Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Christoffer Wilhelm Gran

HD-Maps in Autonomous Driving

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2019

NTNU
Norwegian University of Science and Technology

Christoffer Wilhelm Gran

HD-Maps in Autonomous Driving

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The car industry is heading in an autonomous direction, and many of the world's biggest technology companies are investing in the area. In 2018, The Norwegian University of Science and Technology (NTNU) started it's own project, NTNU Autonomous Perception (NAP). This project's goal is to continously develop and research state-of-the-art models for autonomous driving that are robust to a Nordic environment.

Other projects have already come far in the development of autonomous cars, and during the fall of 2018 it was decided that the NAP project should use the Apollo framework for autonomous driving to quickly reach the state-of-the-art. The Apollo framework is an open source framework developed by chinese company Baidu. In order to use this framework it is necessary to have an HD-map of the area that the car is to drive in. An HD-map is a detailed map made to contain all information a car may require about an area. This includes detailed and accurate information about the road network like number of lanes, the width of each lane and signs along the roads.

There is several ways of obtaining such maps, and during the early stages of this project it was decided that the best way at this time is developing our own method for map generation. This task, with a focus on making maps compatible with the Apollo framework, is the subject of this thesis. As a basis for the maps this project uses data retrieved from the open source map service OpenStreetMap.

The project was successfull in generating maps compatible with Apollo, but with reduced functionality and accuracy. The roads are represented correctly, and they are connected as specified in the specification. The car of the project was being setup simultaneously to this project, so there was limited time for testing the maps in a real situation. The accuracy of the generated maps are relatively low, and should be a focus for future projects. This report hopefully gives useful insight into the Apollo map format that will make map generation easier in the future.

Sammendrag

Bilindustrien går i en autonom retning, og mange av de største teknologifirmaene i verden investerer ressurser i utvikling innenfor dette området. Norsk teknisk-vitenskapelige universitet (NTNU) startet i 2018 sitt eget prosjekt innenfor autonom bilkjøring, NTNU Autonomous Perception (NAP). Dette prosjektet har som mål å utvikle og undersøke løsninger for autonome biler i nordiske forhold.

Andre prosjekter har kommet langt i utviklingen av autonome biler, og for å nå et tilsvarende nivå raskt ble det høsten 2018 besluttet at NAP-prosjektet skal ta i bruk et rammeverk for autonom kjøring, Apollo. Dette rammeverket er laget av kinesiske Baidu, og har åpen kildekode. For å ta i bruk dette rammeverket er det nødvendig å ha et såkalt HD-kart over området bilen skal kjøre i. Et HD-kart er et detaljert kart laget for å inneholde informasjon en bil kan trenge om et område. Dette inkluderer detaljert og nøyaktig informasjon om veinettet som antall filer, filenes grenser og skilt langs veien.

Det finnes flere ulike måter å få tak i HD-kart på, men det ble i starten av prosjektet besluttet at det mest hensiktsmessige ville være å utvikle en egen løsning for generering av HD-kart. Dette prosjektet tar for seg denne oppgaven, med fokus på å lage kart som er kompatible med det utvalgte rammeverket Apollo. Som en basis for kart-genereringen ble data hentet fra den åpne kartløsningen OpenStreetMap (OSM). Dette ble gjort for å komme raskt i gang med utviklingen.

Prosjektet klarte å generere kart som er kompatible med Apollo, men med redusert funksjonalitet og nøyaktighet. Veiene er representert på riktig måte, og de er koblet sammen etter spesifikasjonen. Prosjektets bil ble satt opp parallelt med dette prosjektet, noe som førte til at det ble lite tid til å teste kartet i en reel situasjon. Nøyaktigheten på kartet er relativt lav, og bør være et fokusområde på fremtidige prosjekter. Denne rapporten gir forhåpentligvis nyttig innsikt i Apollos kartformat som vil gjøre det enklere å generere mer nøyaktige kart fremover.

Preface

This thesis is written for the Department of Computer Science (IDI) at Norwegian University of Science and Technology (NTNU) as a result of the project in the course "TDT4900 Computer Science, Master's Thesis".

I would like to thank my supervisor Frank Lindseth for continous support throughout the semester. I would also like to thank Florent Revest for ideas for the project and great help during the project.

Christoffer Wilhelm Gran

Contents

Abstract . . . . **i**
Sammendrag . . . . **iii**
Preface . . . . **v**
Contents . . . . **vii**
List of Figures . . . . **ix**
List of Tables . . . . **x**
Listings . . . . **xi**
1 Introduction **1**
1.1 Background and motivation 1
1.2 Research questions 2
1.3 Contributions 2
1.4 Theses outline 2
2 Background **3**
2.1 HD-maps 3
2.1.1 Existing HD-map solutions 3
2.1.2 Techniques for HD-map generation 5
2.1.3 HD-maps from sensor data 5
2.1.4 HD-maps from available map data 6
2.2 Apollo 8
2.3 OpenDRIVE 9
2.3.1 Roads 9
2.3.2 Lanes 10
2.3.3 Objects and signals 10
2.4 Apollo OpenDRIVE 11
2.4.1 Geometry 11
2.4.2 Roads 11
2.4.3 Road-record 12
2.4.4 Lanes 12
2.4.5 Road Objects and Signals 14
2.4.6 Junctions 14
2.4.7 Differences from OpenDRIVE 15
2.5 Global Navigation Satellite System 17
2.5.1 World Geodetic System 17
2.5.2 Universal Transverse Mercator coordinate system 17

3 Methodology . . . **18**
3.1 Data source . . . 18
3.1.1 Data for road generation . . . 18
3.1.2 Available existing map data . . . 18
3.1.3 Recording sensor data . . . 19
3.1.4 Data source conclusion . . . 19
3.2 Map generation . . . 20
3.2.1 Information parsing . . . 20
3.2.2 Road generation . . . 21
3.2.3 Road connections and junctions . . . 25
3.3 Apollo OpenDRIVE . . . 26
3.3.1 Lanes and center lines . . . 26
3.3.2 Border types . . . 26
3.3.3 Sample Associations . . . 26
3.4 Pipe-line for map generation . . . 26
3.4.1 Data gathering . . . 27
3.4.2 Map generation . . . 27
3.4.3 Map conversion . . . 28
4 Results . . . **30**
4.1 Map results . . . 30
4.1.1 Road lanes and width . . . 30
5 Discussion . . . **35**
5.1 Generated maps . . . 35
5.2 Lanes . . . 35
5.3 Junctions . . . 36
5.4 Objects and Signals . . . 36
5.5 Performance . . . 37
5.6 Base data . . . 37
5.7 Reflection . . . 37
6 Conclusion . . . **39**
6.1 Future work . . . 39
Bibliography . . . **40**
A Appendices . . . **43**
A.1 road.py . . . 43
A.2 osm2od.py . . . 44

List of Figures

1 A screenshot of the *Vegkart* website. 7
2 The Apollo architecture. The image is made by Apollo[1]. 8
3 An illustration of different types of geometries in OpenDRIVE. The image is taken from the OpenDRIVE specification 10
4 The UTM zones. Note how some zones differ in size, like the one for southern Norway, 32V. The image is retrieved from Wikipedia. [2] 17
5 A visualization of a road segment. The red lines represent the coordinate data Apollo OpenDRIVE would require to store this segment. 19
6 An illustration of the issue with the first vector based approach. 22
7 An illustration of the scaled vector length to preserve lane width. 23
8 An illustration of the issue with meeting single lane roads that are plotted in different directions. The dotted lines are those generated, the solid lines are the data from OSM. 24
9 How the roads appear when the data from OSM is used as the center line. 24
10 An illustration of connecting roads within junctions. The image is taken from the OpenDRIVE specification [3] 25
11 A screenshot of the OSM website showing how to download XML files. 27
12 Two plots of the same map 30
13 The same roads, but the map is using a config-file telling one of the roads to be wider. 31
14 The same roads, but this time, one of the roads have been configured to have two lanes. 31
15 A plot of a junction where two of the roads meet at a very sharp angle. 32
16 A screenshot from the Dreamview interface showing the simple version of the map. . 33
17 A screenshot showing an overview of a generated map in the Dreamviewer GUI. . . . 33
18 A closer look at the simplified map. 34

List of Tables

1 An overview of companies in the mapping field. 4
2 The structure of the *geometry* records. 11
3 The structure of the *center* record within lanes. 13
4 The structure of the *left* and *right* records within lanes. Some *Overlap* related records have been left out as they are optional and not used in this thesis. 13
5 The structure of the *junction* record. 15

Listings

3.1 Example code for reading data from an OSM XML file. 20
3.2 An example of a node from an OSM XML file. 20
A.1 road.py 43
A.2 osm2od.py 44

1 Introduction

1.1 Background and motivation

The future on the road seem to be going in an autonomous direction. Many of the biggest technology and automotive companies are putting much resources into the field of autonomous vehicles and artificial intelligence.

NTNU has created its own project on autonomous vehicles, NTNU Autonomous Perception (NAP)[4]. This project's goal is to continously develop and research models for autonomous driving that are robust to a Nordic environment and aims to get to the state of the art quickly. To achieve this, the project will be making use of the open source framework for autonomous driving *Apollo*. In addition to this, students are working on different problems related to different tasks separately from Apollo. The project is divided into several teams, each with different responsibilities:

Team 1 Sensors, compute and control
Team 2 Mapping and localization
Team 3 Perception (and data)
Team 4 Planning
Team 5 Simulation (and validation)
Team 6 Safety and security
Team 7 Connected vehicles
Team 8 Shared vehicles
Team 9 Privacy and ethics

This thesis is written within *Team 2: Mapping and localization*, and will be focusing on *HD-maps*. Normal maps are made for humans, and thus feature information relevant to them. HD-maps are different, firstly in that they are made for computers, not humans. They are made to give extra context to the location of a vehicle, and potentially also for assisting in finding a vehicles exact location in the first place. An HD-map can often include information about lanes, speed limits, signs, lights, junctions, and even objects around the road.

Apollo uses HD-maps actively. For instance, the map is used to determine whether a traffic light is near the vehicle, enabling the traffic light module. When a traffic light is not present according to the HD-map, the perception module will not be checking for traffic lights.

This means that to use Apollo, we are in need of an HD-map of the area we will be driving in. Apollo has not made publicly available how they generate maps, but they do provide a service for map generation[5]. This service is based on sensor data, and the process is also partly manual, improving the accuracy and correctness of the resulting maps. To make use of this service, Apollo requires sensor data of the relevanta area recorded with a so-called *data collection vehicle*, a vehicle

setup with sensors for data recording. The NAP lab's vehicle was not ready for data collection during this semester, so we were required to make a different approach. Another issue is that Apollos way of generating maps is dependant on road border markings. Many roads, specially in cities, do not have such markings. This leads us to the purpose of this thesis. Our vehicle is in need of an HD-map, and we do not have a vehicle set up for data recording. The only viable way the project can obtain an HD-map is making one.

1.2 Research questions

The main goal of this master's thesis is to produce an HD-map for the NAP project. Generating a HD-map involves two main sub-objectives, data gathering and map generation. This leads to the following research questions:

RQ1: How can we obtain an HD-map for the NAP project?

- **RQ1.1:** Are there any available open source solutions for HD-map generation?
- **RQ1.2:** Can openly available data be used as a basis for making HD-maps?
- **RQ1.3:** Is it possible to make a map based on open data compatible with the Apollo framework?

1.3 Contributions

This thesis aims to explore HD-maps with a special focus on how they are made. In order to make HD-maps it is necessary to understand how they are represented, and how the data is gathered. This thesis will aim to get an understanding of the format Apollo uses to represent HD-maps as well as an overview of how data is gathered for HD-map generation.

1.4 Theses outline

Section	Description
Section 1: Introduction	Explains what this report is aiming to do, and why.
Section 2: Background	Contains information about the different technologies that were used to solve the problem. Also describes some existing solutions.
Section 3: Methodology	Describes what was done, what was implemented.
Section 4: Results	Contains the results from the implementations.
Section 5: Discussion	Discusses the results presented in section 4, and what could possibly have been done differently to achieve better results
Section 6: Conclusion	The conclusion and discussion about future work with HD-maps.

2 Background

2.1 HD-maps

HD-maps are maps made for computers, and not for humans like maps normally are.[6] An autonomous vehicle obviously need a map for planning and knowing where it should drive, but maps can also have more advanced uses. It is important to have information about how many lanes the road has, where the car has to be positioned to get to its destination.

Regular maps are usually made to be useful for humans, but the same maps does not necessarily make good maps for computers. HD-maps are maps that focus on different things than regular maps. High accuracy is more important, and HD-maps often have centimeter-accuracy. In addition to the high accuracy requirement, HD-maps often feature more and different details in comparison to regular maps. This can include information about speed limits, lanes, as well as descriptions of the buildings and objects around the roads.

HD-maps may strictly not be needed for autonomous driving, but they make several tasks easier. An example is how *Apollo* deals with traffic lights. Unless their HD-map finds a nearby traffic light, the module reading traffic lights is turned off. Apollo queries the HD-map to recieve information about the world around the vehicle. Uses like this is the most obvious use case, providing information to an autonomous driving agent that would be difficult for the agent to obtain from raw sensor data on the fly. Another possible use is localization, by comparing input from a vehicles sensors to the HD-map the car can get improved accuracy on its current position. This can also work as a backup in case the car would lose its GPS signal. This would require the map both to be very accurate, and also would need the map contents to include buildings and other landmarks.

2.1.1 Existing HD-map solutions

Maps for driving is not a new thing, and some companies involved in the mapping field have are starting to realise that autonomous driving is the future. In addition to already established companies like TomTom, there are a number of startups aiming to create HD-maps or provide some map-related product. Some of these companies the are presented in table 1 below.

Organization	HD-map	Open source	Service
comma.ai[7]	Y	Partly	Main product is *OpenPilot* for lane following and adaptive cruise control. Also have made a HD-map[8], although difficult to actually find anywhere. comma.ai's solutions are mostly open source.
Mapillary[9]	N	N	Enhance maps with street level images. Detect map features from street level images. Not necessarily for HD-maps.
DeepMap[10]	Y	N	Accurate and maintainable HD-maps for autonomous driving. Not much further information available.
TomTom[11]	Y	N	HD-map for autonomous driving. Additionally offer *RoadDNA*, for localization purposes through comparing input signals with RoadDNA.
lvl5[12]	Y	N	Aims to make HD-maps from cheap sensors, equivalent to those of a smart phone.
HERE[13]	Y	N	HERE claims to provide a continously updated HD-map for autonomous driving. They specially point out the important of accurate lane border data, and the importance of updated maps.
Civil Maps[14]	Y	N	Provides a scalable edge-based HD mapping and localization platform.
Carmera[15]	Y	N	One of CARMERA's products is a HD map for autonomous driving. There is not that much information available about their map service. CARMERA has teamed up with Toyota Research Institute-Advanced Development, Inc. (TRI-AD) to cooperate on a HD map solution[16].

Table 1: An overview of companies in the mapping field.

As clearly shown in the table, few of these companies have open source solutions. Many of their products are also difficult to get a good overview of, and there is often little available information. It can be difficult to get a clear idea of the quality and overall status of their products. In addition to this they mostly don't specify which format their map supports, possibly making them incompatible with the NAP lab's software stack.

Other available solutions

In addition to companies producing HD-maps there are a few existing ways of generating HD-maps compatible with Apollo. This section will discuss these briefly.

Apollo scripts for map generation

Even though Apollo does not explain in detail how they generate HD-maps, they have a few available undocumented scripts for map generation. One of these is called *map_gen_single_lane.py* and simply takes a list of points and makes it into a road in an Apollo OpenDRIVE map. This tool does, however, simply create one road, not a network of roads. Some other more advanced tools are also available, and these can make maps from *Mobileye* data[17].

LGSVL Simulator

The LGSVL Simulator is a simulator created by LG for testing and development of autonomous driving systems. It is fully integrated with Apollo and AutoWare[18]. This system allows exporting existing Unity scenes to maps following the Apollo OpenDRIVE standard[19].

2.1.2 Techniques for HD-map generation

Making an HD-map can be looked upon as consisting of two major parts, data gathering and map generation. The data gathering process involves obtaining all the relevant data of a certain area, all data the vehicle can possibly make use of. There are mainly two ways of approaching this problem. The first approach is based on using a special vehicle configured for data gathering to collect sensor data, and then use this collected data as a base for the map generation. The second approach is based on already existing, processed map data. The map generation method relies heavily on what kinds of data it is to be based on, and also what kind of output it is supposed to generate. The following sections will firstly take a closer look at the two approaches for data gathering, and then describe the output formats the NAP lab's software stack requires.

2.1.3 HD-maps from sensor data

Generating maps from sensor data is probably the most used method for map generation. This method involves using a *data collection vehicle* for sensor data collection, and then basing the map generation process on this data. Such vehicles typically have the following sensors:

GPS/IMU The GPS/IMU unit will record coordinates that will make up the basis for the roads. Having a high precision GPS is important to achieve highly accurate maps.

Camera There can be one camera, or several cameras facing different directions. This is for detecting and locating objects around the road, and road features like lanes.

LiDAR Used with the cameras to localise objects and details around the road. Some companies try to generate maps without using a LiDAR to keep the sensor package cost down as LiDARs are very expensive.

Road generation

For road generation, the most important sensor is the GPS/IMU-unit. In all maps, roads are simply arrays of points, and HD-maps are not different. The difference is rather all the extra information that HD-maps contain, and the overall accuracy of all information within the map. A stream of coordinate points recorded from the vehicle can be used as a basis for the *reference line* for a road. Other information about the road, such as the road and lane borders, will be based on this line. Additional LiDAR and Camera information can provide information about the lanes, like how wide the lanes are and where the borders should be and what color and type the borders are. This combination of sensor data can result in very accurate maps.

2.1.4 HD-maps from available map data

Available sources

HD-maps can also be made by combining and modifying existing data sources. By doing this it is possible to make a map representation of an area without collecting sensor data. As described more thoroughly below, there exist a number of good data sources for road topology as well as sources for speed limits, traffic signs and other road elements. One problem with this approach, however, is the accuracy. Most of the available sources are mare for humans, with a different accuracy-requirement than cars sometimes have. If the map is to be used for localisation, a centimeter accurate map might be required. If the only requirement is knowing some metadata about the roads as well as some rough model of the topology of the road, this can be a good solution as it does not require sensor data and is simple to assemble.

OpenStreetMap

Founded in 2004, OpenStreetMap is a community driven map service that provides open data freely useable for any purpose. The map is created by volunteers using GPS trackers, cameras and similar tools[20]. OpenStreetMap has a Wiki explaining details about the map and the format the map should follow[21]. Some of the core elements of this format is explained below:

Data format OpenStreetMap data is available in XML-format, and can easily be downloaded from their website, or through third party APIs. In this section, some of the most useful concepts of the OSM data format will be explained.

Node A *Node* represent a single point in space, defined by latitude/longitude coordinates. Each node can also contain additional information, such as who added it, and when.

Way A *Way* is as described in the OpenStreetMap wiki an *ordered list of nodes*. Ways are usually categorized into different types depending on what kind of road it is to represent. They also often contain further information about the road, like the street name. Each way can have different tags, and no way is guaranteed to have any information.

Intersections Intersections in OpenStreetMap are represented in a very simple manner. Whenever two or more roads intersect, they will all share one common node. In contrast to OpenDRIVE, there is no specific tags for intersections in OpenStreetMap.

Relation *Relations* are groups of elements, and can contain nodes, ways, areas and other relations. They are used to give logical or geographical relations between elements. Relations can have different types, with the most common being *Multipolygon* and *Route*. An example of a reation is bus routes, which can be represented as a relation with a *Route* tag

Area Area is not an element in OSM like the others, but it is still a term in frequent use. It can be defined either by the use of closed ways, or by using multipolygon relations. The idea is to represent some closed area.

Nasjonal vegdatabank

The national road data bank is a database containing information about the norwegian road net[22]. In addition to information about the road topology and geometry, it includes statistics about traffic accidents and messages and other information that would otherwise not be easily obtainable.

An online map displaying the database contents is available, a screenshot of this is shown in figure. It is also possible to download data from an online API. The data is accessible under the *Norwegian Licence for Open Government Data*, and it can mostly be used by anyone for any purpose[23]. One drawback with this API is that the topology of the road network is not available. This means that the topology needs to be built by retrieving road network links from the API and connecting the roads manually.

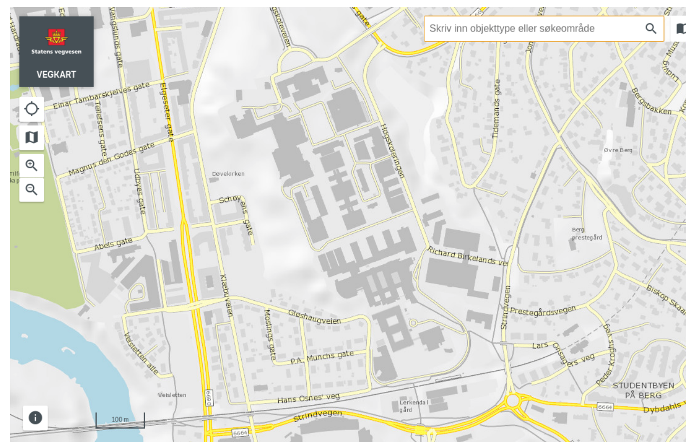


Figure 1: A screenshot of the *Vegkart* website.

2.2 Apollo

Apollo[24] is an open source architecture for autonomous driving developed by chinese company Baidu. Apollo has been in development since 2017 and the newest version as of june 2019 is version 3.5. The first version of Apollo focus on simply driving a car between two coordinates. Gradually the versions extend on this by adding more sensors and more functionality. Version 1.5 support lane following, and the newest version claim to be able to navigate through complex driving scenarios such as residential and downtown areas[25]. Because of a sensor requirement issue[26], the NAP lab is currently built on version 3.0 of Apollo.

Apollo is made up of several modules that solves different parts of the automation process. The architecture of version 3.0 is shown in figure 2. The modules communicate via messages. Until version 3.0 Apollo was built on a modified version of the Robot Operating System (ROS), but from version 3.5 this was changed to their own solution, *Cyber*. *ROS* allows so-called *nodes* to publish messages to *topics* that other *nodes* can subscribe to.

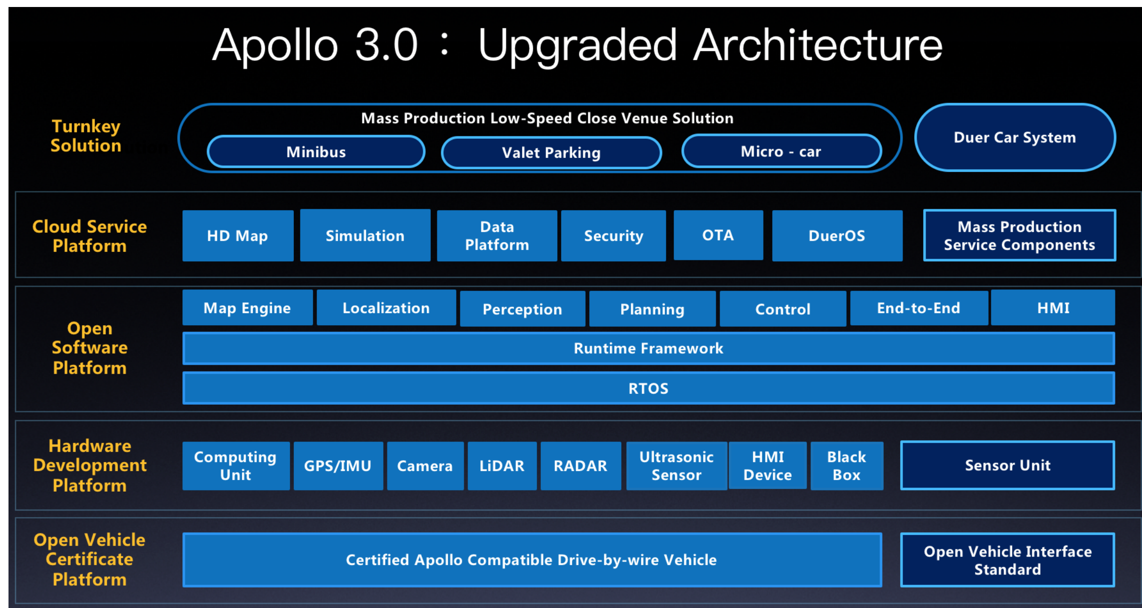


Figure 2: The Apollo architecture. The image is made by Apollo[1].

A part of the architecture shown in figure 2 is the *Cloud Service Platform*. This is a collection of services that Apollo provide as a cloud service, and these are not open source. The HD-map generation feature they provide is a part of the cloud services.

2.3 OpenDRIVE

OpenDRIVE is an open file format for describing road networks. The standard was initially released in 2005, and the latest specification release was published in 2019 [3]. In 2018 OpenDRIVE was transferred from *Vires Simulationstechnologie* to *Association for Standardization of Automation and Measuring Systems (ASAM)*. Following this transfer, *ASAM* expect to release the next version of the standard in 2019.

The OpenDRIVE standard features a number of data models to describe the different elements of a road environment accurately. As this thesis is based on a modified version of the OpenDRIVE standard, the standard itself is only described briefly in this section, with the modified version being explained in more detail in section 2.4.

2.3.1 Roads

One of the main components of OpenDRIVE is the road. Roads in OpenDRIVE consist of a few different parts, the most important being the reference line, the lanes, and the road linkage. The *reference line* defines the main geometry of the road, the shape of the road. The lanes represent the different lanes of the road, this includes how wide the lanes are, and how many of them there are. The road linkage connects the roads to eachother, advanced intersections use a special *junction* record. In this section these concepts will be described a bit closer.

Geometry

Many maps define roads as a set of nodes, where the nodes mark points along a road. This is the case for *OpenStreetMap*. OpenDRIVE, however, defines a road's reference line through what they have called *geometrics*. There is four types of geometric elements:

- Straight lines
- Spirals
- Arcs
- Cubic polynomials

Depending on the type of geometry, it includes different parameters to accurately represent the real world road. The main difference between the type is the curvature. For lines, the curvature is constant zero, for spirals the curvature changes linearly. Arcs have a constant non-zero curvature. A sequence of geometry records define each road's reference line. Some of the different geometry types are illustrated in figure 3.

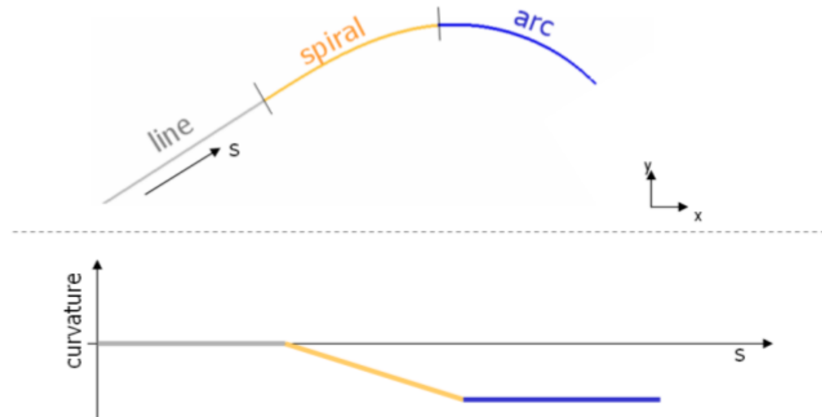


Figure 3: An illustration of different types of geometries in OpenDRIVE. The image is taken from the OpenDRIVE specification

2.3.2 Lanes

Within each road there is a number of lanes. These all have attributes containing metadata about the lane. Most importantly they describe the width and the borders of each lane. The width of the lanes can simply be given in a number in meters, or it can be described in more detail through a polynomial function of third order. In addition to the position of the borders, the lane records support storing information about the road markings, lane material and lane access.

2.3.3 Objects and signals

In real roads, information is broadcast to the drivers mainly though road signs, lights, and markings. OpenDRIVE provides a way of representing this information in the map. Each road can have a number of objects related to it. These objects are defined by a number of attributes that describe the object's location, shape and other relevant information. Signals are similar to objects, they describe signs and signals. Signals are *dynamic* signs, signs that can change state. Both objects and signals are defined to be a type which defines what kind of object or signal the instance represents.

2.4 Apollo OpenDRIVE

For their HD-map solution, Apollo uses a modified version of the OpenDRIVE standard. While the overall structure is similar, Apollo's modified version is very different from the standard as they have changed several fundamental details, for instance how roads are represented. The following section will take a closer look at the Apollo specification of OpenDRIVE[27], and how it differs from regular OpenDRIVE. It should be noted that the Apollo OpenDRIVE specification is only obtainable by requesting it from Apollo directly[28].

2.4.1 Geometry

One of the fundamental records in the OpenDRIVE standard is the *geometry* record. While its function is the same as in the standard, the Apollo specification of OpenDRIVE requires the *geometry* record to follow a different structure. It is also more actively used in Apollo OpenDRIVE, being used for defining lane borders and center lines as well as road reference lines. Where in regular OpenDRIVE geometries are defined by *geometries*, in Apollo OpenDRIVE, they should simply be defined by a set of points.

Tag	Parent	Instances	Attributes
geometry		1	sOffset, x, y, z, length
pointSet	geometry	1	none
point	pointSet	2+	x, y, z

Table 2: The structure of the *geometry* records.

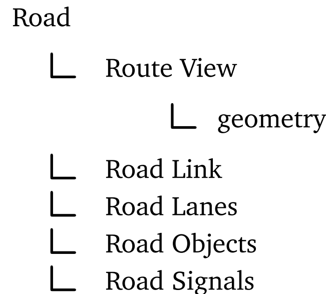
The *geometry* records within the Apollo OpenDRIVE standard generally follow the same format. The only exception is when used for defining the boundaries of a lane. In this situation the *geometry* tag does not use any attributes, but the rest of the structure is identical.

2.4.2 Roads

The way roads are represented is one of the big differences between regular OpenDrive and the Apollo specification. As described earlier, regular OpenDrive uses *geometries* to represent the reference line of the roads. While the tag uses the same name, geometry, the Apollo-specification uses points rather than actual geometries to represent the roads. This geometry tag is used frequently in different parts of files following the Apollo OpenDRIVE specification, and it is not only used for describing road center lines. This also ties into how lanes are represented in Apollo OpenDRIVE, which is more detailed than in regular OpenDRIVE. In Apollo OpenDRIVE, lanes are required to specify a number of boundaries and borders with separate point sets from the road's reference line. This allows for more detailed lane shapes, but also requires more data. In regular OpenDRIVE, while there is support for describing lane widths with the same type of geometries as used for describing road reference lines, there is also possible to simply specify lane width. The lane does then not contain further information about where the actual borders lie. This is not possible in Apollo OpenDRIVE where it is required to specify the borders.

2.4.3 Road-record

The road-record represents a road within the map. The structure of road records are explained below.



Route View

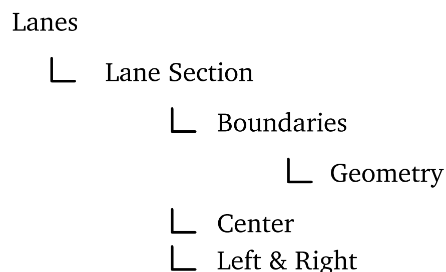
The *Route View*-record is poorly explained in the Apollo OpenDRIVE specification, as it is not explained at all apart from the actual XML format. It is also not a required tag. It is easy to get the impression that this is the record specifying a roads geometry, but that is seemingly not the case. Its format is simple, it has no attributes, and consists of a basic geometry child element with following pointSet and point children as explained in section 2.4.1.

Road Link

Road links describe a road's relation to other nearby roads, namely the previous, the next and parallel roads. This follows the same format as the regular OpenDRIVE standard, and is beter explained in that specification.

2.4.4 Lanes

The lanes of a road are specified within the *lanes*-record. The biggest difference between lanes in Apollo OpenDRIVE and the OpenDRIVE standard is that the borders have to be defined by coordinates, and that it is required to specify a center line. The overall structure of the *lane* record in Apollo OpenDRIVE is shown below:



Lane Sections

Lane sections represent different parts of the road. In the Apollo OpenDRIVE specification these sections are specified differently from the standard. Instead of simply specifying the start coordi-

nate of the section as in the OpenDRIVE standard, it is necessary to define the area explicitly by specifying left and right boundaries with geometries. Note, however, that the geometries used to define borders are different from other geometries. These geometry tags do not have any attributes, but the structure otherwise is identical to other instances of the geometry record.

Center, Left and Right

Within each lane section, it is necessary to define the lanes. The first tag that is required is a *left*, *center* or *right* tag. While these are presented as equivalent in the Apollo OpenDRIVE specification, this does not appear to be the case when looking at example files. The *center* tag is different from the other two, and represents the reference line of the road. The structure of the *center* records are simpler than the scructure of the *right* and *left* records. This is, however, not explained in the specification. The differences are shown below, also bear in mind that the attributes necessary within the *center* tag's children are different from the *left* and *right* tag's children's attributes. The tables below show the differences between the *center* tag and the *left* and *right* tags.

center			
Tag	Parent	Instances	Attributes
lane	center	1	id, uid, type
border	lane	1	virtual
borderType	border	1+	sOffset, type, color
geometry	border	1+	See section 2.4.1

Table 3: The structure of the *center* record within lanes.

left/right			
Tag	Parent	Instances	Attributes
lane	left/ right	1	id, uid, type, direction, turnType
centerLine	lane	1	None
geometry	centerLine	1+	See section 2.4.1
border	lane	1	virtual
borderType	border	1+	sOffset, type, color
speed	lane	0...1	min, max
geometry	border	1+	See section 2.4.1
sampleAssociates	lane	1	None
sampleAssociate	sampleAssociates	1+	sOffset, leftWidth, rightWidth
roadSampleAssociations	lane	1	None
roadSampleAssociation	roadSampleAssociations	1+	sOffset, leftWidth, rightWidth

Table 4: The structure of the *left* and *right* records within lanes. Some *Overlap* related records have been left out as they are optional and not used in this thesis.

The *left* and *right*-lanes defines the lanes of the road. These records need to specify coordinates for the border on the left or right side respectively. For the border it is also required to specify type

and color. In addition to this, the Apollo OpenDRIVE specification also require coordinates for the center line of the lane. Any speed limit information available should also be stored within the lane, namely in the *speed* record.

Sample Associations

There are two records within lanes in the Apollo OpenDRIVE specification that are not present in the OpenDRIVE standard, the *Sample Association* records. There are two versions of the *Sample Association* record, *Road Sample Associations* and *Sample Associations*. These records are not explained much in the specification, but they specify the distance from the center of each lane to the boundaries of the roads and lane respectively. According to the specification these records are optional, but some of the scripts for working with maps provided by Apollo seem to require the records. The specification is also inconsistent with these scrips when it comes to the record's names. Table 4 describes the actual format.

2.4.5 Road Objects and Signals

Road objects

One of the main parts of a map is road objects. This is all kinds of objects that can have a relation and an importance to the roads in a map. Such objects are represented with the *object* record in Apollo OpenDRIVE. The *object* record in Apollo OpenDRIVE is very similar to the one in regular OpenDRIVE, but it's parameters varies more based on which type of object is being referenced. The overall structure usually has some metadata in the *object* attributes, including *id* and *type*. The objects then also defines an outline the same way as it's defined for junctions, by listing the corner points. In addition to this, some points also have an associated geometry represented in the same way, by defining the corners. One difference between the Apollo specification of OpenDRIVE and the OpenDRIVE standard is that these points should be defined in WGS84 coordinates in Apollo OpenDRIVE, but should be defined relative to the road in the OpenDRIVE standard.

Road signals

In Apollo OpenDRIVE, the *Road Signal* record represents traffic lights, stop signs and yield signs. Similarly to the road objects, the road signals are very similar in Apollo OpenDRIVE and in the OpenDRIVE standard, but Apollo OpenDRIVE only implements a small subset of the signals that the OpenDRIVE standard does. This also means that Apollo OpenDRIVE generally uses fewer attributes for the road signals than the OpenDRIVE standard does. The different signals refer to road objects for describing the line on the ground related to the signals. This line represents where the car has to stop. This means that each *signal* record will refer to some *roadmark* object to describe the location of this line.

2.4.6 Junctions

When roads meet, there needs to be some connection in the map representing the intersection. When it is simply two roads meeting this can be done with the roads own *link* records. However, it is often the case that more than two roads are meeting, making a more advanced intersection

structure necessary. In Apollo OpenDRIVE this is solved using the *junction* record. The structure of the *junction* record is shown in table 5.

junction			
Tag	Parent	Instances	Attributes
junction	OpenDRIVE	0+	id
outline	junction	1	None
cornerGlobal	outline	3+	x, y, z
connection	junction,	1+	id, incomingRoad, connectingRoad
laneLink	connection	1+	from, to
objectOverlapGroup	junction	0+	None
objectReference	objectOverlapGroup	1+	id

Table 5: The structure of the *junction* record.

Outline

The *outline* record describes the outline of the junction, simply the area that the junction spans. This area is defined by plotting it's corners using the *cornerGlobal* record.

Connection

In order to connect roads to eachother, the *connection* record is used. This simply states which incoming road to connect to which *connecting road*. A *connecting road* is a road connecting two incoming roads, it describes the path between two roads. The *connection* record also incldues a subrecord, *laneLink*, that describes which lanes in the incoming road links to which lanes in the connecting road.

2.4.7 Differences from OpenDRIVE

Geometry

The geometries is maybe the most obvious difference between the OpenDRIVE standard and the Apollo OpenDRIVE specification. In the Apollo specification, the reference line of a road is defined by a simple list of latitude longitude points. In the OpenDRIVE standard, however, these lines are represented as functions of five different types.

Lanes

The overall structure of the lanes is very similar between the two versions of the standard, and they both use the same lane identification scheme. The main difference here is that Apollo requires more data. Where the OpenDRIVE standard allows to only specify the width of a lane, the Apollo specification requires points for describing lane borders. In addition to this the Apollo specification needs points for the center of each lane, and boundaries for the road.

Junctions

Junctions are more thoroughly defined in the OpenDRIVE specification than they are in the Apollo OpenDRIVE specification. The structure is very similar, but with a few differences. Apollo OpenDRIVE appear to implement a simpler version of junctions than the OpenDRIVE standard provides,

ignoring concepts like junction groups, junction priority, and junction controllers. Apollo OpenDRIVE does require the area of a junction to be defined, as opposed to the OpenDRIVE standard. Apart from this, the structure of the connections and links is more or less the same.

Objects and signals

The objects and signals are very similar, and the main difference here is maybe the reduced amount of attributes in Apollo objects, and the limited number of object types that are supported.

2.5 Global Navigation Satellite System

2.5.1 World Geodetic System

WGS84[29] is a standard used in cartography, geodesy and satellite navigation. WGS operates with two degrees, latitude and longitude. When describing a point using the WGS-standard, the longitude degree represents east-west, and latitude represents north-south. The longitude value is a number between -180 and 180, similarly the latitude degree should be a value between -90 and 90.

2.5.2 Universal Transverse Mercator coordinate system

The Universal Transverse Mercator coordinate system (UTM) is a system of map projections.[30] It divides the surface of earth into a grid of different zones. To refer to a location you will need both an x and y coordinate and which zone they are located within. One very practical property of UTM is that the coordinates are simply in meters, making it very convenient to use for calculations in contrast to the angle-based WGS.

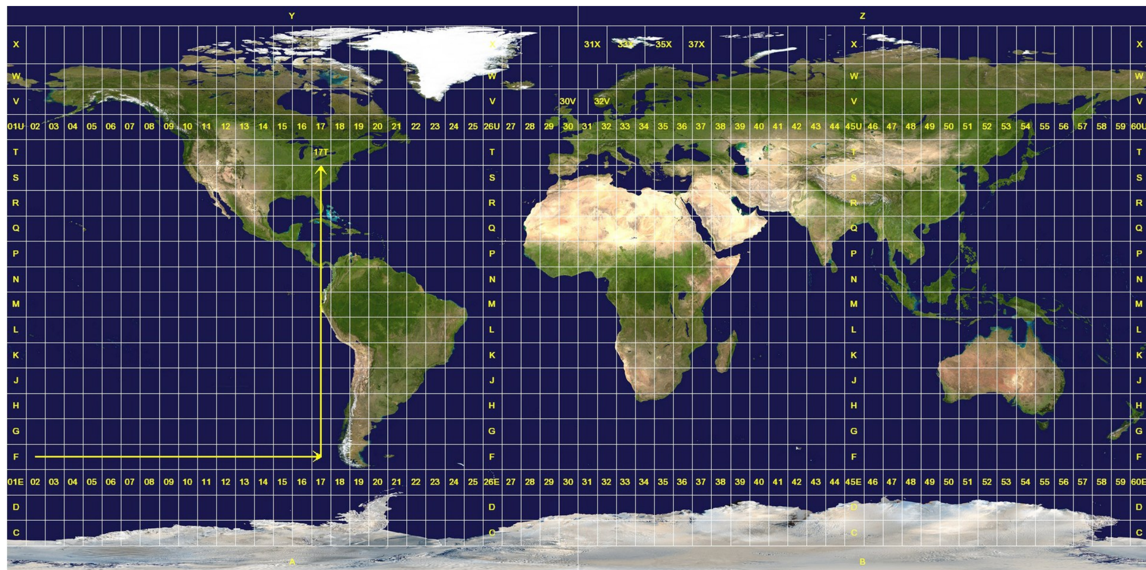


Figure 4: The UTM zones. Note how some zones differ in size, like the one for southern Norway, 32V. The image is retrieved from Wikipedia. [2]

3 Methodology

The goal of this master's thesis is producing a map that is compatible with the software stack of the NAP project's vehicle. This means generating a map following the Apollo OpenDRIVE specification for map representation. The map should be of the area surrounding Campus Gløshaugen in Trondheim, Norway. The main focus of this thesis is exploring the Apollo OpenDRIVE specification and learning how to use it.

3.1 Data source

In order to generate a map, some data is required as a basis. For road generation the desired data is mainly sets of coordinates with any associated metadata. As discussed in the background chapter, there is mainly two approaches to retrieve such data, gathering data from sensors or using existing openly available map data. This section will firstly discuss what information is desired, and then which method is best suited to obtain that information.

3.1.1 Data for road generation

The Apollo OpenDRIVE format requires several sets of coordinate points to represent each road. The center of the road represents the *reference line*, and each road has an overall boundary. In addition to this, every lane requires coordinates for the center of the lane, and for the boundaries of the lane. Assuming that these lines are parallell, the minimum requirement will be to have coordinates for any one of them, as the other lines can be generated using the known coordinates as a base. This is shown in figure 5.

In order to build a map, it is also necessary to have information about the road network topology. This means information about the road network and how the roads are connected to eachother. This includes information about the junctions within the area. Road network topology is a fundamental part of a map, and is necessary to make routes that the car can follow to get between two locations.

3.1.2 Available existing map data

Online map-services such as OpenStreetMap contain much information about roads across the entire world, often including metadata in addition to some line representation of the roads. Information from these sources are often easily available through APIs.

Although specialised sources exist, openly available online sources are usually not made for HD maps, and they rarely meet HD maps accuracy requirements. As for usage with the Apollo OpenDRIVE specification they also rarely include more than one set of points for each road. The metadata is also often inconsistent between roads, and there is no guarantee that important information is present.

Road network topology is usually obtainable through open maps, making it easy to build a

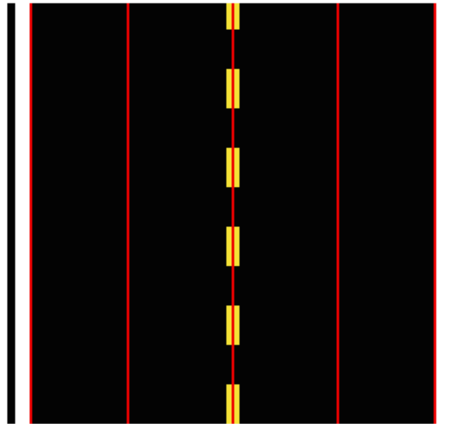


Figure 5: A visualization of a road segment. The red lines represent the coordinate data Apollo OpenDRIVE would require to store this segment.

network of roads, not having to do it from scratch.

3.1.3 Recording sensor data

In its simplest form, it could be sufficient to record coordinates using some simple GPS-device like a cell phone. The metadata would be added manually according to the projects needs. Depending on the accuracy of the GPS device, separate point sets could be recorded for each of the lines necessary in Apollo, although this could be difficult for large roads that would not allow travel by foot.

A more advanced way of doing such data colleciton would be mounting relevant sensors on a vehicle. Using camera and LiDAR input could make it possible to find the lane boundaries through perception and generate point sets of higher accuracy. Such a setup is expensive and requires substantial work to setup properly.

A challenge with using a sensor based approach for data collection, is gathering road network topology information. Retrieving such information is not necessarily that easy to do through sensors, and may require some other approach. One potential solution to this could be manually connecting and separating roads. Such an approach would require large amounts of manual work for generating large maps. Another potential solution could be using a third party map as a basis for the topology. This could possibly be done by matching collected GPS data to the third party map to obtain relevant metadata, however it is an advanced solution.

3.1.4 Data source conclusion

In this project, it was decided to use existing available map data as the method for data gathering. This decision was based on several factors. Firstly, the project's car was not ready for data collection yet as it was being setup simultaneously to this project. This greatly complicates the data gathering process in a sensor based approach. Although a simpler setup using a single GPS/IMU unit and

a camera could possibly be used to gather similar data, this was decided against. Another reason against basing the data gathering on sensor data is the road network topology issue. An HD-map should accurately describe the road network, and this would include intersections and road connections. A final reason for using existing map data is the project's goal of exploring the Apollo OpenDRIVE format. Using already available data meant that time spent processing sensor data and researching relevant algorithms could rather be spent researching the Apollo OpenDRIVE format and generating maps.

For many of the same reasons, it was decided to use OpenStreetMap as the data source. The main reason for using this rather than the norwegian national road data bank (NVDB) is that accessing OpenStreetMap information is simpler, and it represents the data in a way that maintains road network topology information in contrast to NVDB.

3.2 Map generation

The goal of the map generation process is to generate a map of the *Gløshaugen*-area for testing the Apollo software with our vehicle. Using OpenStreetMap as a basis for the map generation, we were able to download a map of the area. As described earlier, these maps are stored in an XML-format, and feature topology information as well as varying amounts of metadata about the different nodes and roads. This section will describe the approach that was taken to develop the proposed solution, the *osm2od.py* tool. The code for this solution can be found in appendix A.1 and A.2, or on GitHub[31].

3.2.1 Information parsing

Reading the information from the stored file is a simple matter of XML-parsing which is made easy by several available libraries. Using the python library *lxml.etree*, reading nodes from an XML-file is as simple as this:

Listing 3.1: Example code for reading data from an OSM XML file.

```
e = etree.parse(filename).getroot()
for node in e.findall('node'):
    print("Node␣{},␣lat␣=␣{},␣lon␣=␣{}".format(node.get("id"), node.
        get("lat"), node.get("lon")))
```

Listing 3.2: An example of a node from an OSM XML file.

```
<node id="78272" visible="true" version="4" changeset="686984"
   timestamp="2008-01-23T23:15:19Z" user="steinarh" uid="17110" lat=
   "63.4215227" lon="10.3989347">
```

The above XML-code will produce the following output by running the example code:

```
Node 78272, lat = 63.4215227, lon = 10.3989347
```

The *findall*-function makes a list of all the matching elements in the XML-file. By using the *get*-function that each element implements, attribute values can easily be read for every element.

3.2.2 Road generation

The Apollo OpenDRIVE specification requires several point sets per road. These points are to represent different boundaries and parts of the roads. Examples include the point sets to describe the borders of the lanes, and the center of the road. In OpenStreetMap, each map is represented with a single point set. It is necessary to assume that this set of points represents the middle of the road. For OSM to be a viable data source, we are required to be able to generate new lines based on the coordinates from the OSM nodes.

Generation using vectors

The new lines would be required to be parallel with the original line, and maintain the same distance between the two lines at any two corresponding points. If this is done wrong, the roads can for instance seem really narrow when they turn. As a solution for this, a method of generation using vectors were implemented. This method would make vectors of every point and the next point. The point data extracted from the OSM XML-files is stored in a ordered list, making it easy to get the next and previous point of every point. Making new orthogonal vectors from each vector is very simple, and would ensure the same distance at every point.

The new points are generated by simply taking the vector between two points, turning them 90 degrees, and scaling their length according to the desired road width. A third party library is used to make sure the conversion between meters and two points in latitude and longitude coordinates is correct.

One issue with generating new points is the geographic goordinate system, where latitude and longitude values would represent a different value in meters, potentially rendering this approach useless. To avoid this, all points are converted to UTM coordinates before the calculations are done, and then converted back to latitude and longitude before they are stored in the map.

A fundamental issue with this approach is its behaviour in steep turns. Consider a turn of 90 degrees as shown in figure 6. This figure shows the corner of a road consisting of three points, represented with the two black vectors. The red vectors show the new, orthogonal vectors created as described in the previous paragraph. The new parallel line, shown in green, will completly ignore the corner.

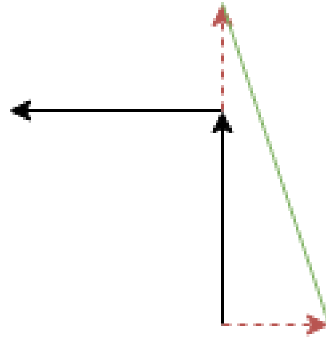


Figure 6: An illustration of the issue with the first vector based approach.

Angle based generation method

To avoid this problem, this method of point generation is only used for the first and last point. For the rest of the points, we take an additional point into consideration when generating the new point. Previously we would make a vector between the current and the next point, but the new solution also uses the previous point. From these points it constructs two vectors instead of just one, and calculates the angle between them. The angle is then halved to find the position for the new point. This removes the issue described in figure 6.

This solution does not maintain a consistent lane width, however. To combat this, simple trigonometry is used to adjust the distance of the new point to the old instead of using a static width.

$$width = |\frac{width}{sin(angle)}| \tag{3.1}$$

Figure 7 shows an example of generating a new point for a point in a 90 degree corner. The red arrow is the vector that is scaled with equation 3.1. The two green vectors show the desired lane width. The large arc is the angle between the two vectors, and the small arc is half of that angle.

Figure 7: An illustration of the scaled vector length to preserve lane width.

In order to make the actual XML for the roads, the list of points from OSM is used as a basis for the road. The method for generating parallel lines as described above is used several times on these points to obtain all the new points required by Apollo. For each lane this method has to be ran two times, once for the center line, and once for the border line. In addition to this, the road requires coordinate points for the overall boundary lines of the road. These points will be the same as the border for the outer lanes in each direction.

An issue with pre-existing information in general is that different roads might be plotted in "different directions", resulting in issues when they meet. An example of this is when two single lane roads meet. If they are plotted in different directions we might get a situation where the roads generated with the method as described above will be skewed relative to each other as described in figure 8.

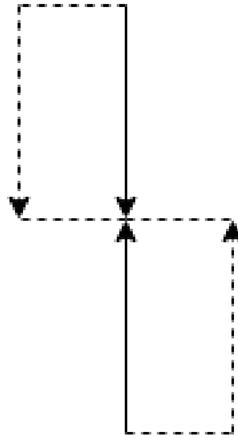


Figure 8: An illustration of the issue with meeting single lane roads that are plotted in different directions. The dotted lines are those generated, the solid lines are the data from OSM.

This was solved by adding single lane roads as a special case where the original points would be used as the middle of the lane instead of the center of the road. The difference is that the center of the road is supposed to be the line that separates lanes that goes in different directions. For a single lane road this would be one of the border lines. As the single lanes are always assumed to be bidirectional, this solution should be sufficient, and is visualised in figure 9.

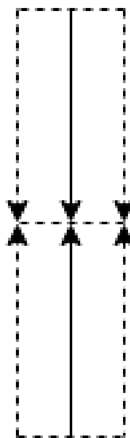


Figure 9: How the roads appear when the data from OSM is used as the center line.

3.2.3 Road connections and junctions

There are several possible ways to connect roads in Apollo's OpenDRIVE. Each road has fields for preceeding and succeding roads and lanes, where connections can be done directly. There is also a *junction*-tag made to describe intersections. This tag is relatively advanced, and can describe intersections involving several meeting roads. This naturally means it is challenging to generate junctions correctly.

In OpenStreetMap, intersections are represented as a single node that all intersecting roads share. Detecting intersections is a simple task, look for nodes that are present in two or more road's node sets. When generating the roads as described earlier, all the nodes for the current road is matched with the nodes of all the other roads. If there is a match, that node is registered as a junction, and the relevant roads are added to the junction.

Once all the roads are generated, we have a complete list of all the intersections in the area to be mapped. This list is then iterated to generate the necessary XML for each junction. For each junction we have a number of *incoming roads*. The specification requires the junction's area to be specified, and also uses so-called *connecting roads* to connect incoming roads to eachother. The connecting roads are regular roads, but as opposed to other roads they *belong* to a certain junction. In the proposed solution, we assume that all incoming roads are to be connected to eachother.

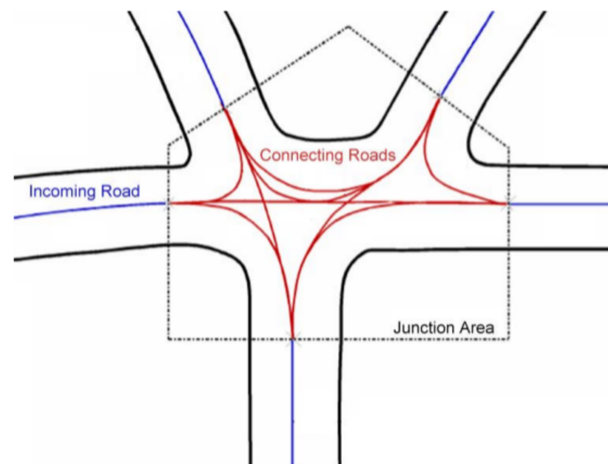


Figure 10: An illustration of connecting roads within junctions. The image is taken from the OpenDRIVE specification [3]

For each pair of incoming roads, a Bézier curve[32] of new points is generated to represent the curve a car could drive to connect from one road to the other. Both of the incoming roads and the connecting road is then tagged with the appropriate succeeding and preceeding roads. Connections are also established between the lanes of the incoming roads and the connecting roads. In the current solution, the connection roads always only have one lane. This is the simplest way

of establishing the connections, but will not represent fully realistic paths a car can actually follow. This means that if an incoming road has more lanes, they will all be connected to the same lane on the connecting road.

3.3 Apollo OpenDRIVE

While working with the Apollo OpenDRIVE specification, it is apparent that it contains numerous errors and inconsistensies. In this section these discovered issues will be described to aid future work with the specification.

3.3.1 Lanes and center lines

The specification gives the impression that the records defining the right, left and center lanes should follow the same structure. This is however not the case. The right and left lanes do follow the same structure, but the center lane is completly different, and this format is not explained in the specification. In Apollo OpenDRIVE the center lanes only defines the reference line, and not an actual lane. The actual format to be used is explained in section 2.4.

3.3.2 Border types

In the specification, the structure of a lane's type is supposed to consist of one or more *borderType* tags within a *borderTypes* record. This is however not the case. Apollo's XML parser expects the border type to be specified within one or several *borderType* tags directly within the *border* record. These tags should have *sOffset*, *type* and *color* attributes. The *sOffset* attribute defines at which section of the road the border type should be valid for.

3.3.3 Sample Associations

According to the specification, this is an optional tag that is not required. When trying to use the *proto_map_generator*-tool to generate *.bin* and *.txt*-maps, however, it will crash if it cannot find the sample association tags. It will generate maps successfully if the script is modified to ignore the sample associations, but it is unknown whether the tag is required by Apollo or not.

When generating these tags, it also comes clear that the specification uses the wrong names on the tags. The list below to the left describes how the specification claims the format should be, and the list to the right shows how it should be in reality.

Specification	Reality
sampleAssociations	sampleAssociates
└ sampleAssociation	└ sampleAssociate
roadSampleAssociations	roadSampleAssociations
└ sampleAssociation	└ sampleAssociation

3.4 Pipe-line for map generation

The code produced in this project results in a stack that makes it easy to generate new Apollo OpenDRIVE maps from OSM data. By following a few simple steps, the generation can easily be

done to get maps of new areas. The process consists of data gathering, map generation and finally some converter scripts made by Apollo is required to be run to generate all the necessary files before it can be used with Apollo.

3.4.1 Data gathering

The program only needs OpenStreetMap data to function, and it requires this as an XML-file containing all available data of the selected area. Such a file can easily be downloaded through OSM's website. In order to do this through the OSM website, simply pan and zoom the map to show the desired area. Then click the export-button on the top of the website. Finally click the new export-button to the left to download the map. If this fails, the website should list some available mirrors as alternative download options.

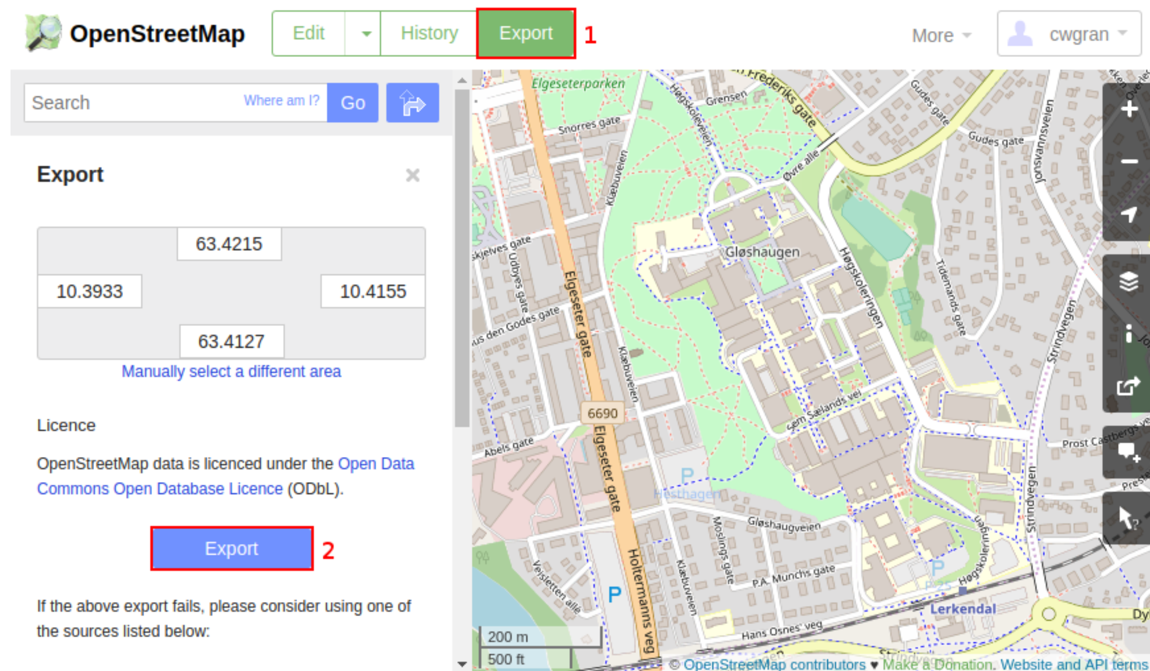


Figure 11: A screenshot of the OSM website showing how to download XML files.

3.4.2 Map generation

Once a map has been downloaded, map generation is as simple as inputting the downloaded file as a parameter to the python script. The script also needs to know which UTM zone the map is within. It assumes zone 32V unless an other zone is specified. Given a map simply called *map*, the following command can be used to generate an OpenDRIVE map using the *osm2od* tool:

```
python3 osm2od.py map -z 32V
```

Custom road configuration

If necessary, it is possible to specify custom lane widths and the number of lanes for specific roads given by their road names, and input this to the program as a CSV-file. An example of a line in such a file is:

```
Sem Sælands Vei,1,5
```

This would apply a lane width *5 meters* and specify the amount of lanes to be *1* for the road *Sem Sælands Vei*. Every road-section the program encounters that has a *name*-parameter with this name, will use these properties instead of those specified in the OSM file. The specified lane width will be applied to every lane within the relevant roads. This feature can be useful whenever the OSM data is incomplete or outdated. The program also assumes that all roads have equally wide lanes, but this might not be the case. Being able to specify this information can help improve the accuracy of an otherwise relatively inaccurate map.

Applying such a config-file is simple:

```
python3 osm2od.py map -z 32V -c config.csv
```

3.4.3 Map conversion

The *osm2od*-program generates an Apollo OpenDRIVE XML file, but Apollo actually don't use this file specifically much. Instead it uses modified versions of the map in other file formats. In order to use the map with Apollo, it is necessary to generate these files, and place them in the appropriate folder. Luckily, Apollo provides the necessary scripts for these conversions.

Generating .bin and .txt

The *.bin* and *.txt* files are simply representations of the map in different formats than XML. To generate these files, it is necessary to use the script *proto_map_generator.cc* from the */apollo/modules/map/tools*-folder. Inside the docker, once Apollo is appropriately built, the compiled file is placed within */apollo/bazel-bin/modules/map/tools*.

The simplest way of using this script is specifying an input and output location of the same folder, this wil ensure that the generated files are placed correctly. The appropriate folder to store maps is */apollo/modules/map/data*. Create a new subfolder within that directory and place the generated XML file within the new folder. The command to be ran to generate the new files is:

```
./bazel-bin/modules/map/proto_map_generator --map_dir=[map-path]
    --output_dir=[map-path]
```

Here *[map-path]* should be replaced with the path to the new map folder. If no output directory is specified, the files will be placed in a temporary folder.

Generating routing map

Apollo's routing module requires a routing map. This can be generated by using another script provided by Apollo, this time located in the */apollo/scripts*-folder. The name of this script is *gen-*

erate_routing_topo_graph.sh, and in contrast to the other scripts, it only needs one parameter, the map directory. It will automatically place the output files in the same directory.

```
./scripts/generate_routing_topo_graph.sh --map_dir=[map-path]
```

Generating simple map

The Dreamview frontend can display a simple version of the map. To generate this simple map, use the provided script called *sim_map_generator.cc*. Similarly to the script for generating *.bin* and *.txt*-files this script is located in the *bazel-bin* directory, specifically inside the map-tools folder. In the docker, this can be run similarly to the other scripts:

```
./bazel-bin/modules/map/tools/sim_map_generator --map_dir=[map-
    path] --output_dir=[map-path]
```

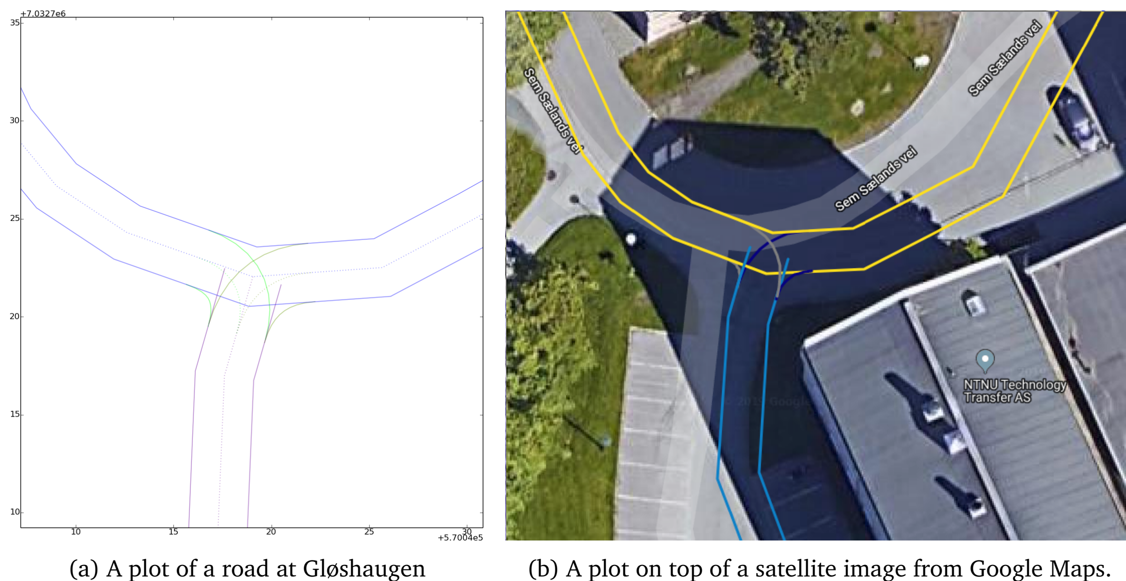
4 Results

4.1 Map results

The main goal of this project was generating Apollo OpenDRIVE maps, and more specifically the main focus was successfully importing roads and intersections from OpenStreetMap. In this section the final version of the generated maps will be presented. The plots are obtained through scripts made by Apollo.

4.1.1 Road lanes and width

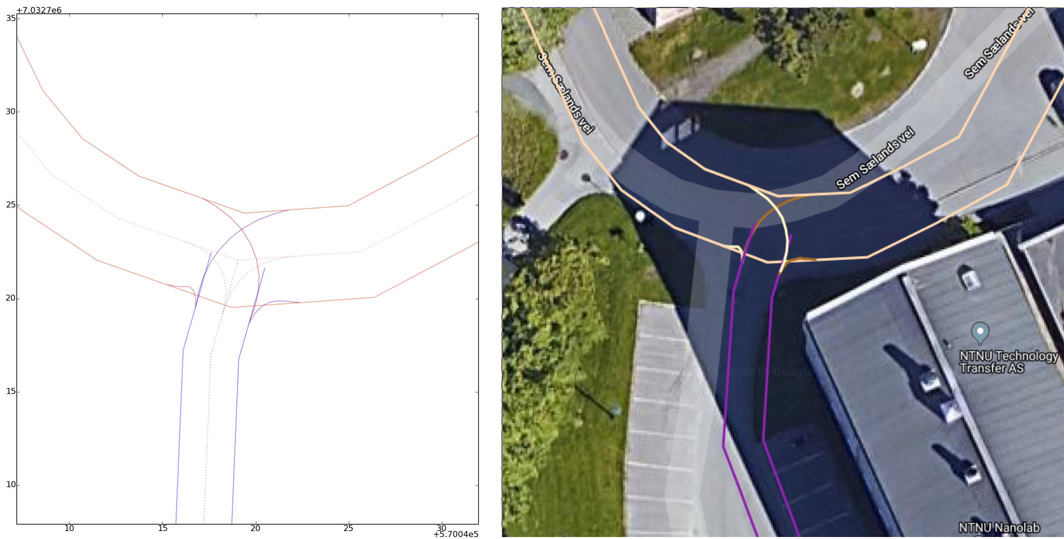
Figure 12, 13 and 14 shows a part of a map of campus Gløshaugen in Trondheim, Norway. This part of the map has a junction where two roads meet. Figure 12 simply visualises the data read directly from OpenStreetMap. In figure 13 and 14 a configuration file has been applied to change the lane width and number of lanes respectively.



(a) A plot of a road at Gløshaugen

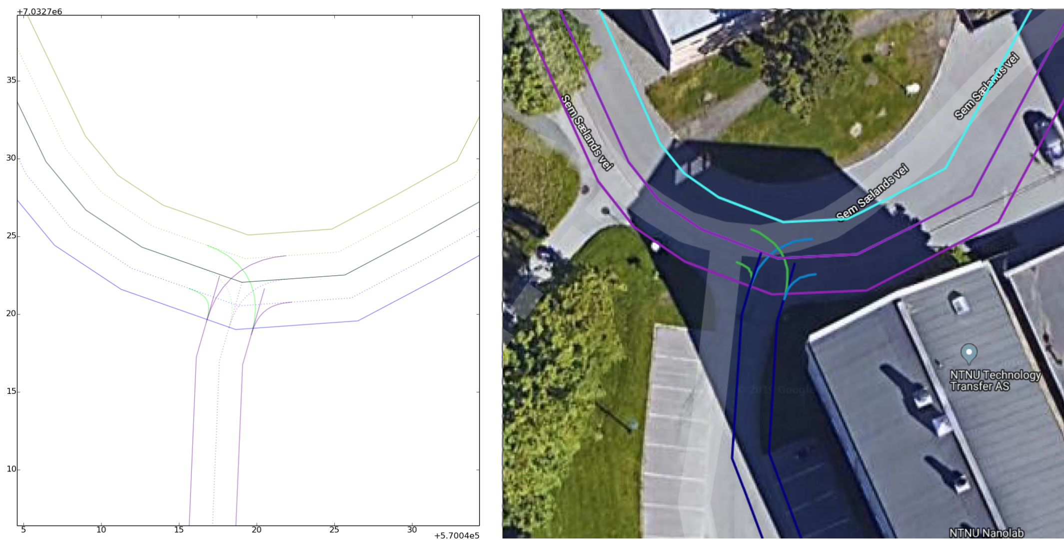
(b) A plot on top of a satellite image from Google Maps.

Figure 12: Two plots of the same map



(a) A plot of a road at Gløshaugen (b) a plot on top of a satellite image from Google Maps.

Figure 13: The same roads, but the map is using a config-file telling one of the roads to be wider.



(a) A plot of a road at Gløshaugen (b) a plot on top of a satellite image from Google Maps.

Figure 14: The same roads, but this time, one of the roads have been configured to have two lanes.

In figure 14 it is shown a junction between a two-lane road and a one-lane road. This illustrates how every connecting road currently will only be one lane, no matter what kinds of roads it

connects. This may look wrong in a plot, but it does follow the intended behaviour. All lanes are connected appropriately from the incoming roads to the connecting roads in the map file, but as the connecting roads are actual roads defined by coordinates the visualization does not show these connections.

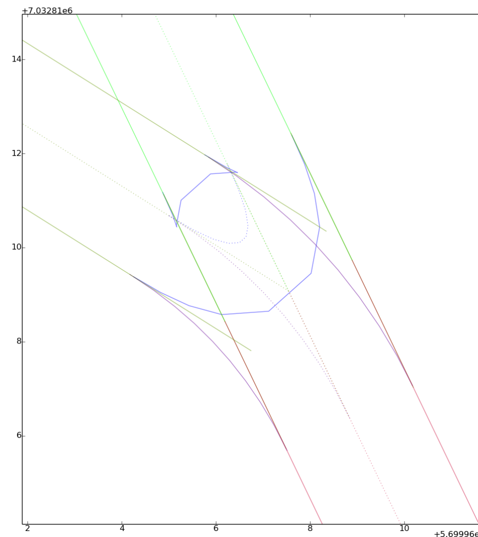


Figure 15: A plot of a junction where two of the roads meet at a very sharp angle.

In figure 15 it is shown how sharp turns currently produces an erroneous inner border line. This is not a bug, but rather a consequence of how the border lines are generated. Each point in the border represents a point in the center line, but moved a set length in a certain direction. To fix this, a new method for parallel line generation would be required, and as this is only a problem in some junctions, it has not been an area of focus in this project.

Apollo's graphical user interface, Dreamview, display a simple version of the map as a background when the car is driving. The next two figures shows how the map looks in Dreamview. Figure 16 shows the view of the car when it is driving. As recorded sensor data from a trip is required to make the car actually move, this image is the default starting location in the map. Figure 17 shows an overview of the entire map through the *Route Editing* interface. In these figures the line colors and types are represented as defined in the map, yellow and white in this instance. As the map is a simplified version of the full map, some coordinate points are missing. Figure 18 shows the same junction as figure 12, 13 and 14, but it is apparent that the connecting roads have fewer points in this version of the map.

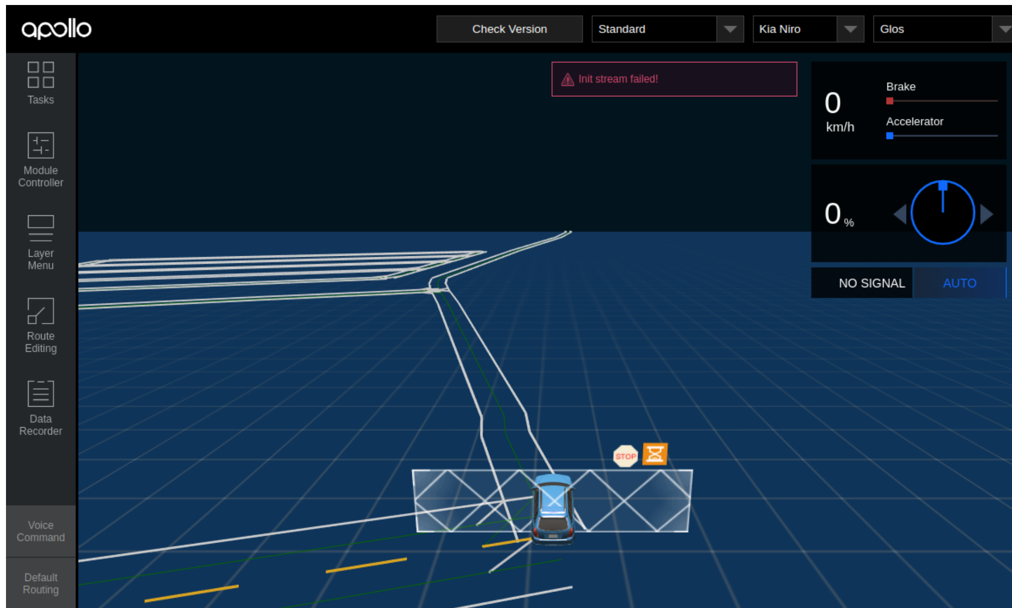


Figure 16: A screenshot from the Dreamview interface showing the simple version of the map.

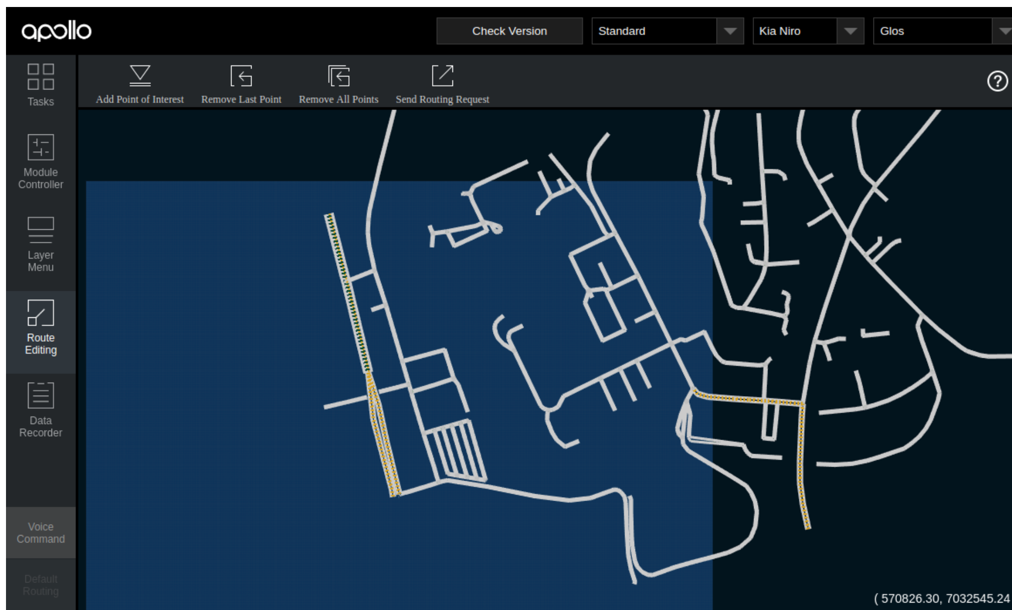


Figure 17: A screenshot showing an overview of a generated map in the Dreamviewer GUI.

Figure 18: A closer look at the simplified map.

5 Discussion

This chapter will discuss the results presented in the previous section.

5.1 Generated maps

It was discovered in chapter 2.1.1 that there are no available open source solutions for generating HD-maps, answering **RQ1.1**. The maps generated with the *osm2od* tool developed in this thesis are successfully parsed by the different scrips provided by Apollo for map conversion as explained in section 3.4.3. The figures shown in chapter 4 are made by other Apollo provided scripts. This makes it clear that the generated maps follow the specification well enough to be accepted by Apollo. Whether this means that they will work with the car is still unknown, and would require testing with the vehicle before anything can be concluded.

5.2 Lanes

The lanes are one of the most important parts of the map, and it is important that they function correctly. In the plots shown in figure 12, 13, 14 and 15, the lane borders and center can be seen, so the point sets describing the different parts of the lanes are structured correctly. However, these figures also show that compared to a satellite image from Google Maps, the roads are not plotted particularly accurately. This can come down to several factors, but the main problem is the quality of the input data. There is no accuracy guarantees on the data from OpenStreetMap[33]. Other potential factors can include an erroneous number of lanes or lane width. For each road there is also only one set of points, which will normally represent the center of the road. The points for the centers of the lanes and all the borders will be estimated based on how wide the road is, and this estimation may be incorrect. Alternative ways of solving or improving this issue would require a different approach of data gathering altogether. Two potential ways could be:

1. Lane detection on satellite images to detect lane borders.
2. Lane detection on sensor data from a data collection vehicle.

The first option would require satellite data of the area that is to be mapped. Further it would require researching algorithms for extracting information about lanes from these images. This approach could potentially be utilised to get more accurate data for lane widths.

The second option is an advanced approach that could utilise several sensors, including a GNSS reciever, LiDAR and cameras. Combining output from these sensors can make it possible to get accurate coordinates for lane borders and centers, possibly by using simultaneous localization and mapping (SLAM) algorithms.

Another inaccuracy is that the script currently only allows either one lane, or an even number of lanes. This is not realistic as many roads have a different amount of lanes for each direction.

OSM actually provides some information about the number of lanes in each direction, but the script currently does not take this into account. Incorporating this into the script, would require a large structural change of the code. This is because the approach taken assumed we would only deal with an even number of lanes, and the scope of this thesis mainly span roads with one lane. An HD-map would require the actual number of lanes to be represented.

5.3 Junctions

Connecting meeting roads correctly is important for an accurate representation of the topology of the road network. As shown in figure 12, 13, 14 and 15 the map presented in this thesis does implement connections between roads, but currently limited to one connection for each road. An accurate map would likely require connecting roads that represent the path a car can follow from each of the lanes in one incoming road to all the lanes of another, potentially in the form of a multi-laned connecting road. The visualisation scripts provided by Apollo have no way of showing road connections other than the actual connecting roads. The connections are represented with *predecessor* and *successor* records, and with data from a third party it can sometimes be a challenge to understand which road would be which. Not being able to see what Apollo interpret visually means that this part of the program would require testing with the Apollo software. This has not been possible during this time period, and this part of the program is poorly tested and may contain errors.

If the connecting roads were to be implemented in a way where there is one connecting road for each possible path a car can take in a junction, it would be necessary to use the lane's driving direction into consideration to figure out which paths are possible. Currently all single lane roads are bidirectional, so in junctions featuring only such roads, all lanes of a road would connect to all lanes of the other roads. For multilane roads, the current solution only supports an equal amount of lanes on each side of the center line, meaning there will always be at least one entering lane and one exiting lane. Should the implementation of lanes be extended to include roads with a different amount of lanes on each side of the center line, and one-way roads with only one lane, these two guarantees no longer hold. This could potentially be an issue, specially when dealing with data from a third party. It is easy to imagine a situation with three incoming roads all entering the junction with no possible connecting paths. This problem would need to be considered when implementing a more advanced junction solution.

5.4 Objects and Signals

Objects and signals make up an imporant part of HD-maps. The main priority in this thesis was representing the road structure of a given area, not including objects and signals. One of the main challenges concerning objects and signals in the Apollo OpenDRIVE specification is that the *object* and *signal* records are to be registered under a *road* record. When using third party data as a basis for the map generation, this would require detecting which of the map's roads is closest to the object or sign. This road is not necessary identifyable by it's name as long roads may be split into several OpenDRIVE roads. The road name parameter is also not guaranteed to be present in

OpenStreetMap. Signs in OpenStreetMap are marked using nodes. These nodes can either be a part of a way or not. If they are part of a way their exact location is not necessarily given. If they are not part of a way, it is not necessarily given which road they belong to and which direction they face. Apollo also uses three dimensional coordinates to describe the position of a sign, where the z-coordinate is given in meters and represents the height above ground, while the other two are defined using the *WGS84* standard. The data from OpenStreetMap could likely be used to estimate the location of a sign, but it is unknown whether this would be sufficiently accurate to satisfy the requirements of Apollo.

5.5 Performance

The performance of the script has not been an area of focus during this project, and generating large maps is very slow. If this code is to be developed further, this is something that should be looked into. The main focus this period has simply been generating a map, and some assumptions and shortcuts limits the maintainability. Extending it with a parallel approach to road and junction generation would likely improve performance, and while this is a simple task in Python, it would require a major overhaul of the code. The best approach is possibly to make entirely new code with a greater focus on maintainability, using this project's code as inspiration.

5.6 Base data

Using already existing open map data was convenient to allow exploration of the Apollo OpenDRIVE specification. The data is easily accessible both to gather and process, and makes it possible to focus mainly on the generation of the XML-code. For a real use, however, the map data likely cannot be used, as it is too inaccurate and not detailed enough to satisfy HD-map requirements. In the visualisations presented in chapter 4, it becomes clear that the OSM data contains to few points and that they are not accurate enough. Information about the lane width is also required to obtain an accurate estimation of lane borders. As explained roughly in section 5.2 this would require a different approach to solving the data gathering problem. **RQ1.2** asks whether it is possible to use open data for HD-map generation. While **RQ1.3** is satisfied using open data, the accuracy requirement of an HD-map is unlikely to be satisfied using open data, making the answer of **RQ1.2** no.

5.7 Reflection

If this project were to be redone knowing the information learned in this project, the approach would likely be different. In regards to the data collection, an approach based on sensors would probably achieve better accuracy, and better HD-maps if done correctly. On the other hand, the car was not ready for data collection during this semester, so it was not really an option. And one of the main goals for this thesis is, as presented in **RQ1.3**, exploring the Apollo specification of OpenDRIVE. This could maybe have been done more efficiently by working more with the tools provided by Apollo too. The output from the code should maybe have been tested more frequently with the Apollo software stack to make sure the maps were going in the right direction.

In terms of the code for generating the maps, it would likely have been wise to make the code a bit more scaleable. Firstly the code makes assumptions from the start that have shown to limit how much it can be improved with further features. Secondly, generating large maps takes an unnecessarily large amount of time. Restructuring the code could reduce its complexity and also the amount of code. Currently the connection roads are generated seperately from the regular roads, but their implementation is very similar and could likely be combined to one. The current assumptions that every road has an equal amount of lanes on each side of the center line would also require a large restructuring of the code to change. Future projects should consider having better structured code with maintainability more in mind, to make sure it is easy to extend it with features.

6 Conclusion

Obtaining a HD-map is a matter of gathering the right information, process it appropriately and output it in the right format. There are some HD-map services available, but the research done in this thesis found that they are generally not open source, and rarely explain in detail how they make their maps. This led to the best way of obtaining a map for the NAP lab being generating one by making a script for map generation. In order to generate a map it was necessary to collect enough appropriate information about the relevant area. This can be done in several ways, but in this thesis the approach is based on open data from OpenStreetMap. The result was compatible with the Apollo framework, at least as far as it has been tested currently. Some inconsistencies and errors were found in the Apollo OpenDRIVE specification, but analysis of the code that reads the map made it possible to make a compatible map. The accuracy of the map is questionable, however, and it is likely not accurate enough to satisfy the requirements for an HD-map.

6.1 Future work

This thesis provides insight into how the Apollo OpenDRIVE specification works, and how maps can be generated to fit the standard. Future maps, however, will have higher requirements in terms of accuracy and details than the results from this thesis currently provides. Future projects should investigate how maps can be generated using data collected with data collection vehicles. Such data includes coordinates, images and point clouds. This kind of approach likely have a higher potential in terms of accuracy and details, but it is a far more advanced method, and might require substantial amounts of manual work. An alternative to this, could also be using some kind of image detection on satellite images to increase the accuracy of map data, specifically lane widths, the amount of lanes and junctions.

In addition to the roads themselves, an important aspect of the maps are the junctions. These need to be properly setup for the map to function optimally, and for the map to properly represent the road network topology. This includes connecting intersecting roads with eathother properly, as well as making sure the lanes within the connected roads are linked to represent the possible paths a car can follow.

It should also be a priority to include road objects and signals to future maps, as they provide useful information to the application. This will require detecting roadside objects, including signs, traffic lights, and road markings. Some of these may be available in online databases like the norwegian national road data bank, but then again the accuracy of the data may not meet the required level.

Bibliography

[1] Baidu. 2018. Apollo 3.0 software architecture. https://github.com/ApolloAuto/apollo/blob/master/docs/demo_guide/images/Apollo_3.0_diagram.png. Accessed: 2019-06-26.

[2] Universal transverse mercator coordinate system. https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system. Accessed: 2019-06-24.

[3] Dupuis, M. *OpenDRIVE Format Specification, Rev 1.5*. VIRES Simulationstechnologie GmbH, February 2019.

[4] 2019. Ntnu autonomous perception. https://www.ntnu.edu/web/ntnu-autonomous-perception/naplab. Accessed: 2019-06-26.

[5] Apollo. 2018. Map open service. http://data.apollo.auto/hd_map_intro?locale=en-us&lang=en. Accessed: 2019-06-24.

[6] Apollo. 2018. Lesson 2: Hd map. http://apollo.auto/devcenter/courselist.html?target=2. Accessed: 2019-06-24.

[7] comma.ai. https://comma.ai/. Accessed: 2019-06-25.

[8] Hd maps for the masses. https://medium.com/@comma_ai/hd-maps-for-the-masses-9a0d582dd274. Accessed: 2019-06-25.

[9] Mapillary. https://www.mapillary.com/. Accessed: 2019-06-25.

[10] Deepmap. https://www.deepmap.ai/. Accessed: 2019-06-25.

[11] Tomtom hd map with roaddna. https://www.tomtom.com/automotive/automotive-solutions/automated-driving/hd-map-roaddna/. Accessed: 2019-06-25.

[12] lvl5. https://lvl5.ai/. Accessed: 2019-06-25.

[13] Here hd live map. https://www.here.com/products/automotive/hd-maps. Accessed: 2019-06-25.

[14] Civil maps. https://civilmaps.com/. Accessed: 2019-06-25.

[15] Carmera. https://www.carmera.com/. Accessed: 2019-06-25.

[16] Tri-ad and carmera team up to build high definition maps for automated vehicles using camera data. https://global.toyota/en/newsroom/corporate/26879165.html. Accessed: 2019-06-25.

[17] Apollo. 2019. Map creation tool. https://github.com/ApolloAuto/apollo/tree/master/modules/tools/create_map. Accessed: 2019-06-24.

[18] LG Electronics Inc. 2019. About lgsvl. https://www.lgsvlsimulator.com/about/. Accessed: 2019-06-24.

[19] LG Electronics Inc. 2019. Lgsvl simulator: Map annotation. https://www.lgsvlsimulator.com/docs/map-annotation/. Accessed: 2019-06-24.

[20] OpenStreetMap. 2019. About openstreetmap. https://www.openstreetmap.org/about. Accessed: 2019-06-24.

[21] 2019. openstreetmap wiki. https://wiki.openstreetmap.org/wiki/Main_Page. Accessed: 2019-06-24.

[22] Statens vegvesen. 2018. The national road database. https://www.vegvesen.no/en/professional/roads/national-road-database/. Accessed: 2019-06-24.

[23] Difi. Norwegian licence for open government data (nlod) 2.0. https://data.norge.no/nlod/en/2.0. Accessed: 2019-06-24.

[24] Baidu. 2019. Apollo. http://apollo.auto/. Accessed: 2019-06-26.

[25] Baidu. 2019. Apollo. https://github.com/ApolloAuto/apollo. Accessed: 2019-06-26.

[26] Florent Revest, H. H. 2019. Software platform - introduction to apollo. https://nap-lab.gitbook.io/nap-lab/software-platform#introduction-to-baidu-apollo. Accessed: 2019-06-26.

[27] Apollo. *Apollo 3.0 HDMAP OpenDRIVE Format*. Baidu, July 2018.

[28] 2019. How to generate opendrive formate file like base_map.xml. https://github.com/ApolloAuto/apollo/issues/3005. Accessed: 2019-06-26.

[29] Øystein B. Dick. 2018. Wgs84 - store norske leksikon. https://snl.no/WGS84. Accessed: 2019-06-26.

[30] Kartverket. Jordas rutenett. https://www.kartverket.no/kunnskap/Kart-og-kartlegging/Jordas-rutenett/. Accessed: 2019-06-24.

[31] Gran, C. W. 2019. osm2opendrive. https://github.com/CWGran/osm2opendrive. Accessed: 2019-06-26.

[32] Nicholas M. Patrikalakis, Takashi Maekawa, W. C. 2009. Shape interrogation for computer aided design and manufacturing (hyperbook edition). http://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node12.html. Accessed: 2019-06-26.

[33] Accuracy - openstreetmap wiki. https://wiki.openstreetmap.org/wiki/Accuracy. Accessed: 2019-06-24.

A Appendices

A.1 road.py

Listing A.1: road.py

```
class Road:

    def __init__(self, id):
        self.id = id
        self.nodes = []

class Node:

    def __init__(self, id, lat, lng):
        self.id = id
        self.lat = float(lat)
        self.lng = float(lng)
```

A.2 osm2od.py

Listing A.2: osm2od.py

```
import numpy as np
import math
import datetime
import argparse
import utm

from geopy import distance
from lxml import etree
from tqdm import tqdm

from road import Node, Road

utmz = {"zone":32, "letter":"V", "full":"32V"}

def readNodes(e):
    nodes = {}

    # Read all nodes and their coordinates into an array
    for node in e.findall('node'):
        n = Node(node.get("id"), node.get("lat"), node.get("lon"))
        nodes[node.get("id")] = n
    return nodes


def readRoads(e, nodes):
    roads = []

    # Desired road types
    driveable = ["motorway", "trunk", "primary", "secondary", "
        tertiary", "residential", "service", "living_street", "track"
        , "road", "unclassified"]
    for r in driveable.copy():
        driveable.append(r + "_link")

    # Read all roads/ways into an array
    for road in e.findall('way'):
        r = Road(road.get("id"))

        supported = False

        # Read all information about each road
        for tag in road.findall('tag'):
            setattr(r, tag.get("k").replace(":", "_"), tag.get("v"))
```

```
42
43                # Filter out unwanted roads
44                if tag.get('k') == "highway":
45                    if tag.get('v') in driveable:
46                        supported = True
47            if not supported:
48                continue
49
50            # Connect to the nodes
51            for nd in road.findall('nd'):
52                r.nodes.append(nodes[nd.get("ref")])
53
54            roads.append(r)
55
56        return roads
57
58  def readOSM(filename):
59        e = etree.parse(filename).getroot()
60
61        print("Reading␣file␣{}".format(filename))
62
63        nodes = readNodes(e)
64        roads = readRoads(e, nodes)
65
66        print("Finished␣reading␣file,␣found␣{}␣nodes␣and␣{}␣roads.".
              format(len(nodes), len(roads)))
67
68        return nodes, roads
69
70  def read_config(filename):
71        f = open(filename)
72        conf = {}
73        for l in f:
74            s = l.strip().split(",")
75            s[0] = s[0].lower().replace("␣", "")
76            if s[1] != "":
77                s[1] = int(s[1])
78            else:
79                s[1] = None
80            if s[2] != "":
81                s[2] = float(s[2])
82            else:
83                s[2] = None
84            conf[s[0]] = s[1:3]
85        f.close()
86        return conf
```

```
87
88 def format_coord(n):
89     return "{:.9e}".format(n)
90
91 def buildXML(filename, roads, pretty, conf):
92
93     name = filename.split(".")[0].split("/")[-1]
94     #filename = name + ".xml"
95     filename = "base_map.xml"
96
97     print("Building␣XML␣output...")
98
99     root = etree.Element('OpenDRIVE')
100    root.set("xmlns", "http://www.opendrive.org")
101    tree = etree.ElementTree(root)
102
103    # Setup header record
104    header = etree.SubElement(root, "header")
105    header.set("revMajor", "1")
106    header.set("revMinor", "0")
107    header.set("vendor", "Baidu")
108    header.set("name", name)
109    header.set("version", "1.0")
110    header.set("date", datetime.datetime.now().strftime("%Y-%m-%dT%H
           :%M:%S"))
111
112    # Maximum and minimum coordinate values
113    # North, south, east, west
114    max_coord = [None,None,None,None]
115
116    # Setup Geo Reference
117    georef = etree.SubElement(header, "geoReference")
118    georef.text = etree.CDATA("+proj=longlat␣+ellps=WGS84␣+datum=
           WGS84␣+no_defs")
119
120    junctions = {}
121    for r in tqdm(roads, "Generating␣roads"):
122        road = etree.SubElement(root, "road")
123
124        lane_width = 3.0
125
126        road.set("id", r.id)
127
128        road.set("junction", "-1")
129
130        # Lanes
```

```
131         lanes = etree.SubElement(road, "lanes")
132 
133         num_lanes = 0
134         lane_nums = [0,0]
135         if hasattr(r, "lanes_forward"):
136             lane_nums[0] = int(r.lanes_forward)
137         if hasattr(r, "lanes_backward"):
138             lane_nums[1] = int(r.lanes_backward)
139 
140         if hasattr(r, "lanes"):
141             lane_nums[0] = int(r.lanes)//2 + int(r.lanes)%2
142             lane_nums[1] = int(r.lanes)//2
143 
144         num_lanes = lane_nums[0] + lane_nums[1]
145         if num_lanes == 0:
146             lane_nums[0] = 1
147             num_lanes = 1
148 
149         if hasattr(r, "name"):
150             road.set("name", r.name)
151 
152             name = r.name.lower().replace("␣", "")
153             if conf != None:
154                 if name in conf.keys():
155                     if conf[name][0] != None:
156                         num_lanes = conf[name][0]
157                         print(num_lanes)
158                     if conf[name][1] != None:
159                         lane_width = conf[name][1]
160         else:
161             road.set("name", "")
162 
163         laneSec = etree.SubElement(lanes, "laneSection")
164         laneSec.set("singleSide", "true")           # True if both
               directions share the same laneSection
165 
166         boundaries = etree.SubElement(laneSec, "boundaries")
167 
168         # If the lane number is odd and greater than 1, only care
               about num_lanes-1 lanes
169         if num_lanes > 1 and num_lanes % 2 != 0:
170             num_lanes -= 1
171         elif num_lanes == 0:
172             num_lanes = 1
173 
174         setattr(r, "lanes", num_lanes)
```

```
175
176         # Lane boundaries
177         left_boundary = etree.SubElement(boundaries, "boundary")
178         left_boundary.set("type", "leftBoundary")
179
180         right_boundary = etree.SubElement(boundaries, "boundary")
181         right_boundary.set("type", "rightBoundary")
182
183         # Lane boundary geometries
184         leftb_geo = etree.SubElement(left_boundary, "geometry")
185         leftb_geo_ps = etree.SubElement(leftb_geo, "pointSet")
186
187         rightb_geo = etree.SubElement(right_boundary, "geometry")
188         rightb_geo_ps = etree.SubElement(rightb_geo, "pointSet")
189
190         nodes = []
191         for n in r.nodes:
192             nodes.append([n.lat, n.lng])
193
194         if num_lanes == 1:
195             left_boundary_points = find_parallel(r.nodes, True,
                    lane_width/2, lane_width/2)
196             right_boundary_points = find_parallel(r.nodes, False,
                    lane_width/2, lane_width/2)
197         else:
198             boundary_width = num_lanes/2.0 *lane_width
199             left_boundary_points = find_parallel(r.nodes, True,
                    boundary_width, boundary_width)
200             right_boundary_points = find_parallel(r.nodes, False,
                    boundary_width, boundary_width)
201
202         for i in range(len(r.nodes)):
203             # Left
204             lp = etree.SubElement(leftb_geo_ps, "point")
205             lp.set("x", format_coord(left_boundary_points[i][1]))
206             lp.set("y", format_coord(left_boundary_points[i][0]))
207             lp.set("z", format_coord(0.0))
208
209             # Right
210             rp = etree.SubElement(rightb_geo_ps, "point")
211             rp.set("x", format_coord(right_boundary_points[i][1]))
212             rp.set("y", format_coord(right_boundary_points[i][0]))
213             rp.set("z", format_coord(0.0))
214
215         # Center is supposed to store the reference line
216         # Left/right stores the borders of left/right lanes
```

```
217
218         center = etree.SubElement(laneSec, "center")
219         center_lane = etree.SubElement(center, "lane")
220
221         center_lane.set("id", "0")
222         center_lane.set("uid", "{}_0".format(r.id))
223         center_lane.set("type", "none")
224         #center_lane.set("direction", "bidirection")
225         #center_lane.set("turnType", "noTurn")   # Not sure what
               this means
226
227         center_border = etree.SubElement(center_lane, "border")
228         center_border.set("virtual", "FALSE")
229         cl_geo = etree.SubElement(center_border, "geometry")
230         cl_geo.set("sOffset", "0")
231         cl_geo.set("x", format_coord(r.nodes[0].lng))
232         cl_geo.set("y", format_coord(r.nodes[0].lat))
233         cl_geo.set("z", format_coord(0.0))
234         cl_geo.set("length", str(road_length(r.nodes)))
235
236         cborder_type = etree.SubElement(center_border, "borderType")
237         cborder_type.set("sOffset", "0")
238         cborder_type.set("type", "solid" if num_lanes == 1 else "
               broken")
239         cborder_type.set("color", "white" if num_lanes == 1 else "
               yellow")
240
241         cl_geo_ps = etree.SubElement(cl_geo, "pointSet")
242
243         center_nodes = nodes if num_lanes > 1 else find_parallel(r.
               nodes, True, lane_width/2.0, lane_width/2.0)
244         for n in center_nodes:
245             p = etree.SubElement(cl_geo_ps, "point")
246             p.set("x", format_coord(n[1]))
247             p.set("y", format_coord(n[0]))
248             p.set("z", format_coord(0.0))
249
250             # Check for min/max values:
251             # North
252             if max_coord[0] == None or max_coord[0] < n[0]:
253                 max_coord[0] = n[0]
254
255             # South
256             if max_coord[1] == None or max_coord[1] > n[0]:
257                 max_coord[1] = n[0]
258
```

```
259             # East
260             if max_coord[2] == None or max_coord[2] < n[1]:
261                 max_coord[2] = n[1]
262 
263             # West
264             if max_coord[3] == None or max_coord[3] > n[1]:
265                 max_coord[3] = n[1]
266 
267         right = etree.SubElement(laneSec, "right")
268 
269         if num_lanes > 1:
270             left = etree.SubElement(laneSec, "left")
271 
272         num_side_lanes = math.ceil(num_lanes/2)
273         for i in range(num_side_lanes):
274             # Right, only add this if num_lanes == 1
275             right_lane = etree.SubElement(right, "lane")
276             right_lane.set("id", "-{}".format(i+1))
277             right_lane.set("uid", "{}_1{}".format(r.id, i+1))
278             right_lane.set("type", "driving")
279             right_lane.set("direction", "bidirection" if num_lanes
                    == 1 else "forward")
280             right_lane.set("turnType", "noTurn")    # Not sure what
                    this means
281 
282             # Lane center
283             right_center = etree.SubElement(right_lane, "centerLine"
                    )
284 
285             center_pos = i*lane_width+(lane_width/2)
286             if num_lanes == 1:
287                 right_center_points = nodes
288             else:
289                 right_center_points = find_parallel(r.nodes, False,
                        center_pos, center_pos)
290 
291             rc_geo = etree.SubElement(right_center, "geometry")
292             rc_geo.set("sOffset", "0")
293             rc_geo.set("x", format_coord(right_center_points[0][1]))
294             rc_geo.set("y", format_coord(right_center_points[0][0]))
295             rc_geo.set("z", format_coord(0.0))
296             rc_geo.set("length", str(road_length(right_center_points
                    )))
297 
298             rc_geo_ps = etree.SubElement(rc_geo, "pointSet")
299 
```

```
300             for n in right_center_points:
301                 p = etree.SubElement(rc_geo_ps, "point")
302                 p.set("x", format_coord(n[1]))
303                 p.set("y", format_coord(n[0]))
304                 p.set("z", format_coord(0.0))
305
306             # Lane border
307             right_border = etree.SubElement(right_lane, "border")
308             right_border.set("virtual", "FALSE")      # "Identify
                    whether the lane boundary exists in real world"
309
310             rborder_type = etree.SubElement(right_border, "
                    borderType")
311             rborder_type.set("sOffset", "0")
312             rborder_type.set("type", "solid" if i == num_side_lanes
                    -1 else "broken")
313             rborder_type.set("color", "white")
314
315             if num_lanes == 1:
316                 right_border_points = find_parallel(r.nodes, False,
                        lane_width/2.0, lane_width/2.0)
317             else:
318                 right_border_points = find_parallel(r.nodes, False,
                        (i+1)*lane_width, (i+1)*lane_width)
319
320             rb_geo = etree.SubElement(right_border, "geometry")
321             rb_geo.set("sOffset", "0")
322             rb_geo.set("x", format_coord(right_border_points[0][1]))
323             rb_geo.set("y", format_coord(right_border_points[0][0]))
324             rb_geo.set("z", format_coord(0.0))
325             rb_geo.set("length", str(road_length(right_border_points
                    )))
326
327             rb_geo_ps = etree.SubElement(rb_geo, "pointSet")
328
329             for n in right_border_points:
330                 p = etree.SubElement(rb_geo_ps, "point")
331                 p.set("x", format_coord(n[1]))
332                 p.set("y", format_coord(n[0]))
333                 p.set("z", format_coord(0.0))
334
335             if num_lanes > 1:
336                 left_lane = etree.SubElement(left, "lane")
337                 left_lane.set("id", "{}".format(i+1))
338                 left_lane.set("uid", "{}_0{}".format(r.id, i+1))
339                 left_lane.set("type", "driving")
```

```
340                 left_lane.set("direction", "backward")
341                 left_lane.set("turnType", "noTurn")    # Not sure
                        what this means
342
343                 # Lane center
344                 left_center = etree.SubElement(left_lane, "
                        centerLine")
345
346                 left_center_points = find_parallel(r.nodes, True,
                        center_pos, center_pos)
347
348                 lc_geo = etree.SubElement(left_center, "geometry")
349                 lc_geo.set("sOffset", "0")
350                 lc_geo.set("x", format_coord(left_center_points
                        [0][1]))
351                 lc_geo.set("y", format_coord(left_center_points
                        [0][0]))
352                 lc_geo.set("z", format_coord(0.0))
353                 lc_geo.set("length", str(road_length(
                        left_center_points)))
354
355                 lc_geo_ps = etree.SubElement(lc_geo, "pointSet")
356
357                 for n in left_center_points:
358                     p = etree.SubElement(lc_geo_ps, "point")
359                     p.set("x", format_coord(n[1]))
360                     p.set("y", format_coord(n[0]))
361                     p.set("z", format_coord(0.0))
362
363                 # Lane border
364                 left_border = etree.SubElement(left_lane, "border")
365                 left_border.set("virtual", "FALSE")     # "Identify
                        whether the lane boundary exists in real world"
366
367                 lborder_type = etree.SubElement(left_border, "
                        borderType")
368                 lborder_type.set("sOffset", "0")
369                 lborder_type.set("type", "solid" if i ==
                        num_side_lanes-1 else "broken")
370                 lborder_type.set("color", "white")
371
372                 left_border_points = find_parallel(r.nodes, True, (i
                        +1)*lane_width, (i+1)*lane_width)
373
374                 lb_geo = etree.SubElement(left_border, "geometry")
375                 lb_geo.set("sOffset", "0")
```

```
376                    lb_geo.set("x", format_coord(left_border_points
                           [0][1]))
377                    lb_geo.set("y", format_coord(left_border_points
                           [0][0]))
378                    lb_geo.set("z", format_coord(0.0))
379                    lb_geo.set("length", str(road_length(
                           left_border_points)))
380
381                    lb_geo_ps = etree.SubElement(lb_geo, "pointSet")
382
383                    for n in left_border_points:
384                        p = etree.SubElement(lb_geo_ps, "point")
385                        p.set("x", format_coord(n[1]))
386                        p.set("y", format_coord(n[0]))
387                        p.set("z", format_coord(0.0))
388
389                # Sample Associations
390                # Distance from the center to the edges
391                right_sample = etree.SubElement(right_lane, "
                       sampleAssociates")
392                right_road_sample = etree.SubElement(right_lane, "
                       roadSampleAssociations")
393
394                if num_lanes > 1:
395                    left_sample = etree.SubElement(left_lane, "
                           sampleAssociates")
396                    left_road_sample = etree.SubElement(left_lane, "
                           roadSampleAssociations")
397
398                road_len = road_length(r.nodes)
399                for s in range(len(r.nodes)):
400                    s_pos = road_len/len(r.nodes) * s
401                    right_samp = etree.SubElement(right_sample, "
                           sampleAssociate")
402                    right_samp.set("sOffset", str(int(s_pos)))
403                    right_samp.set("leftWidth", str(lane_width/2))
404                    right_samp.set("rightWidth", str(lane_width/2))
405
406                    far_boundary = (num_lanes//2 + i + 0.5) * lane_width
407                    close_boundary = (math.ceil(num_lanes/2) - (i + 1) +
                            0.5) * lane_width
408                    right_road_samp = etree.SubElement(right_road_sample
                           , "sampleAssociation")
409                    right_road_samp.set("sOffset", str(int(s_pos)))
410                    right_road_samp.set("leftWidth", str(far_boundary))
```

```
411                 right_road_samp.set("rightWidth", str(close_boundary
                       ))
412
413                 if num_lanes > 1:
414                     left_samp = etree.SubElement(left_sample, "
                           sampleAssociate")
415                     left_samp.set("sOffset", str(int(s_pos)))
416                     left_samp.set("leftWidth", str(lane_width/2))
417                     left_samp.set("rightWidth", str(lane_width/2))
418
419                     left_road_samp = etree.SubElement(
                           left_road_sample, "sampleAssociation")
420                     left_road_samp.set("sOffset", str(int(s_pos)))
421                     left_road_samp.set("leftWidth", str(
                           close_boundary))
422                     left_road_samp.set("rightWidth", str(
                           far_boundary))
423
424         # Junctions
425         # OSM draws junctions as a shared node between ways
426
427         for road in roads:
428             if r == road:
429                 continue
430
431             for n in r.nodes:
432                 if n in road.nodes:
433                     if n not in junctions.keys():
434                         junctions[n] = set([])
435                     junctions[n].update([r, road])
436
437     for i, j in tqdm(enumerate(junctions.keys()), "Generating␣
           junctions", len(junctions)):
438         junc = etree.SubElement(root, "junction")
439         junc.set("id", str(i))
440
441         junc_outline = etree.SubElement(junc, "outline")
442         point = np.array(utm.from_latlon(j.lat, j.lng)[0:2])
443         v = np.array([2*lane_width, 0])
444         # This is probably a bit unnecessary as its the same in
               every iteration
445         outline = list(map(lambda x: rotate_vector(x[1], x[0]*math.
               pi/2), enumerate([v]*4))) + point
446         for c in outline:
447             p = utm.to_latlon(c[0], c[1], utmz["zone"], utmz["letter
                   "])
```

```
448                 cb = etree.SubElement(junc_outline, "cornerGlobal")
449                 cb.set("x", format_coord(p[1]))
450                 cb.set("y", format_coord(p[0]))
451                 cb.set("z", format_coord(0.0))
452 
453             # Generate connecting roads
454             vecs = []
455             for r in junctions[j]:
456                 n_i = r.nodes.index(j)
457                 p = utm.from_latlon(j.lat, j.lng)
458                 if n_i == 0:
459                     p2 = utm.from_latlon(r.nodes[1].lat, r.nodes[1].lng)
460                     vecs.append({"vec" : np.array([p2[0]-p[0], p2[1]-p
                        [1]]), "road" : r , "pos" : -2})
461                 elif n_i == len(r.nodes)-1:
462                     p2 = utm.from_latlon(r.nodes[-2].lat, r.nodes[-2].
                        lng)
463                     vecs.append({"vec" : np.array([p2[0]-p[0], p2[1]-p
                        [1]]), "road" : r , "pos" : 2})
464                 else:
465                     p0 = utm.from_latlon(r.nodes[n_i-1].lat, r.nodes[n_i
                        -1].lng)
466                     p1 = utm.from_latlon(r.nodes[n_i+1].lat, r.nodes[n_i
                        +1].lng)
467 
468                     vecs.append({"vec" : np.array([p1[0]-p[0], p1[1]-p
                        [1]]), "road" : r , "pos" : 1})
469                     vecs.append({"vec" : np.array([p0[0]-p[0], p0[1]-p
                        [1]]), "road" : r , "pos" : -1})
470 
471             for v in vecs:
472                 v["vec"] = v["vec"]*lane_width/np.linalg.norm(v["vec"])
473 
474             conn_roads = []
475             v = 0
476             while v  < len(vecs)-1:
477                 v2 = v+1
478                 while v2 < len(vecs):
479                     v3 = vecs[v2]["vec"]-vecs[v]["vec"]
480                     if v3[0] != 0 and v3[1] != 0:
481                         conn_roads.append({"vecs" : [vecs[v]["vec"], v3
                            ], "start" : {"road" : vecs[v]["road"], "pos"
                             : vecs[v]["pos"]}, "end" : {"road" : vecs[v2
                            ]["road"], "pos" : vecs[v2]["pos"]}})
482                     v2 += 1
483                 v += 1
```

```
484
485         conns = 0
486         for r_id, r in enumerate(conn_roads):
487             if r["start"]["road"] == r["end"]["road"]:
488                 continue
489             road = etree.SubElement(root, "road")
490             road.set("name", "connroad")
491             road_id = "conn_{}_{}".format(i, str(r_id))
492             road.set("id", road_id)
493             road.set("junction", str(i))
494
495             points = [point+r["vecs"][0], point+r["vecs"][0]+r["vecs
                    "][1]]
496
497             # Make curve
498             center = np.array(utm.from_latlon(j.lat, j.lng)[:2])
499             points = list(map(lambda x: points[0] + x, np.array(
                    make_curve(np.array([0, 0]), center - points[0],
                    points[1] - points[0]))))
500             points = list(map(lambda x: utm.to_latlon(x[0], x[1],
                    utmz["zone"], utmz["letter"]), points))
501
502             road_geo_link = etree.SubElement(road, "link")
503             rgl_pre = etree.SubElement(road_geo_link, "predecessor")
504             rgl_pre.set("elementType", "road")
505             rgl_pre.set("elementId", r["start"]["road"].id)
506             rgl_pre.set("contactPoint", "start" if r["start"]["pos"]
                     < 0 else "end")
507
508             rgl_succ = etree.SubElement(road_geo_link, "successor")
509             rgl_succ.set("elementType", "road")
510             rgl_succ.set("elementId", r["end"]["road"].id)
511             rgl_succ.set("contactPoint", "end" if r["start"]["pos"]
                    > 0 else "start")
512
513             conn_link = etree.Element("link")
514             start_road = root.xpath("//road[@id = '{}']".format(r["
                    start"]["road"].id))[0]
515             if abs(r["start"]["pos"]) > 1:
516                 start_link = start_road.findall("link")
517                 if len(start_road.findall("link")) == 0:
518                     start_link = etree.SubElement(start_road, "link"
                            )
519                 else:
520                     start_link = start_link[0]
521                 if abs(r["start"]["pos"]) > 0:
```

```
522                 succ = etree.SubElement(start_link, "successor" if r["start"]["pos"] < 0 else "predecessor")
523                 succ.set("elementType", "road")
524                 succ.set("elementId", road_id)
525                 succ.set("contactPoint", "start")
526 
527             sr_lanes = start_road.xpath(".//left/lane | .//right/lane")
528             for lane in sr_lanes:
529                 sr_lane_link = lane.findall("link")
530                 if len(sr_lane_link) == 0:
531                     sr_lane_link = etree.SubElement(lane, "link")
532                 else:
533                     sr_lane_link = sr_lane_link[0]
534                 sr_link = etree.SubElement(sr_lane_link, "successor" if r["start"]["pos"] < 0 else "predecessor")
535                 sr_link.set("id", "{}_11".format(road_id))
536 
537         sr_lanes = start_road.xpath(".//left/lane | .//right/lane")
538         for lane in sr_lanes:
539             conn_sr_link = etree.SubElement(conn_link, "predecessor")
540             conn_sr_link.set("id", lane.get("uid"))
541 
542         end_road = root.xpath("//road[@id = '{}']".format(r["end"]["road"].id))[0]
543         if abs(r["end"]["pos"]) > 1:
544             end_link = end_road.findall("link")
545             if len(end_road.findall("link")) == 0:
546                 end_link = etree.SubElement(end_road, "link")
547             else:
548                 end_link = end_link[0]
549             if abs(r["end"]["pos"]) > 0:
550                 succ = etree.SubElement(end_link, "predecessor" if r["end"]["pos"] > 0 else "successor")
551                 succ.set("elementType", "road")
552                 succ.set("elementId", road_id)
553                 succ.set("contactPoint", "end")
554 
555             er_lanes = end_road.xpath(".//left/lane | .//right/lane")
556             for lane in er_lanes:
557                 er_lane_link = lane.findall("link")
```

```
558                 if len(er_lane_link) == 0:
559                     er_lane_link = etree.SubElement(lane, "link")
560                 else:
561                     er_lane_link = er_lane_link[0]
562                 er_link = etree.SubElement(er_lane_link, "predecessor" if r["end"]["pos"] > 0 else "successor")
563                 er_link.set("id", "{}_11".format(road_id))
564 
565         er_lanes = end_road.xpath(".//left/lane | .//right/lane")
566         for lane in er_lanes:
567                 conn_er_link = etree.SubElement(conn_link, "successor")
568                 conn_er_link.set("id", lane.get("uid"))
569 
570         lanes = etree.SubElement(road, "lanes")
571         lane_sec = etree.SubElement(lanes, "laneSection")
572         lane_sec.set("singleSide", "true")
573 
574         start_width = lane_width
575         end_width = lane_width
576         if hasattr(r["start"]["road"], "name"):
577             start_name = r["start"]["road"].name.lower().replace(" ", "")
578             if conf != None:
579                 if start_name in conf.keys():
580                     if conf[start_name][1] != None:
581                         start_width = conf[start_name][1]
582 
583         if hasattr(r["end"]["road"], "name") :
584             end_name = r["end"]["road"].name.lower().replace(" ", "")
585             if conf != None:
586                 if end_name in conf.keys():
587                     if conf[end_name][1] != None:
588                         end_width = conf[end_name][1]
589 
590         left_boundary_points = find_parallel(points, True, start_width/2, end_width/2)
591         right_boundary_points = find_parallel(points, False, start_width/2, end_width/2)
592 
593         ls_right_boundary = etree.SubElement(lane_sec, "boundaries")
```

```
594                 ls_right_boundary.set("type", "rightBoundary")
595 
596                 ls_right_boundary_geo = etree.SubElement(
                        ls_right_boundary, "geometry")
597                 lslb_geo_ps = etree.SubElement(ls_right_boundary_geo, "
                        pointSet")
598 
599                 for p in right_boundary_points:
600                     ps_point = etree.SubElement(lslb_geo_ps, "point")
601                     ps_point.set("x", format_coord(p[1]))
602                     ps_point.set("y", format_coord(p[0]))
603                     ps_point.set("z", format_coord(0.0))
604 
605                 ls_left_boundary = etree.SubElement(lane_sec, "
                        boundaries")
606                 ls_left_boundary.set("type", "leftBoundary")
607 
608                 ls_left_boundary_geo = etree.SubElement(ls_left_boundary
                        , "geometry")
609                 lslb_geo_ps = etree.SubElement(ls_left_boundary_geo, "
                        pointSet")
610 
611                 for p in left_boundary_points:
612                     ps_point = etree.SubElement(lslb_geo_ps, "point")
613                     ps_point.set("x", format_coord(p[1]))
614                     ps_point.set("y", format_coord(p[0]))
615                     ps_point.set("z", format_coord(0.0))
616 
617                 center = etree.SubElement(lane_sec, "center")
618                 center_lane = etree.SubElement(center, "lane")
619                 center_lane.set("id", str(0))
620                 center_lane.set("uid", "{}_0".format(road_id))
621                 center_lane.set("type", "none")
622 
623                 center_lane_border = etree.SubElement(center_lane, "
                        border")
624                 center_lane_border.set("virtual", "TRUE")
625 
626                 cborder_type = etree.SubElement(center_lane_border, "
                        borderType")
627                 cborder_type.set("sOffset", "0")
628                 cborder_type.set("type", "none")
629                 cborder_type.set("color", "none")
630 
631                 center_geo = etree.SubElement(center_lane_border, "
                        geometry")
```

```
632                 center_geo.set("sOffset", str(0))
633                 center_geo.set("x", format_coord(left_boundary_points
                        [0][1]))
634                 center_geo.set("y", format_coord(left_boundary_points
                        [0][0]))
635                 center_geo.set("z", format_coord(0.0))
636                 center_geo.set("length", str(road_length(
                        left_boundary_points)))
637
638                 center_geo_ps = etree.SubElement(center_geo, "pointSet")
639
640                 for p in left_boundary_points:
641                     cg_point = etree.SubElement(center_geo_ps, "point")
642                     cg_point.set("x", format_coord(p[1]))
643                     cg_point.set("y", format_coord(p[0]))
644                     cg_point.set("z", format_coord(0.0))
645
646                 right = etree.SubElement(lane_sec, "right")
647                 right_lane = etree.SubElement(right, "lane")
648                 right_lane.set("id", str(-1))
649                 right_lane.set("uid", "{}_11".format(road_id))
650                 right_lane.set("type", "driving")
651                 right_lane.set("direction", "bidirection")
652                 right_lane.set("turnType", "noTurn")
653
654                 right_lane.insert(1, conn_link)
655
656                 right_lane_cl = etree.SubElement(right_lane, "centerLine
                        ")
657
658                 right_geo = etree.SubElement(right_lane_cl, "geometry")
659                 right_geo.set("sOffset", str(0))
660                 right_geo.set("x", format_coord(points[0][1]))
661                 right_geo.set("y", format_coord(points[0][0]))
662                 right_geo.set("z", format_coord(0.0))
663                 right_geo.set("length", str(road_length(points)))
664
665                 right_geo_ps = etree.SubElement(right_geo, "pointSet")
666
667                 for p in points:
668                     rg_point = etree.SubElement(right_geo_ps, "point")
669                     rg_point.set("x", format_coord(p[1]))
670                     rg_point.set("y", format_coord(p[0]))
671                     rg_point.set("z", format_coord(0.0))
672
```

```
673                 right_lane_border = etree.SubElement(right_lane, "border
                        ")
674                 right_lane_border.set("virtual", "TRUE")
675 
676                 rborder_type = etree.SubElement(right_lane_border, "
                        borderType")
677                 rborder_type.set("sOffset", "0")
678                 rborder_type.set("type", "none")
679                 rborder_type.set("color", "none")
680 
681                 right_border_geo = etree.SubElement(right_lane_border, "
                        geometry")
682                 right_border_geo.set("sOffset", str(0))
683                 right_border_geo.set("x", format_coord(
                        right_boundary_points[0][1]))
684                 right_border_geo.set("y", format_coord(
                        right_boundary_points[0][0]))
685                 right_border_geo.set("z", format_coord(0.0))
686                 right_border_geo.set("length", str(road_length(
                        right_boundary_points)))
687 
688                 right_border_geo_ps = etree.SubElement(right_border_geo,
                         "pointSet")
689 
690                 for p in right_boundary_points:
691                     rg_point = etree.SubElement(right_border_geo_ps, "
                            point")
692                     rg_point.set("x", format_coord(p[1]))
693                     rg_point.set("y", format_coord(p[0]))
694                     rg_point.set("z", format_coord(0.0))
695 
696                 right_sample = etree.SubElement(right_lane, "
                        sampleAssociates")
697                 right_road_sample = etree.SubElement(right_lane, "
                        roadSampleAssociations")
698 
699                 road_len = road_length(points)
700                 for s in range(len(points)):
701                     s_pos = road_len/len(points) * s
702                     right_samp = etree.SubElement(right_sample, "
                            sampleAssociate")
703                     right_samp.set("sOffset", str(int(s_pos)))
704                     right_samp.set("leftWidth", str(lane_width/2))
705                     right_samp.set("rightWidth", str(lane_width/2))
706 
```

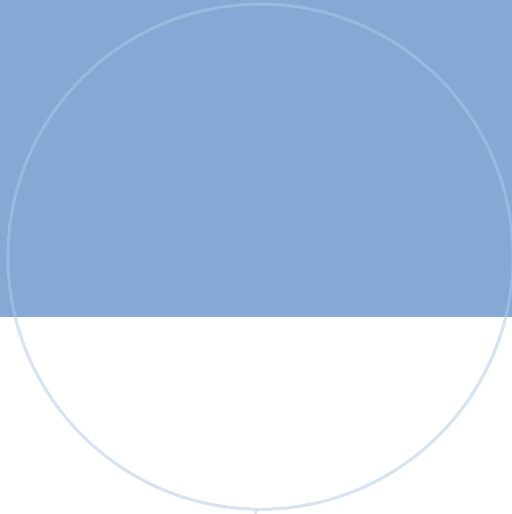
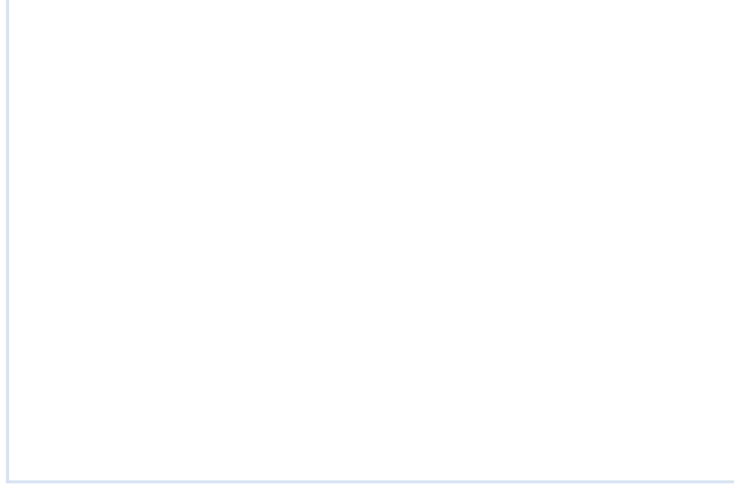
```
707                     right_road_samp = etree.SubElement(right_road_sample
                            , "sampleAssociation")
708                     right_road_samp.set("sOffset", str(int(s_pos)))
709                     right_road_samp.set("leftWidth", str(lane_width/2))
710                     right_road_samp.set("rightWidth", str(lane_width/2))
711
712                 conn = etree.SubElement(junc, "connection")
713                 conn.set("id", str(conns))
714                 conn.set("incomingRoad", str(r["start"]["road"].id))
715                 conn.set("connectingRoad", str(road_id))
716                 conn.set("contactPoint", "start")
717
718                 for l in range(math.ceil(r["start"]["road"].lanes/2)):
719                     conn_s_link = etree.SubElement(conn, "laneLink")
720                     conn_s_link.set("from", str(-(l+1)))
721                     conn_s_link.set("to", str(-1))
722
723                     if r["start"]["road"].lanes > 1:
724                         conn_sl_link = etree.SubElement(conn, "laneLink"
                                )
725                         conn_sl_link.set("from", str(l+1))
726                         conn_sl_link.set("to", str(-1))
727
728                 conns += 1
729
730                 conn = etree.SubElement(junc, "connection")
731                 conn.set("id", str(conns))
732                 conn.set("incomingRoad", str(r["end"]["road"].id))
733                 conn.set("connectingRoad", str(road_id))
734                 conn.set("contactPoint", "end")
735
736                 for l in range(math.ceil(r["end"]["road"].lanes/2)):
737                     conn_s_link = etree.SubElement(conn, "laneLink")
738                     conn_s_link.set("from", str(-(l+1)))
739                     conn_s_link.set("to", str(-1))
740
741                     if r["end"]["road"].lanes > 1:
742                         conn_sl_link = etree.SubElement(conn, "laneLink"
                                )
743                         conn_sl_link.set("from", str(l+1))
744                         conn_sl_link.set("to", str(-1))
745
746                 conns += 1
747
748     header.set("north", format_coord(max_coord[0]))
749     header.set("south", format_coord(max_coord[1]))
```

```
750     header.set("east", format_coord(max_coord[2]))
751     header.set("west", format_coord(max_coord[3]))
752 
753     print("XML successfully generated, writing to '{}'".format(
            filename))
754 
755     tree.write(filename, xml_declaration=True, pretty_print=pretty,
            encoding='UTF-8')
756 
757 # Calculate road length
758 def road_length(road):
759     length = 0
760     for i in range(len(road)-1):
761         p1 = road[i]
762         p2 = road[i+1]
763 
764         if isinstance(p1, Node):
765             length += distance.distance((p1.lat, p1.lng), (p2.lat,
                    p2.lng)).m
766         else:
767             length += distance.distance(p1, p2).m
768 
769     return length
770 
771 def vector_angle(v1, v2):
772     v1_u = v1 / np.linalg.norm(v1)
773     v2_u = v2 / np.linalg.norm(v2)
774 
775     dot = np.dot(v1_u, v2_u)
776     det = v1_u[0]*v2_u[1] - v1_u[1]*v2_u[0]
777 
778     return np.arctan2(det, dot)
779 
780 def rotate_vector(vector, angle):
781     v_x = math.cos(angle) * vector[0] - math.sin(angle) * vector[1]
782     v_y = math.sin(angle) * vector[0] + math.cos(angle) * vector[1]
783     return [v_x, v_y]
784 
785 def curve (A, B, C, t):
786     P0 = A * t + (1 - t) * B
787     P1 = B * t + (1 - t) * C
788     return P0 * t + (1 - t) * P1
789 
790 def make_curve(p1, p2, p3):
791     crv_line = []
792     for x in np.linspace(0,1,10):
```

```
793            crv_line.append(curve(p1, p2, p3, x))
794        return crv_line
795
796
797    def find_parallel(road, left, start_width, end_width):
798        points = []
799
800        for n in road:
801            if isinstance(n, Node):
802                points.append((n.lat, n.lng))
803            else:
804                points.append(n)
805
806        points = np.array(points)
807        vectors = []
808
809        parallel = []
810        for i in range(len(points)-1):
811            # Convert the points to UTM coordinates
812            p1 = utm.from_latlon(*points[i])
813            p2 = utm.from_latlon(*points[i+1])
814
815            # Vector between the current and the next point
816            v = np.array([p2[0]-p1[0], p2[1]-p1[1]])
817
818            widths = np.linspace(0,1,len(points))[0:-1]
819            if i != 0:
820                # If the point is not the first or last point, use both
                       the previous and the next
821                # points to calculate the new point
822                p0 = utm.from_latlon(*points[i-1])
823                v0 = np.array([p1[0]-p0[0], p1[1]-p0[1]])
824
825                # Find  angle between vectors
826                angle = vector_angle(v0, v)
827                angle = math.pi + angle
828                angle = -angle/2.0
829
830                # Make a new point based on the second vector and the
                       calculated angle
831                lv = np.array(rotate_vector(v, angle))
832
833                width = start_width*widths[(i)] + end_width*widths[-(i)]
834
835                # Scale width to maintain the lane width at sharp angles
836                scaled_width = abs(width/np.sin(angle))
```

```
837         else:
838             # If the point is the first point, only use the next
                    point to calculate
839             lv = np.array([v[1], -v[0]])
840             scaled_width = end_width
841 
842         # Move the new point correctly represent the road's width
843         l = scaled_width*lv/np.linalg.norm(lv)
844         if left:
845             lp = (p1[0] - l[0], p1[1] - l[1])
846         else:
847             lp = (p1[0] + l[0], p1[1] + l[1])
848 
849         # Convert back to lat/long and append to the line
850         lp = utm.to_latlon(lp[0], lp[1], utmz["zone"], utmz["letter"
                ])
851         parallel.append(lp)
852 
853         # If this is the last iteration, add a point for the final
                point by using the two last points
854         if i == len(points)-2:
855             lv = np.array([-v[1], v[0]]) if left else np.array([v
                    [1], -v[0]])
856             lv = lv/np.linalg.norm(v)
857             l = start_width*lv/np.linalg.norm(lv)
858             lp = (p2[0] + l[0], p2[1] + l[1])
859 
860             lp = utm.to_latlon(lp[0], lp[1], utmz["zone"], utmz["
                    letter"])
861             parallel.append(lp)
862 
863     return parallel
864 
865 def main():
866     global utmz
867 
868     parser = argparse.ArgumentParser()
869 
870     parser.add_argument('file', help="Input␣filename")
871     parser.add_argument('-c', '--config', help="Manually␣set␣lane␣
            numbers␣and␣widths␣based␣on␣road␣names")
872     parser.add_argument('-z', '--zone', action="store", type=str,
            help="UTM␣zone,␣example:␣-z␣32V")
873     parser.add_argument('-p', '--pretty', action='store_true', help=
            "Prettify␣output")
874     parser.set_defaults(pretty=False)
```

```
875
876     args = parser.parse_args()
877
878     if args.file:
879         filename = args.file
880
881     if args.zone:
882         try:
883             gz = args.zone
884             utmz["zone"] = int(gz[0:-1])
885             utmz["letter"] = str(gz[-1]).upper()
886             utmz["full"] = gz
887
888             if utmz["zone"] > 60 or utmz["zone"] < 1:
889                 raise ValueError("Zone number out of range, must be 
                        between 1 and 60")
890
891             if not utmz["letter"].isalpha() or utmz["letter"] in ["A
                    ", "B", "Y", "Z"]:
892                 raise ValueError("Zone letter out of range, must be 
                        between C and X")
893
894         except (TypeError, ValueError) as e:
895             print("Erroneous UTM zone \"{}\", using default \"{}\"."
                    .format(args.zone, utmz["full"]))
896
897     if args.config:
898         conf = read_config(args.config)
899     else:
900         conf = None
901     nodes, roads = readOSM(filename)
902     buildXML(filename, roads, args.pretty, conf)
903
904 if __name__ == "__main__":
905     main()
```

NTNU
Norwegian University of
Science and Technology