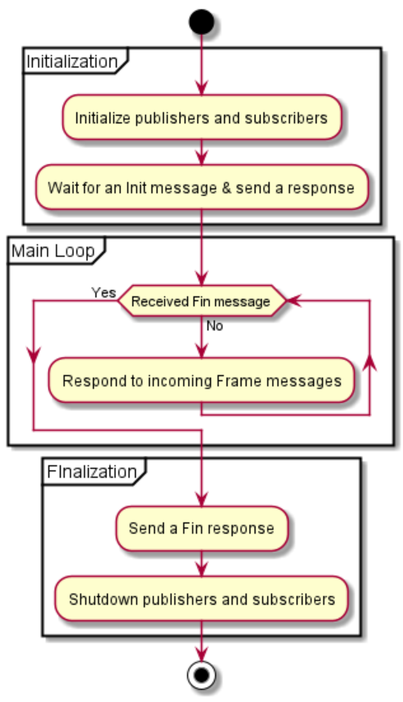

Initialization
Initialize publishers and subscribers
Wait for an Init message & send a response
Main Loop
Yes
Received Fin message
No
Respond to incoming Frame messages
FInalization
Send a Fin response
Shutdown publishers and subscribers

Figure 27 - Flow of the template method

The purpose of the template method (Figure 27) is to perform the communication of messages with the simulator. Firstly, the algorithm creates three publishers and subscribers for all the message types. The parameters for the publishers and subscribers are extracted from the simulator shared library. As explained in Subsection 5.9.7, we are using latched publishers to ensure that no messages are lost.

After creating the publishers and subscribers, the algorithm waits for an *Init* message (Section 5.4). After receiving this message, the algorithm publishes an *InitResponse* and starts the main loop.

In the main loop, the algorithm keeps reading the incoming *Frame* messages and generates responses for every single one of them. The main loop ends when the algorithm receives a *Fin* message, which signals that the algorithm should finish.

After exiting the main loop, the algorithm sends a response to the *Fin* message and shuts down all the publishers and subscribers. When this is done, the algorithm ends.

6.3 The algorithm stubs

During the early development of the simulator, two different algorithms were created. The algorithms themselves are considered to be stubs and should be used only for the purposes of demonstrating the simulator. Using these algorithms, we were able to verify the process of initialization, simulation, visualization, and the detection of different errors during the early development of the simulator.

6.3.1 Straight algorithm

The first implementation, *StraightAlgorithm*, starts by ignoring any information provided in the *Init* message (Section 5.4) except for the id of the ego-vehicle. Afterward, the algorithm responds to any *Frame* message with the status `WAITING`. After receiving a first message containing a vehicle with the specified id, the algorithm saves the orientation of this vehicle.

Afterward, the algorithm keeps moving the vehicle forward at a constant speed while colliding with any obstacle in its path. This algorithm can be useful if we want to test whether the simulator is detecting collisions correctly since this algorithm will probably produce a substantial amount of them.

6.3.2 Echo algorithm

The second algorithm, called *EchoAlgorithm,* saves only the id of the ego-vehicle in the *Init* message (Section 5.4). Until a first *Frame* message containing a vehicle with this id is received, the algorithm responds with the status `WAITING.` Subsequently, the algorithm sends back the ground-truth position with the status of `RUNNING`. This is repeated until a *Frame* is received in which the vehicle is missing. This causes the algorithm to switch to the status `FIN`, in which it stays until the end.

This algorithm was very useful during the development of the simulator, notably the communication between the simulator and the algorithm. Similarly, this algorithm was used to test the different methods of visualization.

6.4 The autonomous driving algorithm

The last task was to implement a functional self-driving algorithm. This algorithm should be able to find a path to the destination and adjust the vehicle's velocity and direction to reach the destination while avoiding collisions.

6.4.1 Initialization

After an *Init* message is received, the algorithm loads the map and uses the *lanelet2* framework to build a routing graph from the scene. The graph is then used to acquire a path in the form of a sequence *lanelets*, which can be traversed to reach the destination.

Using this sequence, the algorithm calculates the path from start to destination by computing the center lines for the selected *lanelets*. The path is then smoothed by limiting the maximum angle (set as a constant) between two consecutive points.

The algorithm stores the information contained in the message in memory. This includes data such as the initial and maximum velocity, and the size of the vehicle. These parameters are later used when calculating responses to the *Frame* messages.

The algorithm also plots the calculated trajectory into an image for debugging purposes. An example is shown in Figure 28, where the vehicle has to drive through a roundabout. The path is displayed as a red curve.

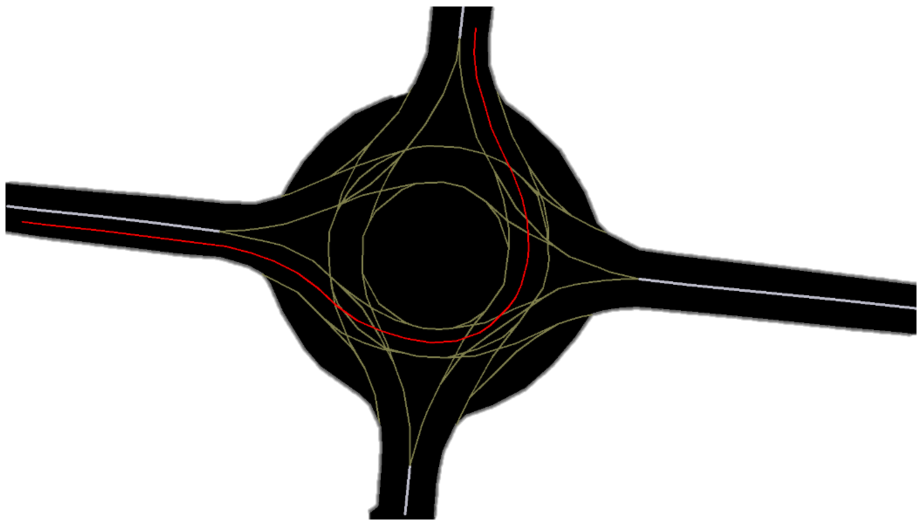


Figure 28 - Path of the vehicle when driving through a roundabout

6.4.2 Frame response step

When responding to the *Frame* message, if the object replaced by the ego-vehicle is not yet present in the scene, the algorithm responds with the status `WAITING`. After receiving the first message, the algorithm switches to the status `RUNNING` and responds with the position of the vehicle.

When receiving a *Frame* while the status is `RUNNING`, the algorithm first updates the positions of other vehicles in the scene. Then, it calculates their predicted positions according to their orientations and their instantaneous velocities. The velocity of every vehicle is estimated using the distance between the current position and the position from half a second ago. We chose half of a second to compensate for the noise in the positions of objects between consecutive frames but to still be able to react fast enough. The speed of the vehicle also determines the predicted future positions, so in case that the current state is expected to lead to a collision, there is still enough time to brake. The predicted vehicle positions are inflated by a percentual amount to account for the uncertainty and also to avoid near-misses if possible.

The algorithm attempts to estimate whether it is necessary to change the velocity. The algorithm tends to keep the vehicle at the maximum allowed speed. However, when this leads to a collision with another vehicle, the algorithm causes the vehicle to brake and decrease the velocity or even bring the vehicle to a full stop.

Moreover, the algorithm attempts to keep a safe distance between itself and any other vehicle in front of it. For this, we use the routing graph to search for any vehicle in front of the car, which is in the same or the following *lanelet*. Under §19 of the Traffic regulation act ('Zákon o silničním provozu' in Czech), drivers should keep enough distance between the vehicles [49]. The law does not specify this distance, but the usual value is the distance covered by the vehicle at the current speed in two seconds [50]. However, this value takes into account the time necessary for a human driver to react, which can be as long a 1.5 seconds. As the algorithm runs with the frequency of *fps* we can set the reaction time to the value of fps^{-1}.

After adjusting the velocity according to the behavior of other vehicles, the algorithm steers the vehicle to follow the pre-computed trajectory. After this, the algorithm calculates the new position of the ego-vehicle and sends it to the simulator in the form of a *FrameReponse* message.

For debugging purposes, if the symbol `DRAW_DEBUG_IMAGES` is defined, the algorithm also saves an image with objects and their predicted locations drawn for every frame. An example image is shown in Figure 29, where the ego-vehicle is denoted by the blue rectangle and other vehicles by green or magenta. The yellow rectangles are the predicted positions of other vehicles. The blue number describes the instantaneous velocity of the vehicle in meters per second. In the image, the ego-vehicle is braking, as it is too close to the vehicle in front. This is symbolized by coloring the vehicle with magenta rather than green.

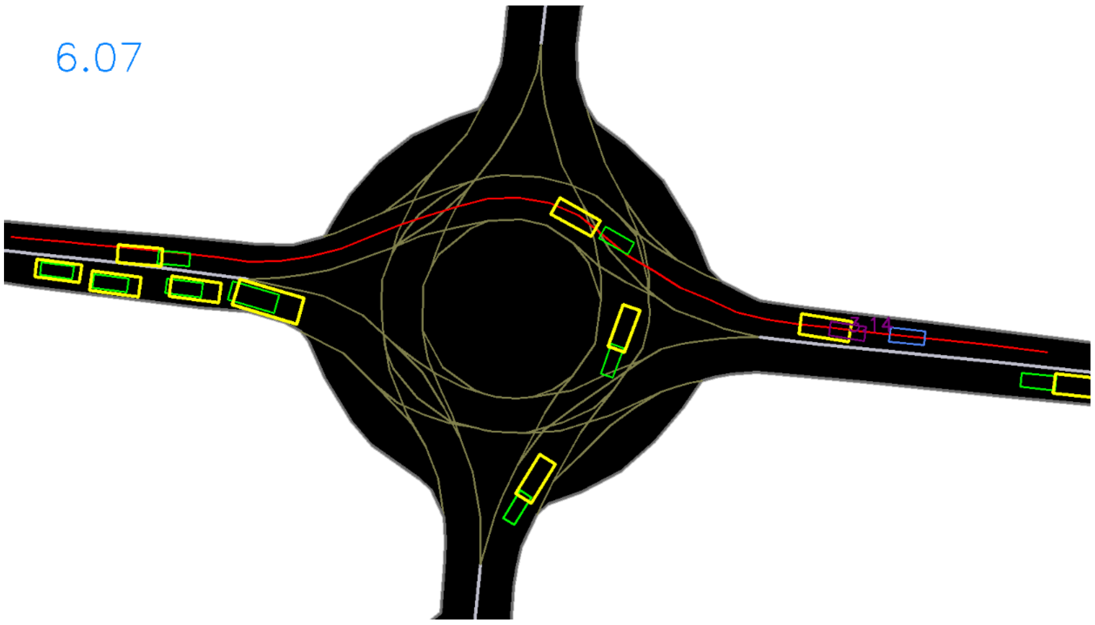


Figure 29 - The visualized model of the algorithm

6.4.3 Finalization

After reaching the destination with the ego-vehicle, the algorithm sends a *FrameResponse* with the status `FIN` to signal the simulator that the simulation can be terminated. Since the sending of the messages is pipelined, there still might be unprocessed *Frame* messages enqueued. Each of the messages should be replied to with the status `FIN` until a *Fin* message is received. The switching between states is illustrated by the state diagram in Figure 30.

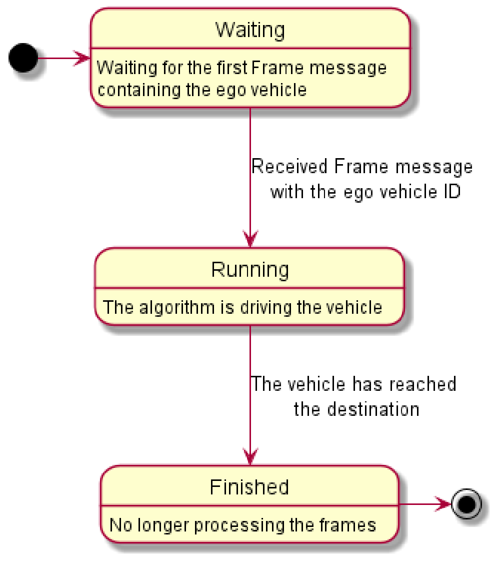

Waiting
Waiting for the first Frame message containing the ego vehicle
Received Frame message with the ego vehicle ID
Running
The algorithm is driving the vehicle
The vehicle has reached the destination
Finished
No longer processing the frames

Figure 30 - The algorithm switching between states

6.5 Testing

For the purposes of testing the whole system, we assembled a dataset consisting of trajectories and *lanelets* from six different scenes. These scenes include a simple highway, roundabout, and two junctions. The last two scenes are from a junction and a roundabout and both are fairly complex.

We have the following requirements for the system.

- The system is able to benchmark one or multiple algorithms on one or multiple scenes.
- The system is able to work with different configurations. This includes the time limit or types of visualization.
- The system must not encounter any exception, deadlock, or erroneous memory access.
- The system must correctly detect collisions or other errors made by the ego-vehicle.
- The routing graph must be connected, and a path exists from start to destination.

For each one of the scenes, we provide configuration files with cars that should be replaced with the ego-vehicle for the purpose of benchmarking the algorithm. Almost any vehicle can be substituted, except those either leaving the scene a few seconds after the start of the footage or entering shortly before the end. Even though the system is able to work with a benchmark created by substituting such a vehicle, the results of this benchmark would not hold much of an informational value. To demonstrate the functionality of the system, we chose a sample of vehicles with multiple combinations of starting and ending *lanelets*.

We tested the system on the following configurations.

- All of the scenes with one algorithm and *OpenCV*.
- A scene with one algorithm and *rViz*.
- A scene with multiple algorithms.
- A scene with a time limit.

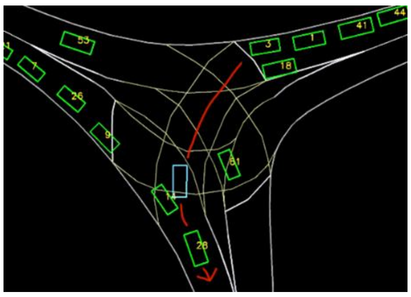
Both the simulator and the algorithm were linked with *Address Sanitizer*, as mentioned in Subsection 5.13.2, to check for erroneous memory accesses during runtime. The tool did not detect any error while running the benchmarks.

The resulting videos and evaluations were then manually inspected to ensure that the encountered errors are reported correctly. The outputs of the system can be found on the enclosed DVD in the folder *results* as described in Appendix.

6.6 Results

In this section, we discuss the behavior of the implemented algorithms during different traffic situations. The following examples are taken from the benchmark results of the algorithm described in Section 6.4.

In Figures 31 and 32, we see the blue ego-vehicle attempting to join a lane, but other vehicles are standing in its way. The algorithm correctly waits for a gap in the traffic to appear and joins the lane shortly after.



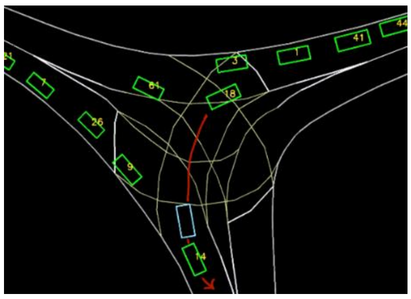
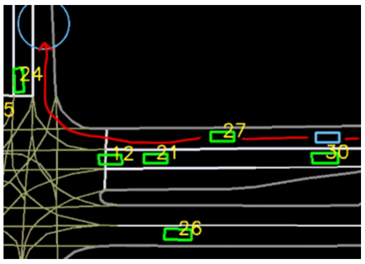
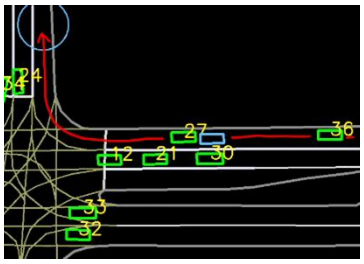


Figure 31 - The ego-vehicle waiting to join a lane

Figure 32 - The ego-vehicle has joined the lane

Figures 33 - 35 describe a scenario at a different intersection. In the beginning, the ego-vehicle enters the scene but is blocked by another car with an identifier of 27. The ego-vehicle correctly waits for the path to get clear and then continues to the destination marked by the circle.





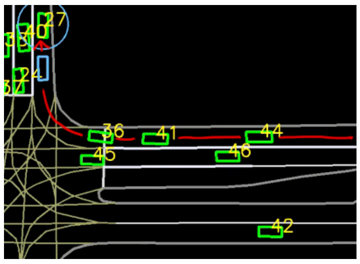
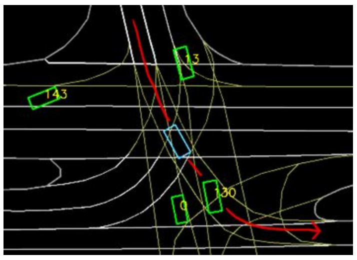


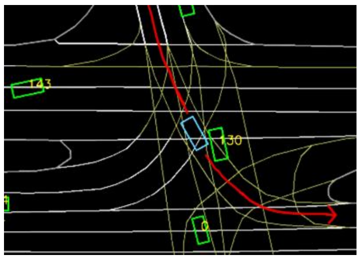
Figure 33 - The ego-vehicle has entered the scene

Figure 34 - The ego-vehicle waits at the intersection

Figure 35 - The ego-vehicle reaches the destination

Figures 36 - 38 illustrate a situation from another intersection. The ego-vehicle takes a left turn and meets another car traveling in the opposite direction. After the other car passes, the ego-vehicle continues along its planned trajectory.





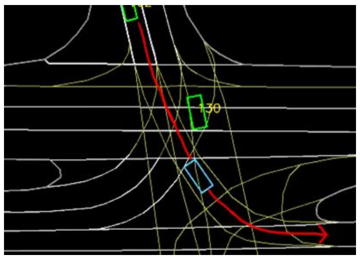


Figure 36 – The ego-vehicle takes a left-turn and meets another vehicle in the middle

Figure 37 - The ego-vehicle waits for the other vehicle to pass

Figure 38 - The ego-vehicle continues when the path becomes clear

An example where the algorithm does not solve the traffic situation correctly is shown in Figures 39 and 40. As a consequence of joining a traffic lane too soon, the ego-vehicle collides with another vehicle. The simulator detects this and colors the vehicle with red.

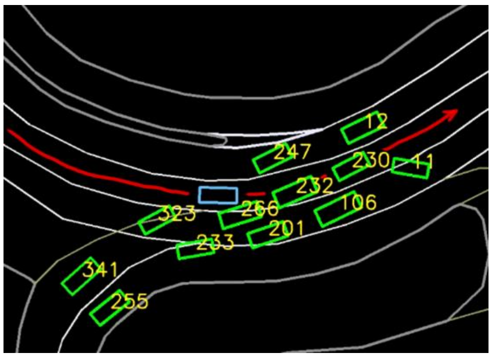


Figure 39 - The ego-vehicle joining a traffic lane

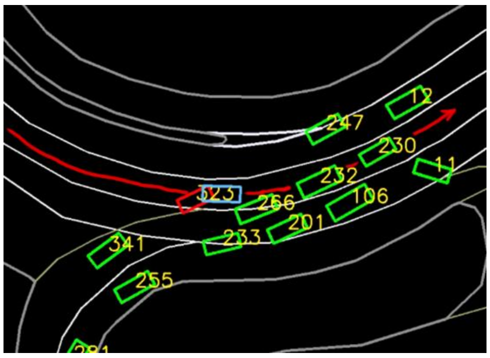


Figure 40 - The ego-vehicle collides with another vehicle as a result of joining too soon

We also provide examples for the algorithm stubs (described in Section 6.3). The *StraightAlgorithm* is predictable as it drives in a straight line and thus generates several collisions. This behavior is shown in Figures 37 and 38, where the ego-vehicle collides with multiple vehicles and a curbstone. The *EchoAlgorithm* does not substitute the original vehicle but only repeats the ground truth. It can be used to demonstrate that not every road user follows the traffic rules, as shown in Figure 43, where the ego-vehicle crosses a full line.

Straight Algorithm

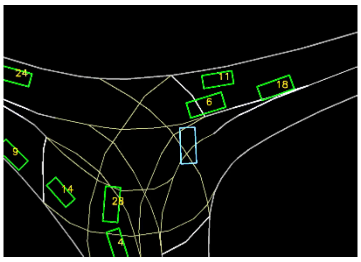


Figure 41 – The StraightAlgorithm drives in a straight line

Straight Algorithm

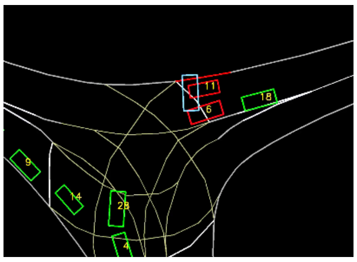


Figure 42 – StraightAlgorithm causes the ego-vehicle to collide with two other vehicles and a curbstone

Echo Algorithm

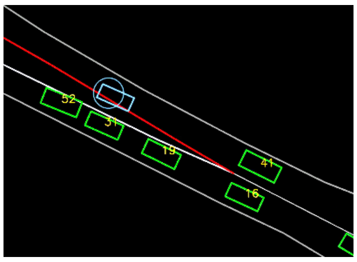


Figure 43 - The EchoAlgorithm causes the ego-vehicle to cross a full line

As stated earlier in Section 6.5, the complete outputs of the benchmarking are provided on the enclosed DVD in the folder *results*.

7 Future works

In the context of this thesis, we assembled a dataset containing vehicle trajectories extracted from six different videos. In the future, it might be necessary to extend this data set with data from other scenes. This could mean exploring the available pre-annotated datasets or searching the internet for more video footage, from which the data could be gathered. Alternatively, if a specific traffic situation is required, record the videos ourselves.

The simulation environment could also be extended with some functionalities. Currently, the system supports only one vehicle controlled by the algorithm at a time. Given that the system uses messages to communicate, it can be modified to be able to communicate with multiple algorithms at once. This would mean that we could evaluate a scenario with multiple agents, which are controlled by external algorithms. Such an approach would render the pipelining of the messages unusable, as the responses from one algorithm contain the information required by the rest of the algorithms. This would hinder the performance, but the system would work correctly nonetheless.

The simulator can also be extended with the detection of other kinds of errors. For example, we classify collisions as an error, but we could also declare that a near-miss is also an error. Similarly, we use a temporal constraint in the form of a time limit. The error is then detected if the algorithm does not respond to a message in the specified time. Similarly, we could specify the time in which the vehicle has to reach the destination. Failing to do so at the specified time would be classified as an error.

8 Conclusion

The main aim of this thesis was to design and implement an environment for benchmarking planning algorithms in *ROS*. To provide a complete solution, we split the project into four major parts.

In the first part, we focused on gathering the dataset for the benchmarks. Given that we wanted to provide information from real traffic, rather than one that is artificially generated, we searched for video footage, from which we could extract the trajectories of the vehicles.

To this point, we assembled a data set consisting of over six hundred unique vehicle trajectories from six different scenes. The data set is focused on roundabouts and junctions, as they are generally more complex than a simple highway, where all the vehicles move in the same direction and usually at a constant speed.

We also provide a *ROS* package devoted to manipulating the data. Using this package, one can remove trajectories or add artificial ones. One of the tools can also be used to convert data from the format used by detection frameworks and datasets to a format usable for benchmarking.

Furthermore, we discussed several methods of describing the scene itself. From the available formats, we chose *lanelet2*, due to the benefit of having an open-source editor and providing a library to handle the mapping data as a *ROS* package. The *lanelet2* framework was designed for the purpose of automated driving and gave us enough tools to describe even complex traffic scenarios. We also extended this format with one custom tag, which is used to project the trajectories from the data set onto the map.

The third part is dedicated to the design and implementation of the simulation environment. This package was also created to work as a *ROS* node. The simulator is able to run and benchmark planning algorithms in different scenarios. The scenarios are described using a configuration file, which serves as input for the system. Multiple benchmarks and algorithms can be listed in the configuration file to automate the process. The system then sequentially runs all of the algorithms on the specified data.

We use message passing to communicate with the algorithm. Therefore, the implementation of the algorithm is independent of the simulator. The simulator is responsible for running the algorithm and exposing the information about the scene using *ROS* messages. The simulator then collects the responses from the algorithms and writes any detected collisions or other errors into files in *JSON* or *CSV* format. It is also possible to visualize the whole scene using *OpenCV* or *rViz*.

Lastly, to verify the design of the whole environment consisting of the benchmarks and the simulator, we required an algorithm that can communicate with the simulator. However, later on, this was changed, so that we should provide our own example on how to use the routing library to plan the trajectory and how to communicate with the simulator. We benchmarked this algorithm using the whole dataset to demonstrate the simulator's behavior.

The gathered set of benchmarks, coupled with an environment to work with the data, creates a framework suitable for building new benchmarks for planning algorithms. The simulator is able to evaluate different algorithms using the designed benchmarks. All of the implemented software is compatible with *ROS* with an emphasis on using only libraries and tools, which are open-source or at least free to use.

The system is designed to be used by developers of planning algorithms for personal vehicles. It enables the developers to test the performance of their algorithms in a safe and simulated environment on several traffic scenarios from the real world. The output of the system is in a programmatically parsable format, allowing for automated running of simulations, and using the simulation results to adjust the parameters of the algorithm.

9 References

[1] M. A. K. Bahrin, M. F. Othman, N. H. N. Azli and M. F. Talib, "Industry 4.0: A review on industrial automation and robotic," *Jurnal Teknologi,* vol. 78, 2016.

[2] Open Robotics, "ROS.org | About ROS," [Online]. Available: https://www.ros.org/about-ros/. [Accessed 30 April 2021].

[3] Open Robotics, "ROS.org | Core Components," [Online]. Available: https://www.ros.org/core-components/. [Accessed 09 May 2021].

[4] M. Althoff, M. Koschi and S. Manzinger, "CommonRoad: Composable benchmarks for motion planning on roads," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017.

[5] Technische Universität München, "New Submission," [Online]. Available: https://commonroad.in.tum.de/new-submission. [Accessed 30 April 2021].

[6] GoodVision Ltd., "GoodVision Video Processing Options," [Online]. Available: http://help.goodvisionlive.com/en/articles/3304457-goodvision-video-processing-options. [Accessed 17 May 2021].

[7] NVIDIA, "NVIDIA DeepStream SDK Developer Guide - DeepStream 5.1 Release documentation," 26 February 2021. [Online]. Available: https://docs.nvidia.com/metropolis/deepstream/dev-guide/. [Accessed 30 April 2021].

[8] The Autoware Foundation, "Autoware.AI," 2020. [Online]. Available: https://www.autoware.ai/. [Accessed 9 May 2021].

[9] OpenStreetMap contributors, *https://www.openstreetmap.org,* 2017.

[10] OpenStreetMaps contributors, "GPS Navigation & Maps," 18 November 2020. [Online]. Available: https://wiki.openstreetmap.org/wiki/GPS_Navigation_%26_Maps. [Accessed 17 May 2021].

[11] ASAM e.V. ®, "ASAM OpenDRIVE," 2021. [Online]. Available: https://www.asam.net/standards/detail/opendrive/. [Accessed 16 April 2021].

[12] Navigation Data Standard (NDS), "Data for all: OpenDRIVE for driving simulators," 22 April 2021. [Online]. Available: https://nds-association.org/opendrive/. [Accessed 17 May 2021].

[13] F. Poggenhans, J.-H. Pauls, J. Janosovits, S. Orf, M. Naumann, F. Kuhnt and M. Mayr, "Lanelet2: A High-Definition Map Framework for the Future of Automated Driving," in *Proc IEEE Intell Trans Syst Conf.*, Hawaii, 2018.

[14] P. Zhu, L. Wen, D. Du, X. Bian, Q. Hu and H. Ling, "Vision Meets Drones: Past, Present and Future," *arXiv preprint arXiv:2001.06303,* 2020.

[15] I. Bozcan and E. Kayacan, *AU-AIR: A Multi-modal Unmanned Aerial Vehicle Dataset for Low Altitude Traffic Surveillance,* 2020.

[16] J. Bock, R. Krajewski, T. Moers, S. Runde, L. Vater and L. Eckstein, "The inD Dataset: A Drone Dataset of Naturalistic Road User Trajectories at German Intersections," 2019.

[17] Pexels, "Free stock videos," 2021. [Online]. Available: https://www.pexels.com/videos/. [Accessed 15 April 2021].

[18] C. Vondrick, D. Patterson and D. Ramanan, "Efficiently Scaling up Crowdsourced Video Annotation," *International Journal of Computer Vision,* pp. 1-21.

[19] P. Dendorfer, H. Rezatofighi, A. Milan, J. Shi, D. Cremers, I. Reid, S. Roth, K. Schindler and L. Leal-Taixé, "MOT20: A benchmark for multi object tracking in crowded scenes," *arXiv:2003.09003[cs],* 3 2020.

[20] Python Software Foundation, "Sunsetting Python2," [Online]. Available: https://www.python.org/doc/sunset-python-2/. [Accessed 15 April 2021].

[21] Q. Chen, Y. Zhao, S. Pan and Y. Wang, "Survey of the Influence of the Width of Urban Branch Roads on the Meeting of Two-Way Vehicle Flows," *PLOS ONE,* vol. 11, p. e0149188, 2 2016.

[22] ROS, "nav_msgs/Path Message," 13 January 2021. [Online]. Available: http://docs.ros.org/en/api/nav_msgs/html/msg/Path.html. [Accessed 15 April 2021].

[23] Python Software Foundation, "unittest - Unit Testing Framework," 2021. [Online]. Available: https://docs.python.org/3/library/unittest.html#module-unittest. [Accessed 15 April 2021].

[24] "Lanelet2," FZI Forschungszentrum Informatik, 14 July 2020. [Online]. Available: https://github.com/fzi-forschungszentrum-informatik/Lanelet2/blob/master/lanelet2_core/doc/LaneletPrimitives.md. [Accessed 16 April 2021].

[25] OpenStreetMap, "JOSM," 2021. [Online]. Available: https://josm.openstreetmap.de/. [Accessed 16 April 2021].

[26] FZI Forschungszentrum Informatik, "Lanelet2 Maps," 22 September 2020. [Online]. Available: https://github.com/fzi-forschungszentrum-informatik/Lanelet2/tree/master/lanelet2_maps. [Accessed 16 April 2021].

[27] Open Street Map creators, "JOSM/Plugins/PicLayer," 29 January 2020. [Online]. Available: https://wiki.openstreetmap.org/wiki/JOSM/Plugins/PicLayer. [Accessed 06 May 2021].

[28] Open Robotics, "ROS/Technical Overview - Ros Wiki," 15 June 2014. [Online]. Available: http://wiki.ros.org/ROS/Technical%20Overview#Topic_Transports. [Accessed 29 April 2021].

[29] G. Guennebaud, B. Jacob and others, *Eigen v3,* 2010.

[30] OpenCV Team, "OpenCV - OpenCV," 2021. [Online]. Available: https://opencv.org/. [Accessed 6 May 2021].

[31] B. Daves, D. Abrahams and R. Rivera, "Boost C++ Libraries," [Online]. Available: https://www.boost.org/. [Accessed 06 May 2021].

[32] M. Quigley, J. Faust, B. Gerkey, T. Straszheim and D. Thomas, "roscpp - ROS Wiki," 2 November 2015. [Online]. Available: http://wiki.ros.org/roscpp. [Accessed 06 May 2021].

[33] Open Robotics, "rviz - ROS," 16 May 2018. [Online]. Available: http://wiki.ros.org/rviz. [Accessed 9 May 2021].

[34] N. Lohmann, "JSON for Modern C++," 4 May 2021. [Online]. Available: https://github.com/nlohmann/json. [Accessed 6 May 2021].

[35] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 international symposium on memory management*, 2013.

[36] "visualization_msgs/Marker documentation," 15 January 2021. [Online]. Available: http://docs.ros.org/en/melodic/api/visualization_msgs/html/msg/Marker.html. [Accessed 9 May 2021].

[37] D. Vandevoorde, N. M. Josuttis and D. Gregor, C++ Templates: The Complete Guide (2nd Edition), 2nd ed., Addison-Wesley Professional, 2017.

[38] Google, "GoogleTest User's Guide | GoogleTest," [Online]. Available: https://google.github.io/googletest/. [Accessed 9 May 2021].

[39] Open Robotics, "catkin - ROS Wiki," 26 July 2017. [Online]. Available: http://wiki.ros.org/catkin. [Accessed 9 May 2021].

[40] J. Seward and N. Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision.," in *USENIX Annual Technical Conference, General Track*, 2005.

[41] The GNOME Project, "GLib Reference Manual - GNOME Developer Center," [Online]. Available: https://developer.gnome.org/glib/. [Accessed 9 May 2021].

[42] GNOME Project, "Valgrind detects possibly lost memory despite suppressions," 12 January 2021. [Online]. Available: https://gitlab.gnome.org/GNOME/glib/-/issues/2296. [Accessed 03 May 2021].

[43] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012.

[44] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review,* vol. 49, p. 71–79, 2015.

[45] Docker Inc., "Docker Engine overview | Docker Documentation," [Online]. Available: https://docs.docker.com/engine/. [Accessed 9 May 2021].

[46] E. G. Tsardoulias, A. Iliakopoulou, A. Kargakos and L. Petrou, "A review of global path planning methods for occupancy grid maps regardless of obstacle density," *Journal of Intelligent & Robotic Systems,* vol. 84, p. 829–858, 2016.

[47] K. Tong, Z. Ajanović and G. Stettinger, *Overview of Tools Supporting Planning for Automated Driving,* 2020.

[48] N. Kudarauskas, "Analysis of emergency braking of a vehicle," *Transport,* vol. 22, p. 154–159, 2007.

[49] "Zákon o silničním provozu č. 361/2000 Sb.," 1 January 2019. [Online]. Available: https://www.kurzy.cz/zakony/361-2000-zakon-o-silnicnim-provozu/cast-1/. [Accessed 9 May 2021].

[50] Centrum služeb pro silniční dopravu, "vzdalenost_mezi_vozidly," [Online]. Available: https://www.cspsd.cz/storage/files/vzdalenost_mezi_vozidly.pdf. [Accessed 9 May 2021].

10 Appendix

The contents of the enclosed DVD are described below.

- */results* – The outputs of the simulator for different configurations
- */memcheck* – The outputs of *Memcheck* with a brief description of each program execution
- */simulator_data* – Folder containing scene descriptions in *lanelet2* format and trajectories
- */src* – This folder contains the source codes for every implemented *ROS* package
- */src/hav_sim_utils* – *ROS* package dedicated to pre-processing the input data
- */src/sim_alg* – *ROS* package containing the planning algorithm example
- */src/hav_simulator* – *ROS* package containing the simulator
- */src/hav_simulator/cfg* – This folder contains several *JSON* configurations for benchmarking
- */src/hav_simulator/docker* – This folder contains a *Dockerfile* to build an image with a pre-installed environment
- */README*.md – Instructions on how to compile and run the system
- */plot.rviz* – A configuration for the scene in *rViz*
- */deepstream.md* – Instructions on how to use *DeepStream* to extract trajectories from a video