Alma Mater Studiorum - Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica - Scienze e Ingegneria

Corso di Laurea in Ingegneria Informatica Magistrale

Digital Twins as Decision Support Systems for Sustainable Smart Cities: a Traffic Analysis Perspective

Candidate

Filippo Lenzi

Supervisor

Prof. Paolo Bellavista

Co-supervisor

Prof. Armir Bujari

Academic Year 2023-2024

Abstract

Digital Twin (DT) technology has revived the interest in the Smart City concept, allowing to build accurate digital models of complex phenomena, serving as decision-making support systems for different urban stakeholders. In this thesis, we focus on main DT building blocks allowing for efficient analysis of the urban environment. The technology could be employed as a simulation tool to analyze environmental impacts, exploring alternate scenarios. Grounding the study, we explore the use of the technology via two use-cases: Acc2Twin for transport packaging and ParallelTwin for traffic analysis — focusing on road network partitioning. The Acc2Twin cases showcases a data processing pipeline aimed at building a continuous road network (path) topology, given in input to a first-principles DT, tasked with simulating stretch forces exerted on palletized products during transportation. Acc2Twin interpolates route data to an arbitrary precision to allow for the replication of real transport conditions. The second use-case, ParallelTwin, aims at optimizing parallel simulations of vehicular traffic with the SUMO software package, reducing simulation time, and enhancing system performance without altering the core pre-existing simulation program. We report on ParallelTwin's framework, operational mechanisms, communication protocols, and the METIS algorithm for efficient graph partitioning, researching efficient partitioning weights to obtain a balanced simulation workload.

Acknowledgements

Acknowledgements are mostly written in the Italian language.

Ringraziamenti

Dedico questo spazio della tesi per ringraziare coloro che mi hanno supportato nella sua realizzazione, e nel mio percorso accademico e di vita prima di essa.

Prima di tutto ringrazio profondamente il mio relatore Prof. Bellavista, e il mio correlatore Prof. Bujari, per le conoscenze trasmesse durante tutto il percorso, per i loro consigli e per la loro grande disponibilità.

Ringrazio i miei genitori, che mi hanno sostenuto in tutto il percorso scolastico e di vita in ogni modo, e il resto della famiglia, per il supporto e l'affetto. In particolare Elena e Matteo, per avermi fatto sentire anziano in anticipo rispetto alla media, con grande affetto.

Un importante ringraziamento va ai miei amici, presenti e passati, senza i quali non sarei arrivato a questa pietra miliare con la compiutezza della mia personalità, seppur talvolta meno tranquillo; e molto affetto voglio dedicare alla "cugina" Chiara, per la sua sempre profonda comprensione e vicinanza.

I want to also thank Team Revelations, for giving me my first real experience in working on my passions in a project, and seeing amazing results from that endeavor.

E infine, ringrazio tutti i miei colleghi di corso, giocatori di ruolo, co-interpeti teatrali, compagni di scuola, e ogni altra persona che ha lasciato un pezzo di sé nella mia vita, costruendo ciò che sono diventato.

Nella speranza e sincera convinzione che i prossimi anni saranno come quelli trascorsi, o ancora meglio.

Contents

1 Introduction **7**

2 Technological Background **9**

2.1 Digital Twins . . . 9

2.1.1 Digital Twin Taxonomy . . . 10

2.1.2 Key Components . . . 10

2.1.3 Supporting Technologies for Digital Twins . . . 11

2.2 OpenStreetMap . . . 12

2.2.1 Overview . . . 12

2.2.2 Data structure . . . 13

2.2.3 Geographical data . . . 14

2.2.4 Data quality . . . 14

2.3 Microscopic traffic simulation: SUMO . . . 15

2.3.1 Creating a simulation . . . 16

2.3.2 Simulation output and emissions . . . 21

2.3.3 Vehicle behavior . . . 23

2.3.4 Interaction: Libsumo and TraCI . . . 23

3 Acc2Twin **25**

3.1 A Digital Twin for package manufacturing . . . 25

3.2 Package structure . . . 27

3.3 Navigation: OSRM . . . 27

3.4 Discrete road geometry interpolation . . . 29

3.4.1 Elevation data . . . 30

3.4.2 Segmented interpolation . . . 30

3.5 Support metadata . . . 36

3.6 Usage . . . 37

3.7 Processing steps following our work . . . 38

3.8 Experimental results . . . 39

4 ParallelTwin **42**

4.1 Goals 42

4.2 General overview 43

4.3 Inter-process interaction 43

4.4 Road graph partitioning 45

4.4.1 Usage 46

4.4.2 METIS algorithm 46

4.4.3 Partitioning weights 47

4.4.4 Partition processing and route cutting 48

4.5 Simulation runtime 49

4.6 Software structure 50

4.7 Test results 51

4.7.1 Partitions workload balance 51

4.7.2 Simulation times 55

5 Conclusions and future developments **60**

A Appendix - ParallelTwin Usage **62**

Bibliography **69**

List of Figures

2.1 First published figure of the Digital Twin model. 9
2.2 Digital Twin classification, as defined by [1]. 10
2.3 Trend of active mappers across recent months and years. 15
2.4 Sample screen from sumo-gui 16
2.5 Sample screen from netedit, inside the network editing window 17
2.6 Interface of the OSM Web Wizard 17
2.7 Sample screen from netedit, inside the demand (traffic) editing window 18
2.8 Sample of a SUMO demand file 19
2.9 A conceptual diagram of an origin-destination matrix. 20
2.10 Example Traffic Assignment Zone file 20
2.11 Example output of SUMO plotXMLAttributes.py from simulated Lane Area Detectors (vehicle cameras) 21
2.12 Example output of SUMO plot_net_dump.py with edges colored depending on simulated emissions 22

3.1 Representation of the pallet Digital Twin developed by Robopac/TechLab 25
3.2 Representation of the pipeline for the Robopac DT 26
3.3 OSRM demo website 28
3.4 Extract of a vehicle profile for OSRM (in Lua) 29
3.5 Examples of coarse grained curves in OSM maps 30
3.6 Average point distance in OSM data for the Emilia-Romagna region 31
3.7 Rendering of *aster30m* dataset. 32
3.8 Interpolation errors: jagged curves with linear interpolation (left), overshooting geometry in straighter roads with bezier interpolation (right). 32
3.9 Line-curve tagging simplified algorithm, in Python 33
3.10 Multi-line interpolation algorithm, in Python 34
3.11 Interpolation examples, with line segments colored depending on curve tagging. 35
3.12 Interpolation examples with sample points highlighted, with a sampling distance of 5 meters. 35
3.13 Example of elevation interpolation (represented in meters) 36
3.14 Example of further processing from our data. 39

3.15 Examples of route interpolation compared to satellite images from ESRI World Imagery . 40
3.16 Sample of the testing Web-UI 40

4.1 Representation of a sample partitioning output 46
4.2 Multilevel partitioning. Source: METIS manual [2] 47
4.3 Minimal example of a route passing multiple times from a partition. Regardless of the partitioning cut, the circular route will start and end in the same partition. 49
4.4 SUMO GUI with more partitions open 50
4.5 Area used in the partitioning tests 52
4.6 Visualization of best partitioning scheme (*route-num* + *connections*) during tests 53
4.7 Vehicle counts across simulation time steps in best partitioning scheme (*route-num* + *connections*) from tests 54
4.8 Simulation times across partitions from select partitioning schemes. Best: *route-num* + *connections* 2nd best: *route-num* + *connections* + *connexp* Middle: standalone *osm* 2nd best: standalone *connections* 54
4.9 Map and results of elapsed time testing with the bologna-sim simulation. 56
4.10 Map and results of elapsed time testing with the bologna-metropolitan-area simulation. . 57
4.11 Map and results of elapsed time testing with various SUMO netgenerate spider configurations. 58
4.12 Map and results of elapsed time testing with the grid_large simulation. 59

Chapter 1

Introduction

Research on climate-related phenomena has been a popular topic in the latest decades, for reasons that are easy to imagine. Waste and vehicle pollution are two important factors in this regard. In particular, with the steep increase in online delivery orders in part caused by the recent pandemic, the waste produced by packaging has seen an increase, with an average 188.7 kg of such waste per EU inhabitant in 2021, increasing by 24.2% from 2020 to 2021 [3].

Meanwhile, the pollution caused by vehicles accounts for a large part of the emissions that affect the climate: for example, the transportation sector accounts for about 29% of US greenhouse gas emissions [4].

These issues are of course worsened in urban areas, due to a much higher population density: vehicular traffic is one of the main source of pollutants in urban areas, including carbon monoxide (CO), carbon dioxide (CO2), volatile organic compounds (VOCs), nitrogen oxides (NOx), and particulate matter (PM) [5]. To enable an ever improving research on the problem, digital transformation is an important factor, allowing the efficient coordination and cooperation of the many stakeholders involved in an urban environment, and a better monitoring of the myriad of affecting factors.

This has made the concept of a smart city more important during this decade, especially due to the arrival and diffusion of IoT and AI technologies to both survey and process the vast amount of data related to urban problem solving. Many institutions, both private and public institutions have started devising innovative solutions to work on complex phenomena [6].

An important role in this front is played by Digital Twin (DT) technologies, which embody a digital replica of physical entities or systems, continuously updated through data streams from their real-world counterparts to aid in decision-making [7], offering both descriptive and predictive capabilities, and often control interfaces to the real entity being replicated. Digital Twins aren't a new concept, but recent advancements in connectivity, AI, and sensor technology have enabled further developments and usage, making it an important field of investment. For example, Grand View Research estimates the Digital Twin market to grow at a compound annual growth rate of 33.7% [8] from 2023 to 2030.

Many public and private institutions have recognized the value of the Digital Twin approach, and are taking part of the "twin transition". A considerable example is the EU's "Destination Earth"/"DestinE", a Digital Twin of the Earth on a global scale [9]. Many cities have developed or are developing an urban

DT to manage traffic, city development, and many other factors: among others some examples are Barcelona [10], Singapore [11] and Helsinki [12]. Finally, an important project in this regard is the EU-funded Change2Twin [13], founded with the purpose of helping manufacturing SMEs in the digitization process using DTs.

The scope of this thesis is to showcase two applications of the Digital Twin approach across varying scales of reference. The initial case involves a smaller scale, focusing on a DT designed for palletized packages aimed at optimizing transport packaging, developed as part of the goals of the Change2Twin project for the packaging company Robopac in its internal project Acc2Twin. This software takes as an initial input simply the origin and destination of the transported goods, and processes route data to calculate the forces the payload will endure in its road, using its Digital Twin to calculate the required packaging in film wrapping.

The subsequent case pertains to a larger scale, involving the use of the traffic simulation software SUMO as an urban digital twin. In particular, we focused on the parallelization of the software to make a more efficient use of system resources in large-scale simulations, focusing on optimizing the partitioning scheme of the road network to balance the workload across different processes.

Outline

The thesis is organized into five chapters. After this introductory chapter, Chapter 2 goes deeper into supporting technologies and the technological background of this thesis.

Chapter 3 details the first main study case of this thesis, the pipeline of the Acc2Twin project of the packaging manufacturing company Robopac, and in particular our work on route data processing as part of this project; while Chapter 4 details the second study case, thus our work to develop a parallel wrapper to the Eclipse SUMO traffic simulation software package.

Finally, Chapter 5 overviews the general conclusions of this thesis and possible future works and improvements.

Chapter 2

Technological Background

This chapter explains the technological background of this thesis, starting from an overview of the concept of Digital Twins, followed by OpenStreetMap, the main data source used in this thesis. Finally, we discuss SUMO, the traffic simulator core to our second study case in Chapter 4.

2.1 Digital Twins

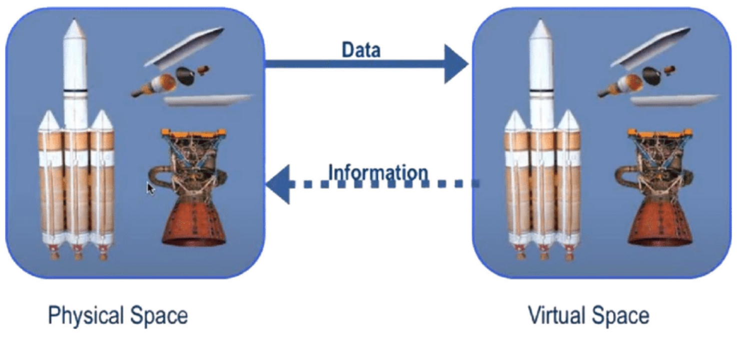


Figure 2.1: First published figure of the Digital Twin model.
Source: Grieves and Vickers; authors of [14].

Digital Twins (DTs) are an important driver of digital innovation, being a powerful tool for real-time analysis, monitoring and simulation of complex real-world physical systems, often used to examine urban settings as in [10], [12] and [11] as previously mentioned.

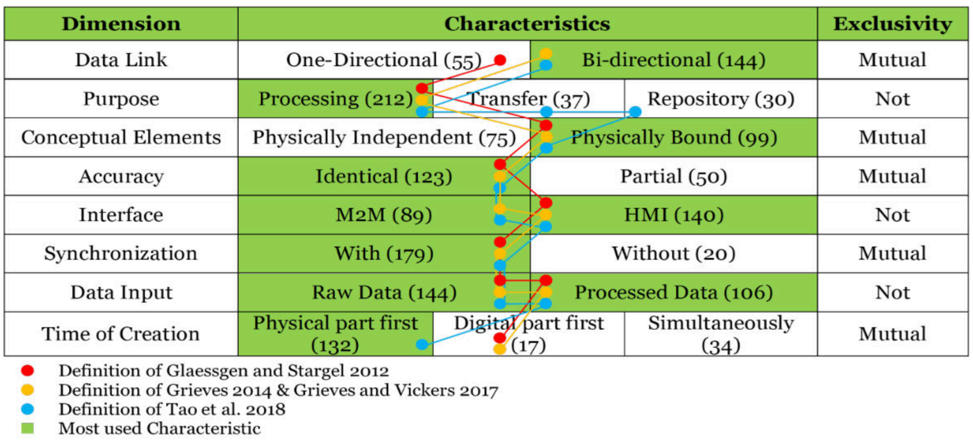
The term Digital Twin was coined in [14] and was used in the context of astronautics and aerospace, even if similar concepts weren't new. It was defined as the idea that a digital informational construct about a physical system can be created as its own entity, being linked to the physical system's information thorough its entire lifecycle.

The term has gained popularity in the recent years thanks to modern computing platforms and advancements in engineering and management methodologies, especially due to the rise of Internet of Things (IoT) technologies, distributed and cloud computing, and more generally a wider data availability and more possibilities for the communication of devices. There isn't a singular accepted modern definition of Digital Twins: they cannot be simply defined as "mirrors" of real world entities (products, systems, places), also called Physical Twins in this context, as some applications of this paradigm retroactively interact with real-world assets to implement strategies computed from obtained data autonomously [15].

2.1.1 Digital Twin Taxonomy

Currently there are many applications of DTs in different fields, such as smart cities as previously mentioned. Other applications include manufacturing (of which the study case detailed in Chapter 3 is a good example) and logistics [16]. While informal terms for various kinds of Digital Twin exist, such as Product Twin, Component Twin, System Twin, and others [17] [18] [19], a commonly accepted taxonomy doesn't yet exist.

The Digital Twins of interest to this thesis, too, are quite different in scale and scope: the first case regards a small-scale analysis of a palletized payload during transport, while the second is related to medium/large-scale simulation of traffic in the context of a smart city.



Dimension	Characteristics			Exclusivity
Data Link	One-Directional (55)	Bi-directional (144)		Mutual
Purpose	Processing (212)	Transfer (37)	Repository (30)	Not
Conceptual Elements	Physically Independent (75)	Physically Bound (99)		Mutual
Accuracy	Identical (123)	Partial (50)		Mutual
Interface	M2M (89)	HMI (140)		Not
Synchronization	With (179)	Without (20)		Mutual
Data Input	Raw Data (144)	Processed Data (106)		Not
Time of Creation	Physical part first (132)	Digital part first (17)	Simultaneously (34)	Mutual

🔴 Definition of Glaessgen and Stargel 2012
🟡 Definition of Grieves 2014 & Grieves and Vickers 2017
🔵 Definition of Tao et al. 2018
🟩 Most used Characteristic

Figure 2.2: Digital Twin classification, as defined by [1].
Source: [1]

Various classifications for DTs exist. The first were provided by [7], classifying the domains of various Digital Twin applications, first emphasizing the importance of DTs in Information Systems research. In particular, they classify Digital Twins in six dimensions, that represent various properties of relevance such as their industrial sector and purpose, as part of a comprehensive literature review. Another classification is provided by [1], in a methodical and empirical multi-dimensional taxonomy approach, resulting in seven dimension depending on several meta-characteristics (as shown in figure 2.2).

Finally, to support the Change2Twin project, a newer taxonomy of DTs is currently in development [20], structured in two dimension with various axes for the purpose of straightforward presentation to end clients (Small/Medium Enterprises).

2.1.2 Key Components

Even lacking a standard approach to the creation of a Digital Twin system, some key components are normally needed for their operation.

Data source A Digital Twin requires a reliable source of real-world data, to allow for a realistic pairing to the real entity. The type and update frequency of the data depends on the usecase: the DT might be linked to a continuous, real-time data source on the Physical Twin, such as sensors on a product or a IoT network on an urban scale; or have more granular or even single-time samplings that are then used to analyze data in simulations. Both of our study cases fall in the latter category, sampling data of the real entity (traffic patterns or 3D model of the payload) sparingly and using it to compute and process useful results. The main data source for this thesis was the OpenStreetMap dataset, detailed in Section 2.2.

Model The Twin can involve various computational and representational models related to its real-world counterpart. These models can vary, including first-principle models based on physical laws, data-driven models that depend on continuous data gathering, geometrical and material models like Computer-Aided Design/Engineering (CAD, CAE), or visualization-focused ones like mixed-reality. The first two will be reviewed in more detail in the next section.

Interaction and APIs A Digital Twin needs to interact with other digital components, such as other twins, external programs or an operator. APIs must be made available to facilitate these operations, requiring at least access to the data but often the possibility of altering the operating process.

2.1.3 Supporting Technologies for Digital Twins

For additional context, especially in relation to our work in Chapter 4, we will review some commonly required technologies in Digital Twin implementations, referencing part of the review done in [20].

Specifically, we'll review the following technologies:

- Physics-based modelling
- Data-driven modelling
- Infrastructure and platforms

Physics-based modelling

Physical models are one of the more frequent types of model used in DTs, as mentioned earlier. Many SMEs already possess product models, serving as an useful starting point for Digital Twin setups. These existing models are usually paired with 3D models of the Physical Twin, to apply the physical model to it.

A common usecase of this kind of model is analysis of the interactions between the PT and the environment, which requires more in-depth representations of the real-object than standard 3D models, which usually only possess surface-level details without information about the interior.

Both of our cases of interest fall into this category: as will be detailed in Chapter 3, the palletized load DT applies physical formulas on the model of the real payload, to aid in the decision process of packaging application, while SUMO, used in the second usecase, applies a variety of models to its simulated vehicles to compute vehicle following, lane changing, and a variety of behaviors.

Data-driven modelling

The other main kind of model used in DTs is measuring various properties of the Physical Twin in real-time and applying data-driven modelling methods. This has the advantage of not requiring knowledge of the physical laws affecting the real system or object, but does require large amounts of data. IoT is a potential source of this kind of data, especially in urban or large-scale environments, or industrial environments with Industrial Internet of Things (IIoT), more focused on accuracy than the consumer market.

Having such a vast array of sensors produces too much data to be analyzed manually, which requires the use of data analysis technologies, an important one being artificial intelligence, with the recent and ever growing advances in machine learning. Hybrid models also exist, using a mix of data-driven and physical approaches to compensate for lower quality or quantity of data.

Infrastructure and platforms

Digital Twins require significant computational power. One of the more important technologies that provides this is cloud computing, which provides remote access to processing resources. This setup grants scalability without businesses needing to set up their own infrastructure, catering to varying needs in different phases of Digital Twin setups. In contrast, edge computing brings processing closer to data sources, reducing latency by handling data locally, ensuring privacy and security, but often requiring hefty local hardware investments. Fog computing combines edge and cloud strengths, optimizing data processing, while high performance computing (HPC), unlike cloud services, tackles immensely complex problems, relying on remote resources and queuing systems for job processing.

In all of these cases, the possibility of partitioning and parallelizing the workload related to a DT is often necessary to be able to properly distribute the task among the various computing nodes. This is the main motivation behind the work of Chapter 4, as being able to handle simulations in parallel is necessary in a modern computing environment.

2.2 OpenStreetMap

In this section we review the OpenStreetMap dataset, used in both our study cases.

2.2.1 Overview

OpenStreetMap [21] (or **OSM**) is the most widely used open source map dataset, containing information on both road geometry and metadata. It is community-driven, with many contributors providing and updating the OSM database with information, and verifying other contributions. The project is supported by the OpenStreetMap Foundation non profit [22], but any user from the internet can contribute to the dataset.

OSM is supported by a great amount of services and applications; for the purpose of this thesis, we used **Nominatim** and **OSRM**. *Nominatim* is a service that solves requests for location data from its name, for example retrieving coordinates from a city name, a process known as geocoding. OSRM is a route navigation service, which will be detailed in Chapter 3.

OSM logo

Due to the large amount of data (100GB compressed at the time of writing), regional and partial downloads are available from various sources. For this thesis, we mainly used the Overpass API, a public read-only API made to retrieve data of OSM elements from their coordinates, and the binary `osm.pbf` region map files made available from *Geofabrik* at download.geofabrik.de.

Contribution is possible by using one of a variety of editors, such as JOSM and Rapid on desktop, or Vespucci and Osm Go! on mobile [23]. Contributors upload changes as a *changeset*, and uploaded changes are immediately reflected on the database - data accuracy and verification is left to local mappers, editors, and the community. Despite this, OSM map accuracy is usually good enough for most usecases, and the crowd-sourcing of data in some cases leads to more up-to-date geographical information even when compared to standard commercial sources [24]. However, this does depend on the area of reference and the amount of active mappers therein, with for example a considerable heterogeneity in data accuracy between italian regions [25].

2.2.2 Data structure

OSM data is made of five kinds of elements:

- **Nodes**: point data, marking locations. Can be connected to other nodes. They contain geographical coordinates for that point.
- **Ways**: connected lines of nodes with their own properties. They define roads, paths, rivers, and so on.
- **Closed ways**: Ways forming a closed loop, usually roundabouts or area delimiters.
- **Areas**: Filled closed ways.
- **Relations**: More complex shapes, or elements related to each other but not connected. Can represent political entities such as regions and provinces.

An important point to note is that nodes and ways aren't analogous to nodes and edges in a graph: a Way is comprised of one or more edges between nodes. This was an important difference to consider during the development of the software for the first study case detailed in Ch. 3.

All elements have *tags*, which are key-value pairs containing the element's information. Tags can be arbitrarily created, but usually have standardized keys and often values. Some examples of tags include:

- `name`: the name of the map element, if any.
- `highway`: identifies the road kind, values include *primary*, *secondary*, *motorway*, *residential*, etc. These values are global, and thus do not necessarily follow legal definitions of roads in a specific nations, though that is often the case. Nodes can contain highway tags, in some cases such as smaller roundabouts.
- `place`: defines the kind of "place" the element represents: examples include *city*, *town*, *island*.
- `maxspeed`: defines the maximum speed limit for a particular road or area, by default as kilometers per hour. An important point is that this tag often isn't present, in which case the default value for the road type in the specific country should be assumed.

A full list of tag examples is included in the OSM wiki at wiki.openstreetmap.org, which has pages for the standard tag keys.

2.2.3 Geographical data

Nodes are the main OSM element containing geographical data, in the form of a point as defined by longitude and latitude. The shape of a road represented by a Way is thus simply the line comprised of the segments connecting its points, without specifying any additional data. Curves are not naturally represented in OSM, instead being a series of close points which is good enough in a consumer navigation usecase.

While OSM does define an elevation tag `ele`, it is only used in 0.08% of elements, specifically in 1.04% of Nodes, 0.57% of Ways, and 1.58% of Relations [26] at the time of writing. Thus, in usecases where elevation is of interest this datum will need to be obtained from other services, such as *OpenTopoData* and *Google Elevation API*, which is what we did for our first study case in Chapter 3.

2.2.4 Data quality

As a collaborative project, the quality of OSM data can vary depending on the location and the level of involvement of local mappers. As previously mentioned, the quality and accuracy of the data depends on the area of interest [24] [25].

The main challenge of OSM data is its reliance on volunteers to collect and update information. This can lead to data inconsistencies, as different mappers may have different levels of expertise and may use different methods for collecting data, such as on-site, aerial photography, comparison with existing sources, etc. Additionally, OSM data is not always as up-to-date as commercial sources, as it can take time for volunteers to make changes to the map: as already mentioned in [24], the inverse can be true too, with OSM data being updated earlier than commercial sources.

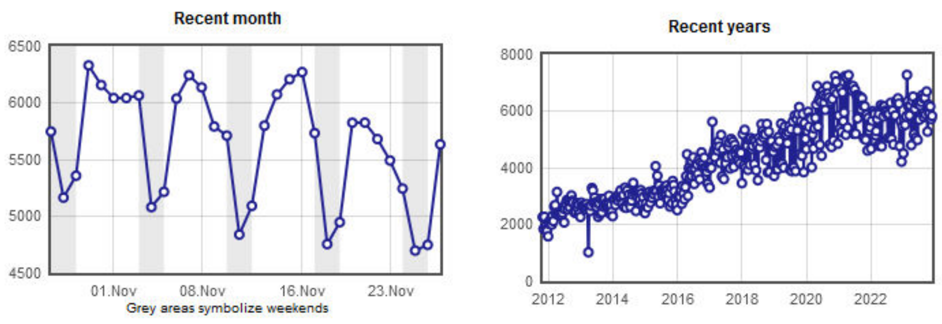


Figure 2.3: Trend of active mappers across recent months and years.
This graph represents ***active*** *mappers in the OSM community, and as can be noticed the number is on average steadily increasing across the years. Source: [27].*

Despite these limitations, OSM is trusted enough also due to its ever growing and very active community, with a registered user count of 10 million in november 2023, as opposed to six million in 2022 [27]. Contributors also include large private and public institutions of many countries [28], such as Geoportale E-R in the Emilia-Romagna region of Italy and DELFI for Germany. [29] finds that about 42% countries of the world are more than 95% complete, and generally the global road network is about 83% complete.

More than 900 applications use OpenStreetMap at the time of writing [30], including major applications such as Uber [31], Lyft [32] and Snapchat [33]. Examples also include famous videogames [34].

2.3 Microscopic traffic simulation: SUMO

The Digital Twin approach often implies some degree of simulation of the target problem or system; for the work of this thesis, in particular that described in Ch. 4 (ParallelTwin), we needed a traffic simulator to act as a Digital Twin for future research in the field of traffic pollution and urban planning. The simulator thus would need to match the common requirements of a Digital Twin as described in section 2.1.2.

First, various methods of traffic simulation exist:

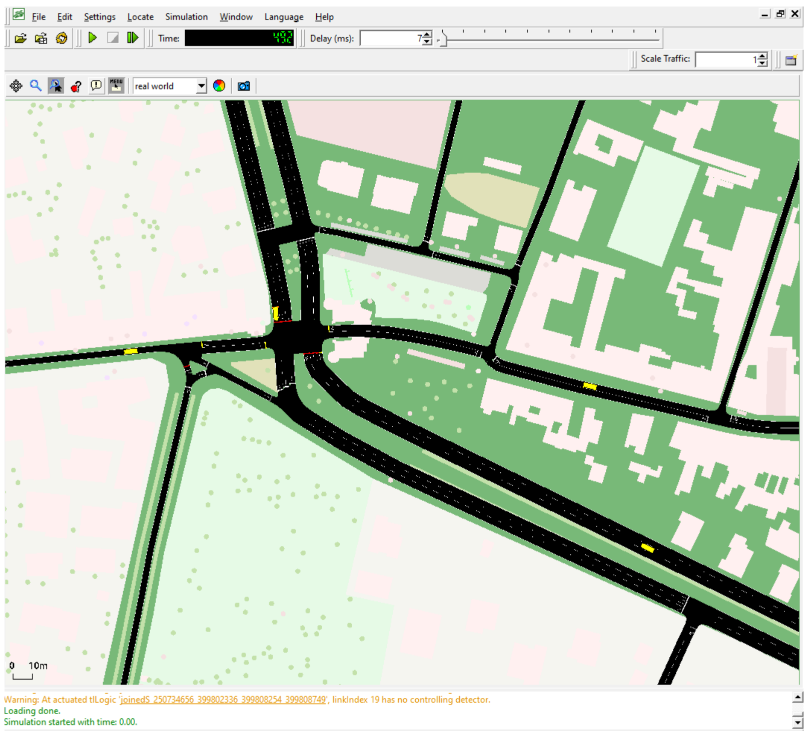
- **Macroscopic**: modeling the flow of traffic.
- **Microscopic**: simulating the external behavior of every vehicle participating in traffic, such as speed and acceleration.
- **Mesoscopic**: a combination of macroscopic and microscopic simulation.
- **Sub-microscopic**: simulating the internal behavior of every vehicle, for example gear shifts or payloads.

For the purpose of analyzing pollution, we chose to use a microscopic traffic simulator for more accurate results, even though a macroscopic model could also be used for broader calculations. Sub-microscopic simulation is not needed in this case, but the usecase detailed in Ch. 3 (Acc2Twin) regards the usage of a model akin to it, as it twins the content of a single transport vehicle.

To run a microscopic traffic simulation, we chose Eclipse SUMO, "**S**imulation of **U**rban **MO**bility" [35], an open source microscopic traffic simulation software package, mainly developed by the Institute of Transportation Systems at the German Aerospace Center.

SUMO logo

SUMO is structured around two main simulation programs, ***sumo*** and ***sumo-gui***, plus various supporting tools and scripts to process map and vehicle data and outputs. *sumo* runs the application in a command-line interface, while *sumo-gui* launches an OpenGL application that visualizes the running simulation in real time.



```
Warning: At actuated tlLogic 'joinedS_250734656_399802336_399808254_399808749', linkIndex 19 has no controlling detector.
Loading done.
Simulation started with time: 0.00.
```

Figure 2.4: Sample screen from sumo-gui

2.3.1 Creating a simulation

In this section we overview the process of building a SUMO simulation. This includes generation of topological information about the area of interest, and the creation of traffic models/demand files.

Geographical road network

The road network is represented by a *network file* (`.net.xml`) in SUMO, which is a custom XML format that contains nodes and edges of the road graph, representing lanes, roads, junctions, and other elements such as traffic lights and special road signs.

This format is not meant to be parsed or understood by humans: for this reasons, SUMO provides

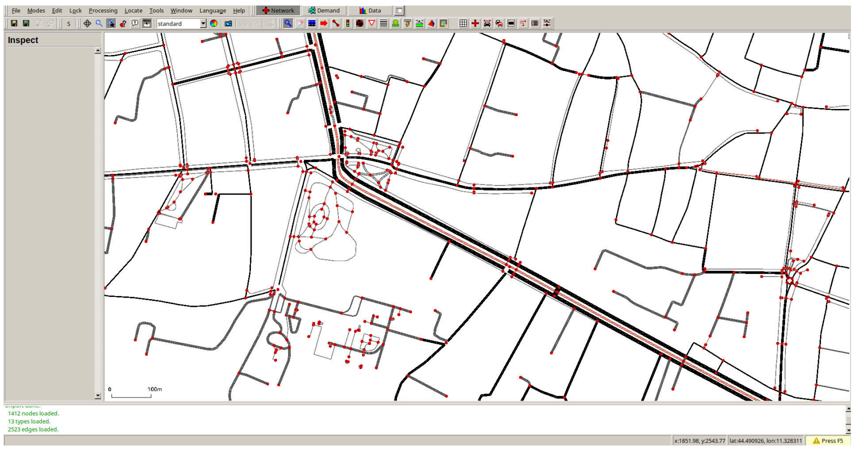


Figure 2.5: Sample screen from netedit, inside the network editing window Portrayed is a OSM-generated network in Bologna

several tools to handle network files and convert other formats to them. One of the main examples is a GUI application to design and edit road networks, **netedit**, mainly meant for either manual tweaking of existing network files or the generation of simple test networks.

The core tool to convert between different road network file formats is **netconvert**, which was used in this thesis for multiple purposes. Netconvert allows to parse other geographical data formats into SUMO networks, and to process SUMO network files in a variety of manners. Possible input formats include OpenStreetMap xml files, VISUM, OpenDrive, Shapefiles, and public transport stops, and the "SUMO plain" XML descriptor. The latter is a simpler XML format compared to SUMO network files, that can be feasibly created by a user to create a road network via a text file.

Netconvert also allows to process networks in a variety of ways: it can merge networks, to update old patches of data or expand the network file, apply geographical projections, and others. Most importantly for this thesis, netconvert can also be used to filter road elements. We mainly used the `--keep-edges.input-file` option for this purpose, which allows passing a plain text file with an edge identifier in each line, causing the output network to only contain the edges specified: this was used to apply the partitioning scheme on the road network in the work detailed in chapter 4.

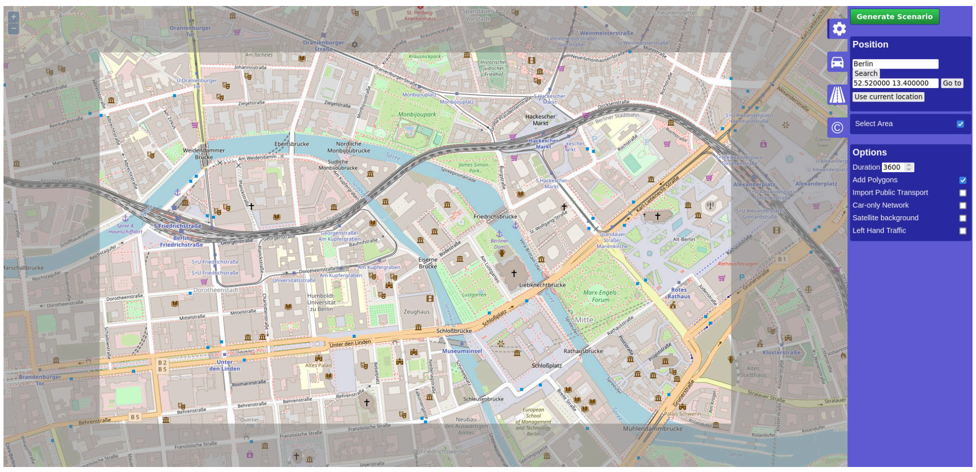


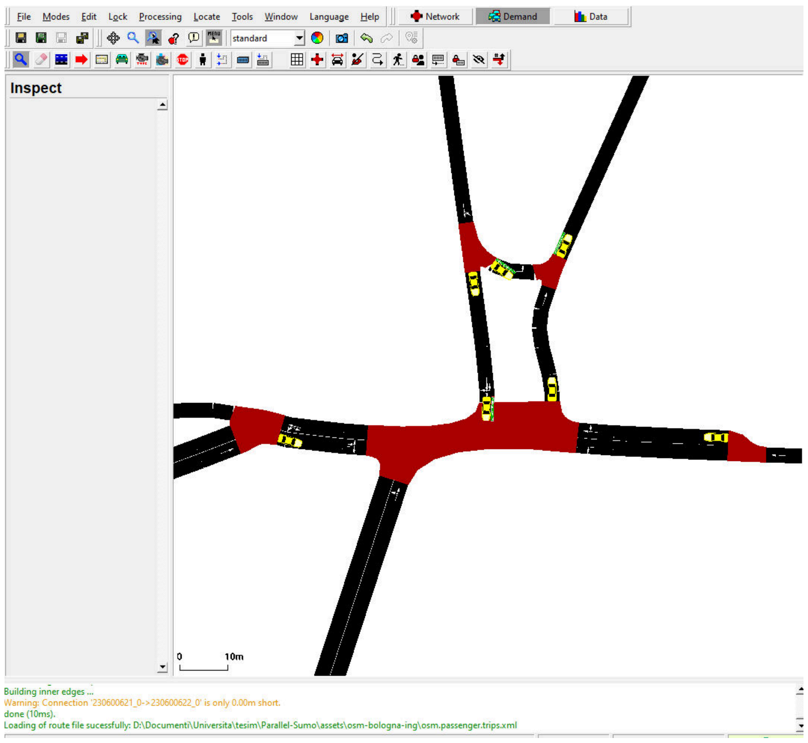
Figure 2.6: Interface of the OSM Web Wizard

To more readily import OSM road data, which is a useful source to simulate real road networks, SUMO offers a Web-UI to generate a simulation from a portion of the OSM database, **OSMWebWizard**. This allows to import OSM road network data, automating the download of this data and its conversion via netconvert, optionally generating random traffic on the created simulation.

An important point to note about this and similar processes is that netconvert transparently handles errors, inconsistencies and incompleteness in the input data, by using several heuristics to attempt to produce a realistic road network in those instances. While this does succeed in always providing a valid SUMO simulation, care must be taken in instances where a realistic simulation of a real road network is the goal, as the assumptions made by netconvert can result in inconsistencies from the real road network. As an example, opposing lanes in a road that connects two nodes are always considered to be connected (thus allowing U-turns), while this is not always the case.

Lastly, a tool of interest for the work of this thesis is **netgenerate**, which allows to generate artificial road networks and was used for testing. Netgenerate can create a variety of road shapes (such as web-shaped and grid-shaped), with an arbitrary size.

Traffic information



```
Building inner edges ...
Warning: Connection '230600621_0->230600622_0' is only 0.00m short.
done (10ms).
Loading of route file sucessfully: D:\Documenti\Universita\tesim\Parallel-Sumo\assets\osm-bologna-ing\osm.passenger.trips.xml
```

Figure 2.7: Sample screen from netedit, inside the demand (traffic) editing window

The traffic information of the simulation is encoded in so-called *demand files* (`.rou.xml`), which are another XML format (simpler when compared to road networks) that contains entries representing vehicles, routes, and related information. SUMO also supports pedestrian simulation, and multimodal simulation, meaning the simulation of people that use different vehicles (cars, public transport) at different legs of their journey.

```
<route id="r_0" edges="E0 E1 E2 E3 E4"/>
<vehicle id="v_0" depart="100.00" route="r_0"/>
<trip id="v_1" type="veh_passenger" depart="50.00"
  departLane="best" departSpeed="max" from="E3" to="E8"/>
```

Figure 2.8: Sample of a SUMO demand file
Contains an implicit vehicle plus route definition (trip), and a vehicle plus explicit route pair

Vehicles are defined by at least an id, an associated route, which defines a vehicle's starting point, a departure time, and usually a vehicle type. The departure time is in the units of the simulation time, which are seconds in SUMO. Vehicle types include information on a vehicle's size, weight, priority, access to public transport lanes, and other metadata. Instead of a specific vehicle, a demand file can contain a *flow*, representing a flux of vehicles departing in a window of time, thus including a start and end time instead of a singular departure time.

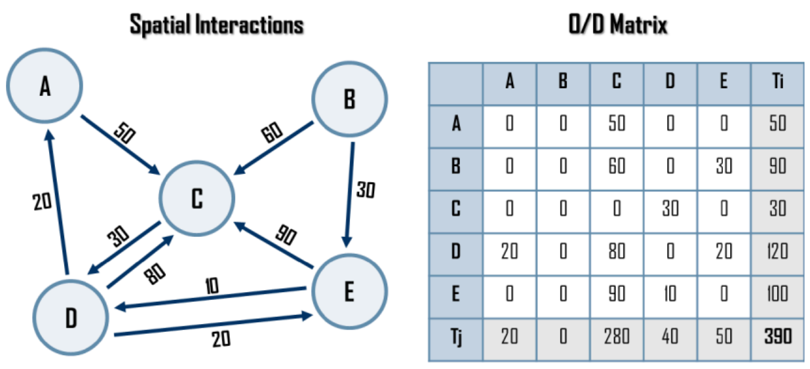
Routes can be defined implicitly via a start and end lane ID, optionally including some inbetween passing points, or explictly by a list of lane IDs the route owners will have to pass through. Implicit routes are called *trips* and are computed at runtime by SUMO, using a series of cost functions in a lowest-cost algorithm. Trips can be converted into explicit routes using the **duarouter** program included wth SUMO, which will evaluate the path required, and transparently remove impossible paths whose start and end have no connection.

Like with network files, SUMO offers many tools to convert and generate traffic information data. *Netedit* also contains a traffic demand modeling window (see figure 2.7), but as with networks this is mostly useful for tweaking existing demand files or creating simple test ones.

The format is simple enough that generating by hand (or through custom-created scripts) is feasible, unlike with road networks. SUMO also offers a **randomTrips** tool that generates random routes, in a random distribution in time, which is a useful way of quickly creating traffic information for a simulation, and was used for testing during the work of this thesis. Other generation methods include the use of induction loops data, and similar heuristics.

An important input format for SUMO traffic model conversion are **O**rigin/**D**estination matrices, which are a data structure that highlights the vehicle flow between two different areas of the network. Based on this information, SUMO can generate a variable number of vehicular routes to reproduce the same amount of vehicular routes.

OD matrices are commonly estimated from traffic counts at specific counting points of the real road network, and are the main datum of reference in a variety of traffic management problems [36] [37]. In general, they represent a flow between nodes of a graph, also used in contexts like demographics [38]. An OD matrix is related to specific temporal and spatial dimensions, as it considers a specific time period (which SUMO defines in seconds), and a certain area of interest. SUMO calls these areas **T**raffic **A**ssignment **Z**ones, identifying them with assigned IDs, and interpreting OD matrices as traffic between Traffic Assignment Zones when generating the traffic flow.



	A	B	C	D	E	Ti
A	0	0	50	0	0	50
B	0	0	60	0	30	90
C	0	0	0	30	0	30
D	20	0	80	0	20	120
E	0	0	90	10	0	100
Tj	20	0	280	40	50	**390**

Figure 2.9: A conceptual diagram of an origin-destination matrix. Source: Dr. Jean-Paul Rodrigue, Dept. of Global Studies & Geography , Hofstra University, New York, USA.

```
<tazs>
  <taz id="<TAZ_ID>" edges="<EDGE_ID> <EDGE_ID> ..."/>
</tazs>
```

Figure 2.10: Example Traffic Assignment Zone file

For the purpose of modelling traffic as OD matrices, data is publicly available for many major cities and locations: examples include the cities of Bologna [39] and Paris [40]. This kind of data generally needs to be converted in a format SUMO understands, first by converting geographic information expressed in latitude and longitude in the Cartesian system of reference of the network, for which SUMO offers the *polyconvert* tool, to generate enclosing polygons for each TAZ. After this, the *edgesInDistrict.py* script locates the constituent edges of each TAZ, which is a required steps as routes need to start and end at specific edges. OD matrices can then be built using these defined TAZs alongside the available traffic data. For simpler usecases, TAZs can also be defined by drawing a polygon in Netedit and then executing *edgesInDistrict.py* in a similar manner.

To convert OD matrices into a proper SUMO traffic demands file, the software package offers the **od2trips** tool to generate individual vehicle trips. Od2trips allows to specify timeline information, a scale, and an uniform spread over a time period if needed. TAZs can be further used in other ways in SUMO, such as interaction with TraCI (section 2.3.4).

The possibility of importing real-world data allows SUMO to better act as a Digital Twin of a specific traffic area, since as mentioned earlier (2.1.2) being able to link to the data of the real-world counterpart or physical twin (in this case, the real city) is a necessity for DTs.

A final tool of interest to this thesis that SUMO provides for traffic demand modelling is **cutRoutes.py**, which is used to partition a traffic demand file made for a larger network to match a smaller subset of that network. Given a network n reduced to a smaller network n', and with r being the traffic file for n, cutRoutes takes n' and r as inputs to produce a traffic file r' that only encompasses the roads of n'.

This script requires all routes to be expressed as explicit *route* elements instead of *trips*, as it needs

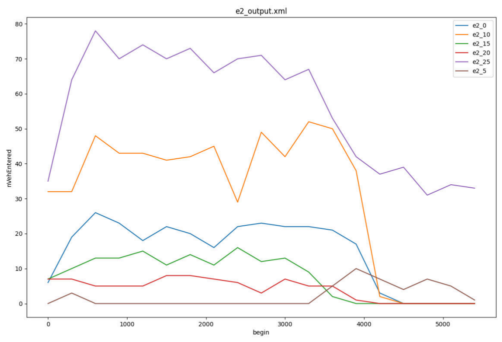


Figure 2.11: Example output of SUMO plotXMLAttributes.py from simulated Lane Area Detectors (vehicle cameras)
Source: SUMO documentation

the knowledge of which edges of a route pass over the partitioned road network. This can be amended by using *duarouter* before the execution of the script, to convert trips into routes. The script adjusts depart times for vehicles as needed, in case a vehicle's starting edges are removed from the network, to simulate the vehicle arriving later in the area of interest. This tool also considers some edge cases, in particular cases in which a route goes out of the area of interest and then returns inside it, in which case the route is split into two parts rather than simply cropped to the network's edges.

This script was initially used to handle the partitioned traffic demand files in our parallel work in chapter 4, but was later replaced by a custom version for reasons that will be explained in that chapter.

Final configuration

To run the sumo simulation, a network file and one or more traffic demand files are needed, as specified by the `-n` and `-r` arguments in *sumo* or *sumo-gui*. To simplify the passing of parameters for each different simulation configuration, the SUMO programs allow to pass a single `sumocfg` file with the `-c` parameter. The `sumocfg` file is an XML file which contains all parameters that should be passed to the program, including network files, demand files, special configurations, output specifications, etc.

The configuration schema for a passed set of parameters can be saved to file with the `--save-schema` option, to simplify rerunning the same simulation by generating a portable image of it. Thus, a simulation that combines various files can be represented by a single `sumocfg` file (which references other XML input files). Other SUMO executables such as netedit, netconvert and duarouter support this mechanism too, allowing for portable complex configurations.

2.3.2 Simulation output and emissions

SUMO supports a variety of output formats [41]. It can output a raw dump of the state of each lane at each simulation time-step via `--netstate-dump`, containing the ID and state of every contained vehicle therein.

As this level of detail usually isn't necessary, one can output data from manually placed sensors inside the simulation, related to specific lanes or junctions, or constrained to specific kinds of data. For example, detectors can be defined in XML files passed as ”additional files” to the simulation, such as Lanearea detectors that simulate vehicle tracking cameras. SUMO also offers tools to visualize many of its outputs [42](see figure 2.11)

Of interest for the work of this thesis the vehicle emission output, which uses the vehicle emission models and classes specified inside vehicle types. It can be enabled with `--emission-output`. SUMO allows to use various emission models of vehicles, that can output the vehicle's emissions of various kinds of pollutants, as resumed by the following table:

model	CO2	CO	HC	NOx	PMx	fuel consumption	electricity consumption
HBEFA v2.1-base	x	x	x	x	x	x	-
HBEFA v3.1-base	x	x	x	x	x	x	-
HBEFA v4.2-base	x	x	x	x	x	x	x
PHEMlight	x	x	x	x	x	x	(x)
PHEMlight	x	x	x	x	x	x	(x)
Electric Vehicle Model	-	-	-	-	-	-	x
No Emissions	-	-	-	-	-	-	-

Table 2.1: Pollutant simulation by emission class, from SUMO website, sumo.dlr.de/docs/Models/Emissions.html

Emissions can be obtained both as unaggregated output for all lanes (`--emission-output`), the emissions for selected vehicles or during a trip (`--tripinfo-output` with either `--device.emissions.probability` or `--device.emissions.explicit` set), and over an edge/lane (defined by additional XML files). Vehicles and lanes can also be colored depending on emissions in the GUI, and visualization tools are available in SUMO to help in showcasing higher emission areas for specified pollutants (see figure 2.12).

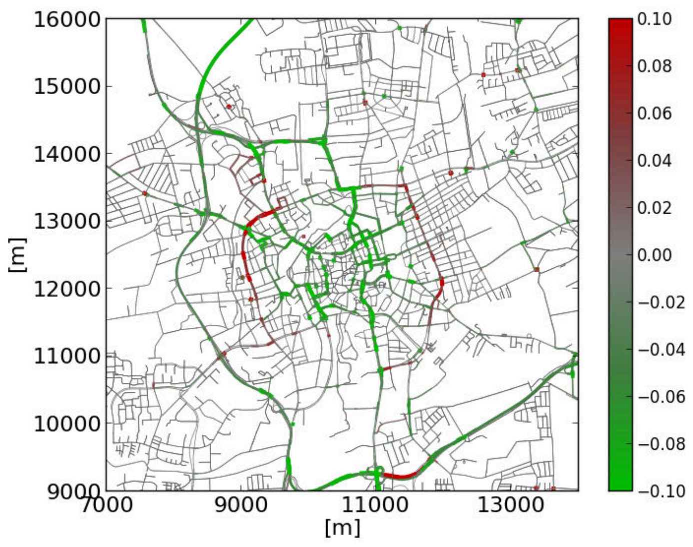


Figure 2.12: Example output of SUMO plot_net_dump.py with edges colored depending on simulated emissions
Shown: city of Brunswick, changes in NOx emissions. Source: SUMO documentation at https://sumo.dlr.de/docs/Tools/Visualization.html#plot_net_dumppy

2.3.3 Vehicle behavior

Each vehicle in SUMO is modelled by a vehicle following model and a lane changing model. All of these are specified in the vehicle type. The default model is *Krauss*, but many other options are available.

Details on model functionalities and parameters go beyond the scope of this thesis. The main point of interest for our work is their general behavior: each vehicle in most models attempts to maintain a minimum gap with the next vehicle, depending on various parameters which include deceleration rates and reaction times. Thus, the behavior of every vehicle is affected by the other vehicles in the simulation.

Vehicles make use of stoplights and traffic signs, which are marked in the network file, and can have more specific road behavior as defined in the network file, such as emergency services lanes.

2.3.4 Interaction: Libsumo and TraCI

As previously stated in section 2.1.2, a key part of a Digital Twin is the possibility of interacting with it from the outside: a Digital Twin should expose an API to allow humans and programs to read and possibly modify its state.

SUMO fulfills this requirement, as it offers two main means of interacting with the simulation at runtime through external programs, Libsumo and TraCI. Both offer functions to obtain data on and affect the current simulation, sharing the same API, but differing in the method of interaction.

TraCI is a TCP-based API, short for "**Tra**ffic **C**ontrol **I**nterface". A simulation can host a server to respond to API requests using the `--remote-port` argument. Libraries for various programming languages are supplied to create client programs that interact with a simulation, including C, Java and Python.

Libsumo is an alternative to TraCI to reduce the overhead caused by using TCP sockets to communicate with the simulation: it's a library of static functions with the same interface as TraCI, that can be used by a program to both include the simulation process and interact with it. A program created in this way will contain the SUMO simulation in the same process as the interaction code. Bindings are supplied by SUMO for various languages, again including C, Java and Python.

Compared to TraCI, Libsumo is limited in some purposes, most importantly GUI usage in Windows. Of course, as it contains the simulation in its same process, it allows only one "client" (that being the same process that runs the simulation), while TraCI allows multiple clients to connect to the same simulation. Still, they can be combined, as a simulation started from Libsumo can use the `--remote-port` argument to also host a TraCI server and allow TraCI clients to connect to it.

The interface for both ways of interacting includes functions to retrieve overall data (for example simulated vehicle ids and current time step) and specific vehicle data (for example speed, position, and emissions). It also includes functions to modify the simulation: creating vehicles, moving them, and other means of interaction. Notably, it is not possible to alter the road network at runtime, preventing the usage of dynamic partitioning with SUMO as-is.

A full documentation of the API shared by TraCI and Libsumo can be found at [43]. We will overview only the main functions of interest to this thesis.

Simulation Control

- `simulationStep(double)`: forces the simulation to run a single time step, or N time steps until the simulation time (in seconds) equals the value passed if different than 0. Notably, SUMO does not proceed as normal if TraCI is enabled or Libsumo is in use, but waits for this function to be called before proceeding with any simulation.

Vehicle Value Retrieval

- `getIDList()`: returns the ID of all vehicles currently inside the simulation.
- `getLastStepVehicleIDs(edgeID)`: get the ID of all vehicles in the specified edge (lane)
- A number of vehicle variables can be retrieved, including position, speed, lane and route ID, index of the current edge among the vehicle's route, etc.

State Changing

- `slowDown(vehID, double, double)`: changes the vehicle speed to the given value in the given time (despite the name, allows to increase speed too).
- `moveTo(vehID, laneId, posInLane)`: moves the vehicle to the given lane, which must be a part of the given route.
- `add(vehID, routeID, vehicleTypeID, departLane, departPos, departSpeed, arrivalLane, arrivalPos, arrivalSpeed, tazFrom, tazTo, publicTransportLine, personCapacity, personAmount)`: adds a vehicle to the simulation. Only vehID and routeID are required. The route referenced can consist of two disconnected edges, and will be calculated as a trip.

An important point to note regarding the state changing functions is that little verification of parameter correctness is done on the SUMO side, which may cause fatal errors in case wrong parameters (duplicate vehicles, non-existing route IDs, wrong lanes) are inserted, especially worrisome when working with LibSUMO as the same process would contain both the simulation and the interacting ”client”.

Similarly, care should be taken when adding vehicles or modifying the position to avoid placing them on top or on route of another vehicle; this is not a fatal error, but will cause a simulated collision that leads to loss of vehicles and as such simulated data.

TraCI also allows to subscribe to an object variable or an object context, allowing to retrieve data periodically. An object can be any simulated sensor, vehicle, route, lane, etc.

Chapter 3

Acc2Twin

The first Digital Twin case that was part of the work for this thesis was in collaboration with Robopac, a packaging manufacturing company that is part of Aetnagroup. In particular the work was a part of their Acc2Twin project, derived from their participation in the Change2Twin EU project.

Our part of the work was focused on the extrapolation and interpolation of continuous route data, to be used in the simulation of a singular transport vehicle to evaluate the forces undergone by its load.

3.1 A Digital Twin for package manufacturing

Change2Twin is an European project to support manufacturing companies in digitalization through Digital Twin solutions. Robopac's participation in the project, and its own effort in its internal Acc2Twin project, is focused on the application of the Digital Twin approach to improve the stability of palletized load in the transport of goods, more specifically affecting the decision process in the choice of the packaging and wrapping film applied to the transported material.

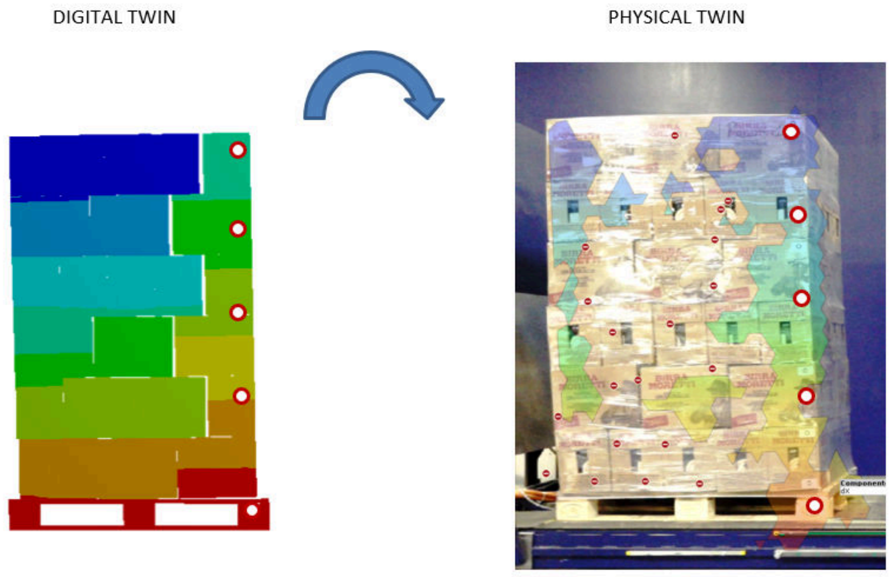


Figure 3.1: Representation of the pallet Digital Twin developed by Robopac/TechLab
Source: Robopac/ [44]

The packaging required to safely transport a pallet load to its destination relies on factors like product weight, shape, packaging characteristics, pallet arrangement, stretch film type, and the stresses encountered during road, train, ship, or air transport. This amount is normally approximated empirically by operators through common sense, without a scientific approach. This approach is not optimal regarding the amount of plastic film used: normally the customer doesn't want to risk damage to the product, using more plastic film than necessary.

The research team of Robopac (TechLab) developed virtual model of a transported pallet, and then using it as the core engine of a Digital Twin. This DT is meant to be used by the clients of Robopac during the wrapping process of plastic film, simulating the pallet in realistic load conditions and using this information to optimize the wrapping process. This allows to reduce plastic usage in transport packaging [45] [45].

This Digital Twin of course requires data related to the physical product and the forces it will endure to accurately represent the real situation; obtaining the latter requires knowledge on the transport route, and an acceleration profile of the vehicle. This process is organized in a pipeline that can be described as follows and that is represented in figure 3.2.

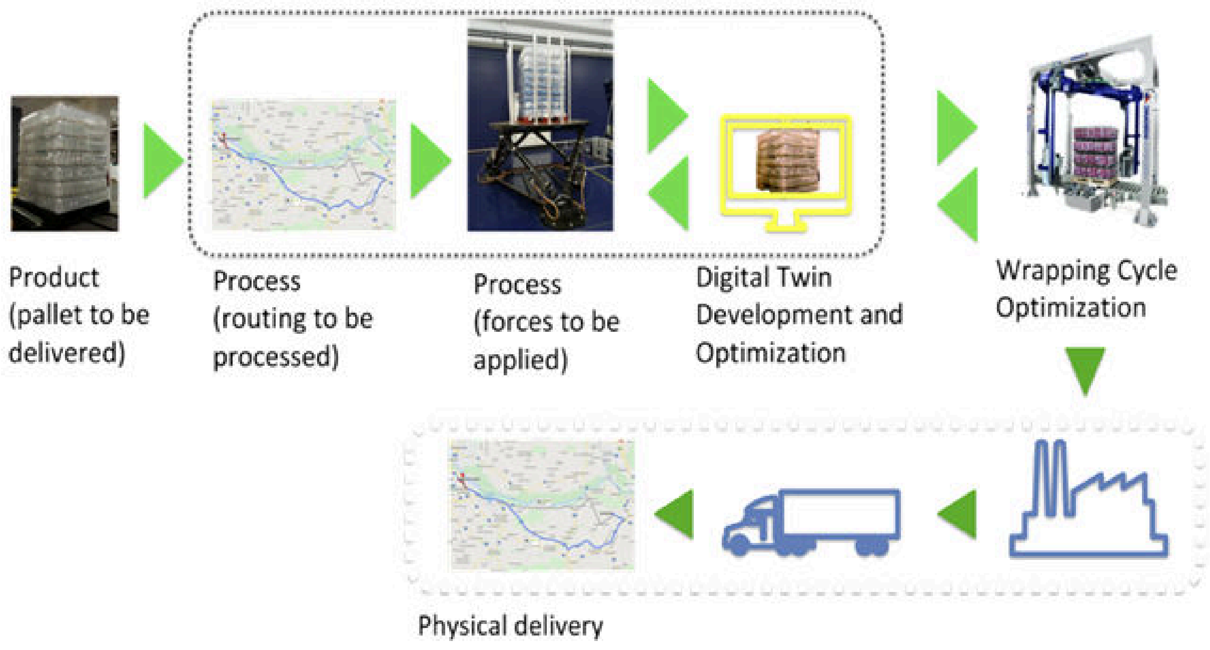

Product (pallet to be delivered)
Process (routing to be processed)
Process (forces to be applied)
Digital Twin Development and Optimization
Wrapping Cycle Optimization
Physical delivery

Figure 3.2: Representation of the pipeline for the Robopac DT

First, data about the product is modelled, describing its physical properties beyond a simple 3d model, to allow for an accurate physical simulation. After that, data about the route must be obtained: first taking geographical data about the road network including relevant metadata such as maximum speeds, then computing the route between the origin and the destination and lastly producing geographical data on the route as a continuous (or anyways fine-grained) curve.

The processing continues with the application of an acceleration model to the route data, to obtain realistic accelerations over time that the transported pallet will endure. The acceleration data is then applied to the virtual palletized product, which is used to optimized the wrapping cycle.

Our work in this thesis applies to the second step, obtaining and processing route data into a continuous line tagged with relevant metadata. We used the OpenStreetMap dataset, computing a route and interpolating and processing its geometrical data to obtain a continuous line (sampled at an arbitrary precision).

For this purpose, our work can be separated in three main steps:

1. Road data extraction from OpenStreetMap and integration between different sources (such as OpenTopoData and Google Elevation API).
2. Route computing, using the third-party service OSRM which interacts with OSM to allow for a robust, well-tested, and most importantly configurable navigation logic.
3. Interpolation of the route geometry to produce a continuous line to allow sampling points at an arbitrary precision.
4. Processing of route metadata to produce a final output data table of geometrical points of arbitrary precision, labelled as necessary with metadata related to road shape and speed information.

We developed these processing steps as a Python package, `route2vel` [46], integrated with the external services mentioned. We provide a simple web application to generate route data between two locations, available through the scripts **launch-webui.ps1** and **launch-webui.sh**.

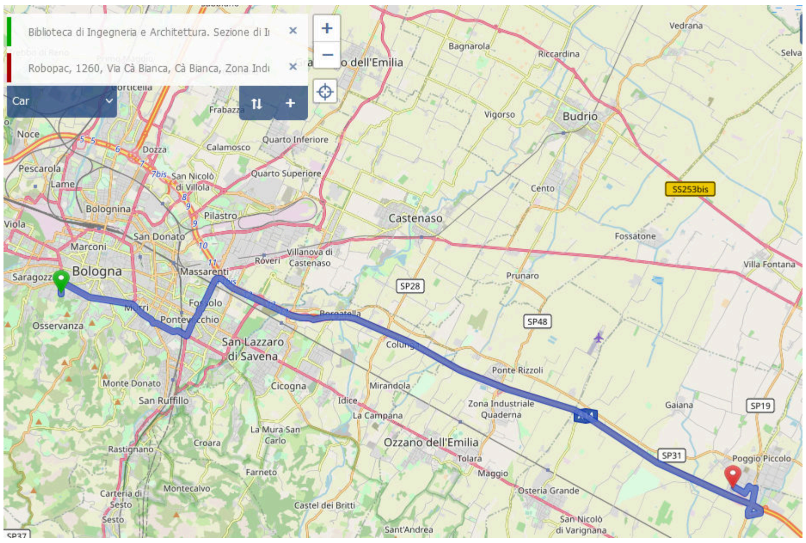
3.2 Package structure

The main files of the Python package are:

- `classes.py`: contains the `RichDirection` and `InterpolatingDirection` classes, which comprise the main access points of the package.
- `postprocess.py`: includes code to add metadata to the output, and generate the output *csv* file.
- `loading.py`: loads map data from OSM and elevation APIs.
- `route.py`: interacts with OSRM to get a route between locations and map it to our data.
- `interp.py`: contains functions to interpolate a set of geometrical data into a curve of points of arbitrary precision.

We mainly used the `osmnx` package to interact with OSM, the `geopandas` package to handle and process the route data, and the frequently-used `scipy` package to interpolate line segments.

3.3 Navigation: OSRM


Biblioteca di Ingegneria e Architettura. Sezione di I

Robopac, 1260, Via Cà Bianca, Cà Bianca, Zona Indi

Car

Figure 3.3: OSRM demo website

OSRM (standing for *Open Source Routing Machine*) is a routing engine based on the OpenStreetMap data format, able to handle large-scale networks at a high performance, and supporting various vehicle modes, such as car, walk and even allowing for custom vehicle profiles. OSRM allows to load map data in a flexible way, only needing to provide the *.osm.pbf* map files for the regions the user requires.

OSRM logo

We decided to use an existing third-party routing service to have a widely-used, optimized and tested path finding service, especially as numerous third party services offer the possibility to specify truck profiles or custom navigation profiles which would have otherwise required a custom implementation, due to the transport navigation usecase. Additionally, using a third-party service allows to have it as the first step of the processing pipeline, before loading map data, allowing to only load the necessary data for the route without the surrounding area that would be used to calculate the route between the specified locations.

OSRM was chosen instead of other OSM-based alternatives (like *Valhalla* and *GraphHopper*[1]) for its support of directly including OSM road IDs in the metadata of the output route, allowing for easier further processing in our usecase.

The services are available as a Docker machine (`osrm-backend`), which includes both the containers used in the offline map data preprocessing and the main container that provides the online routing service. Its API is detailed in the referenced documentation; for our purposes, we used the **route** service, which finds the fastest route between N coordinates on a given vehicle profile. The service includes an `annotations` option to include OSM road IDs, which was useful for our usecase as previously mentioned.

Vehicle profiles (such as car, bycicle, foot) are defined through Lua files, which define size, weight, default speed, and other parameters used in navigation (for an example see Figure 3.4). For the transport usecase, this is a necessary feature as trucks need to avoid roads that do not conform to their size, and tend to prefer driving outside of urban areas.

[1]Available with OSRM in the standard www.openstreetmap.org navigation

```lua
function setup()
  return {
    [...]
    default_mode              = mode.driving,
    default_speed             = 10,
    oneway_handling           = true,
    side_road_multiplier      = 0.8,
    turn_penalty              = 7.5,
    speed_reduction           = 0.8,
    turn_bias                 = 1.075,
    cardinal_directions       = false,

    -- meters
    vehicle_height = 2.0,
    vehicle_width = 1.9,
    vehicle_length = 4.8,
    -- kg
    vehicle_weight = 2000,
    [...]
```

Figure 3.4: Extract of a vehicle profile for OSRM (in Lua)

In our case, we used OSRM through the `routingpy` Python module, which abstracts API calls for route finding over a variety of services, including OSRM and its alternatives Valhalla and Graphhopper. OSRM is hosted in a local Docker container, and we provide scripts to automatically setup and launch the required preprocessing and service containers, **run_osrm.bat** and **run_osrm.ps1**. The scripts also download the North-East Italy map from *Geofabrik* as a sample, and check for updates.

After running the OSRM navigation, we process the result to obtain a table (in the form of a `geopandas` GeoDataFrame) representing the geometry and metadata of traversed roads.

3.4 Discrete road geometry interpolation

The purpose of this part of the work was creating road geometry data to be used in evaluating forces, which requires fine-grained geometrical data. Commercial map datasets usually have a minimum precision of several meters in curve segments, and up to several hundred meters in more linear roads, which is good enough for a normal navigation usecase, but not for force calculation. High precision datasets such as *OpenEarthMap* [47] exist, but they are not tailored to a road navigation usecase and do not cover all possible countries of interest for the company, which include third-world countries that are often neglected in such datasets.

The dataset of choice in our case, OpenStreetMap, is widely used and constantly updated by a community, providing useful metadata such as road type and speed limits, but falls in the first case of having a low precision on average (see figures 3.5 and 3.6). In addition, OpenStreetMap lacks elevation data, necessary for our usecase, which needs to be integrated via third-party services. To acccount for the relatively low precision in the road geometry, we used different kinds of interpolation depending on the road segment to generate an artificial continuous line allowing sampling geometry at an arbitrary distance.

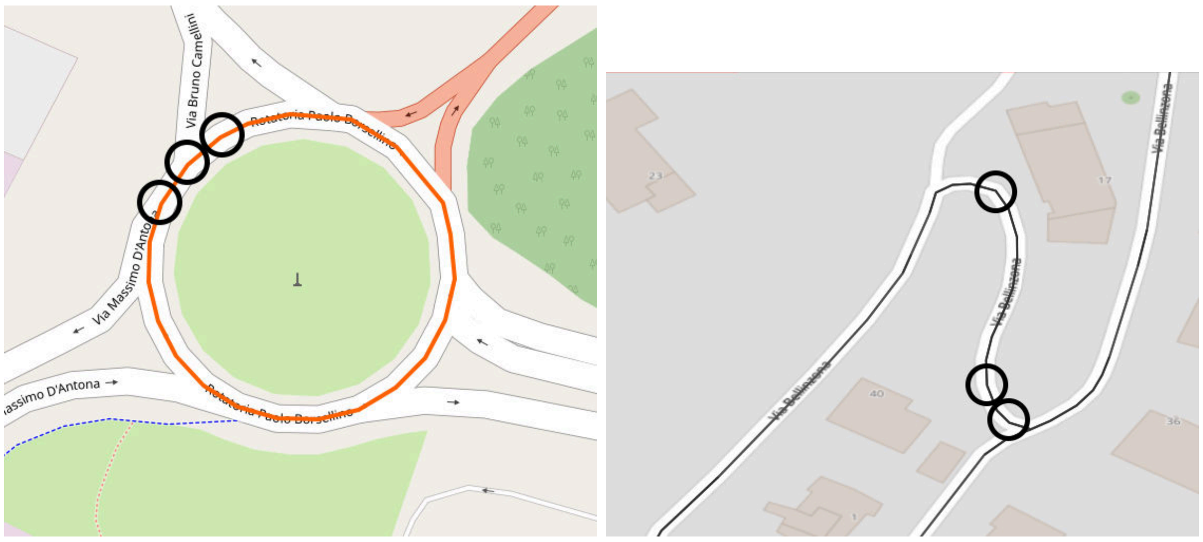


Figure 3.5: Examples of coarse grained curves in OSM maps
The "spikes" caused by linear interpolation are highlighted

3.4.1 Elevation data

Several services exist to obtain the elevation of a point or an area defined by coordinates, such as **OpenTopoData** and **Google Elevation API**. We elected for the former due to its accessibility and the possibility of running the service offline as a Docker container, but allow easily switching to the latter as a parameter in case a faster service is desired.

OpenTopoData is a REST API server that offers both a free public API with a strict rate limit, and a Docker image to self-host the service with downloaded map data, supporting most file formats and tiling schemes used by popular open elevation datasets. Its API has the same interface as Google Maps elevation API, allowing to retrieve the elevation of one or several points, or samples interpolated in an area. We use the public API by default for simplicity, while switching to a local machine requires simply changing the target URL.

The dataset of choice is the *aster30m*, developed by NASA, which is the highest-precision global-spanning dataset available in the public API of OpenTopoData.

An important point to note is the format and precision of elevation datasets: they are in a rasterized format, meaning in a matrix of evelation values (often integers with a precision of 1m), and at precisions of tens of meters. So, there is the same issue with precision mentioned for map geometry, as our usecase requires finer-grained samples; thus, we interpolated the elevation data alongside the 2D geometry map data, as described in the following chapter.

3.4.2 Segmented interpolation

The standard way to generate additional samples between coarse-grained data points is to use one of several kinds of interpolation, such as linear interpolation, Bézier curves and splines. We initially attempted to use a single type of interpolation, but found issues in different road shapes depending on the kind.

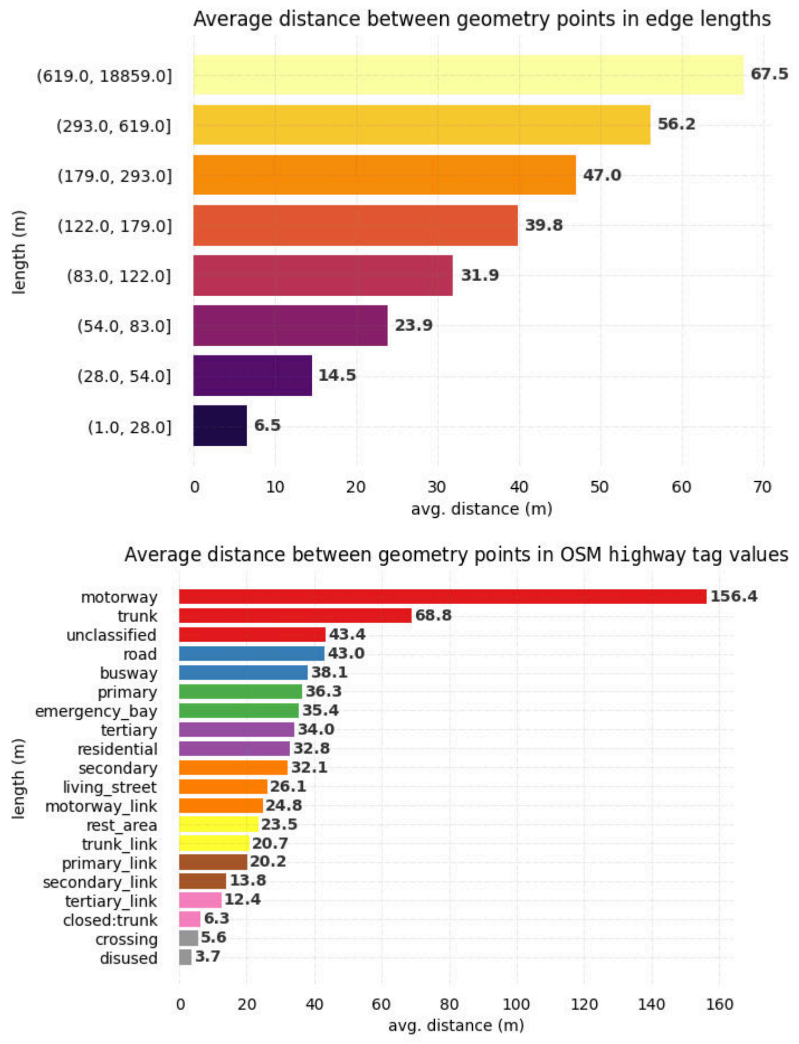


Figure 3.6: Average point distance in OSM data for the Emilia-Romagna region
Data is grouped by length of the full way (top, in groups of equal counts) and by `highway` *tag/road type (bottom). As noticeable, the average distance between points is far higher than the ideal of one meter in longer segments, and in most common road types.*
It can be noticed that road types typically associated to curves (such as links) have a lower point distance on average.

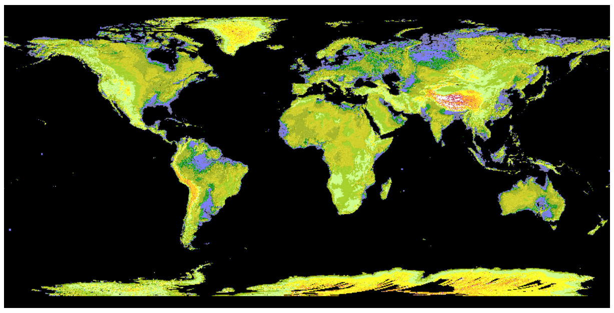


Figure 3.7: Rendering of aster30m *dataset.*
Source: asterweb.jpl.nasa.gov/gdem.asp

Linear interpolations works practically well enough for more linear roads, but generates excessively sharp or ”jagged” turns in curves and roundabouts, even if they normally have a higher precision in the source data. This again is not an issue in normal user-oriented usecase, but could lead to wrong acceleration calculations in ours. Conversely, using a curve interpolation like piecewise Bézier curves or splines approximates the real road shape very well, but leads to ”wavy” lines in a straight road with a high distance between points (see figure 3.8).

To be more precise, Bézier curves lead to issues when used with roads with a high distance between sample points in general, as they will tend to ”overshoot” the lines between points to approximate a curve that connects the points - conversely, straight lines were found to not have many issues when interpolated with Bézier curves in case they had an unusually short sampling distance compared to usual lines in OSM data.

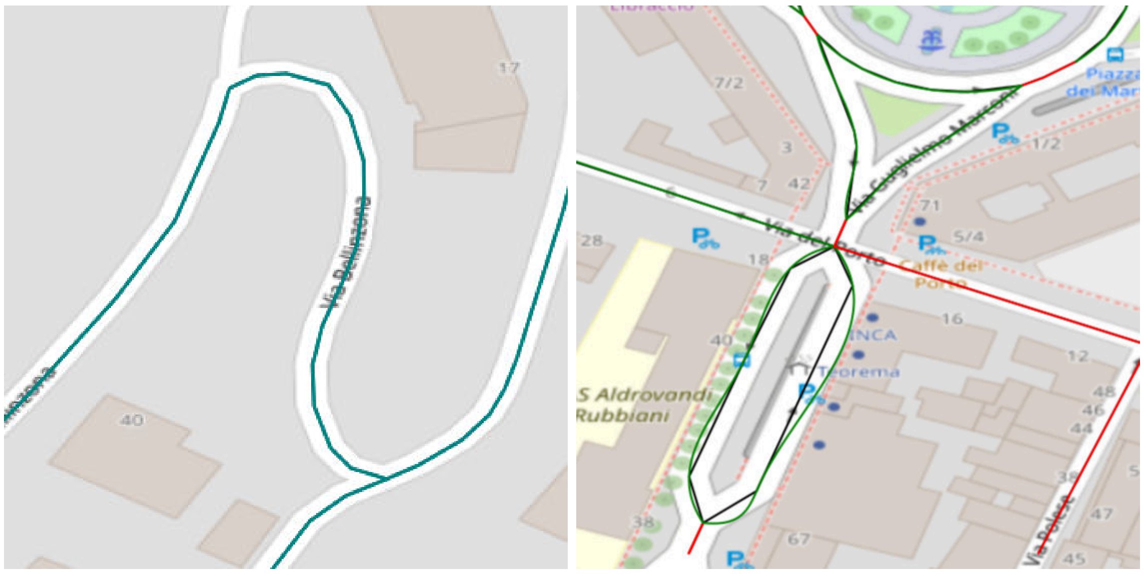


Figure 3.8: Interpolation errors: jagged curves with linear interpolation (left), overshooting geometry in straighter roads with bezier interpolation (right).

Because of this, we decided to use both approaches depending on the type of road. The general process is using a piecewise Bézier (or spline) interpolation for curves and roundabouts, and a linear interpolation for straight road segments.

To distinguish between different road types, we use both OSM metadata, which contains information

```
# input:
#   coords (list of points in the road)
#   way_data (dictionary containing OSM way tags)
#   distance_treshold (distance to separate curves
#     from lines)
current_group = [coords[0]]
group_is_curve = None
for prev_point, current_point in zip(coords[:-1], coords[1:]):
    is_curve = None

    if metadata is not None and is_curve is None:
        is_curve = is_curve_from_metadata(way_data)

    if is_curve is None:
        distance = dist(prev_point, current_point)
        is_curve = distance <= distance_treshold

    if group_is_curve is None:
        group_is_curve = is_curve

    if is_curve == group_is_curve:
        current_group.append(current_point)
    else:
        result.append(current_group)
        labels.append(group_is_curve)

        group_is_curve = is_curve
        current_group = [prev_point, current_point]

result.append(current_group)
labels.append(group_is_curve)
```

Figure 3.9: Line-curve tagging simplified algorithm, in Python (simplified for readability)

on some kinds of roads like roundabouts, and sample distance: curve segments consistently have a higher point density to be able to represent a curved shape through a series of lines between points, so we use a fixed treshold (15 meters) of point distance to tag each point of the road geometry as belonging to a curve or straight segment (see figure 3.9). We then clean up the result to remove excessively short segments of either type which can be attributed to false positives of either kind.

The interpolation takes one or more parameters t from 0 to 1 as a `numpy` array, and returns the same number of points sampled at that fraction of the sequence of road geometries comprising the route. It first evaluates which segment of the route each parameter belongs to, then normalizes t to a t_{adj} value within that segment's bounds, and evaluates the specific interpolation of that segment (see figure 3.10, and figures 3.11 and 3.12 for results).

Note that false positives on curve tagging due to unusually high point density in a straight road segment are possible, but usually not an issue from testing: as mentioned before, straight roads with a short enough distance between points doesn't lead to issues with bezier curves, which would be the case in case of false positives for curve tagging.

To run the curve-tagging and line-curve interpolation, we use efficient `pandas` function to avoid the overhead of iterating operations on large amount of data in Python, while exploiting its powerful data

```python
# input:
#   t_values (numpy array of t parameters)
#   linestrings (series of consecutive road segments as
#     shapely Linestrings)
#   curve_labels (boolean list, being True for each segment
#     if that segment is a curve)
#   full_distance (total length of series of road segments)
#   out (list to store resulting points in)

cur_dist = 0
cur_idx  = 0
t_adj = t_values * full_distance

for (i, line), is_curve in zip(enumerate(linestrings), curve_labels):
    # Make the curve continuous with surrounding lines
    prev_line = linestrings[i-1] if i-1 >= 0 else None
    next_line = linestrings[i+1] if (i+1 < len(linestrings)) else None
    bezier_prev = close_point_towards(line.coords[0],
      np.array(prev_line.coords[-2])) if prev_line else None
    bezier_next = close_point_towards(line.coords[-1],
      np.array(next_line.coords[1])) if next_line else None

    while t_adj[cur_idx] - cur_dist <= line.length + precision:
        line_t = (t_adj[cur_idx] - cur_dist) / line.length
        if is_curve:
            point = bezier_interpolate(
              line.coords, line_t,
              bezier_prev, bezier_next
            )
        else:
            point = linear_interpolation(line.coords, line_t)
        out[cur_idx] = point
        cur_idx += 1
        if cur_idx >= len(t_adj):
            break
    if cur_idx >= len(t_adj):
        break
    cur_dist += line.length
```

Figure 3.10: Multi-line interpolation algorithm, in Python (simplified for readability)

processing interfaces. We also use the `shapely` package to represent geographical data.

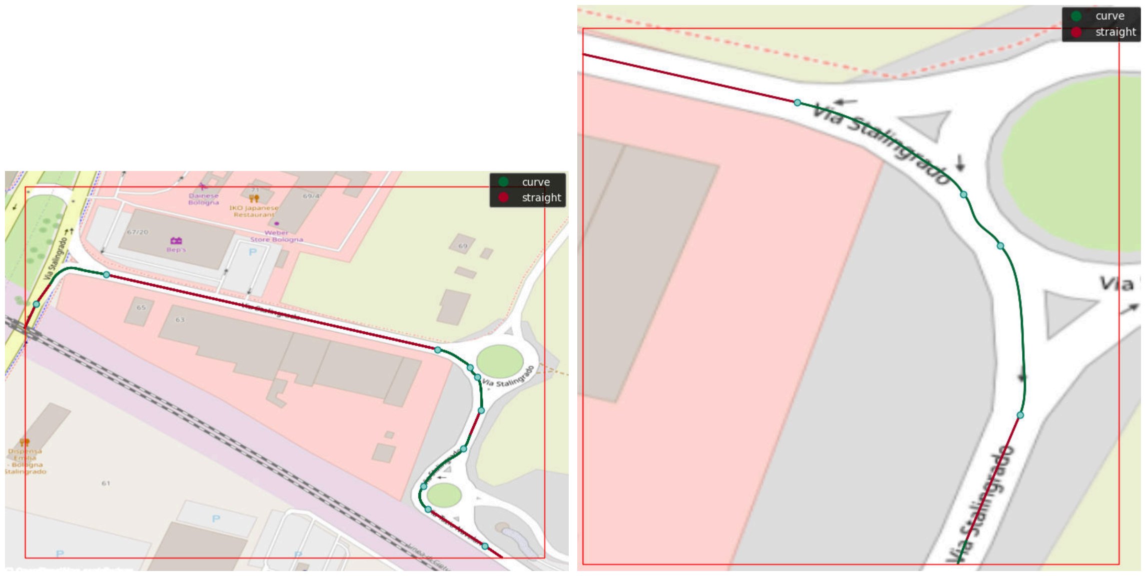


Figure 3.11: Interpolation examples, with line segments colored depending on curve tagging.

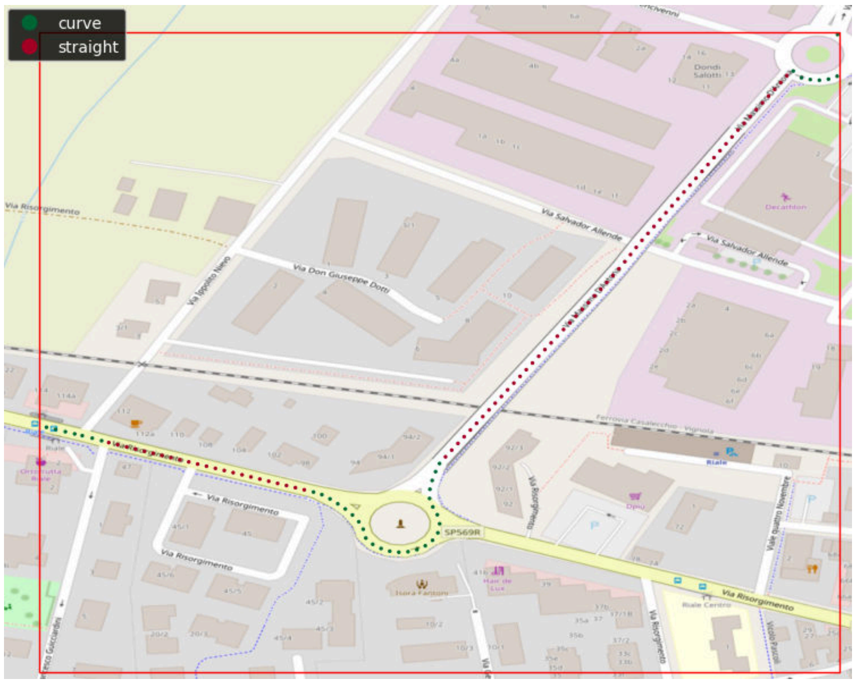


Figure 3.12: Interpolation examples with sample points highlighted, with a sampling distance of 5 meters.

Finally, this process also interpolates elevation data (see figure 3.13). The height of samples between points is always interpolated linearly, rather than treating each point as a 3D point during line-curve interpolation, to have a better accuracy to the real overhead road geometry in curve interpolation that could be worsened by introducing a third dimension to the Bézier curve: the accuracy to the 2D shape of the road has a higher priority than fidelity to the height data.

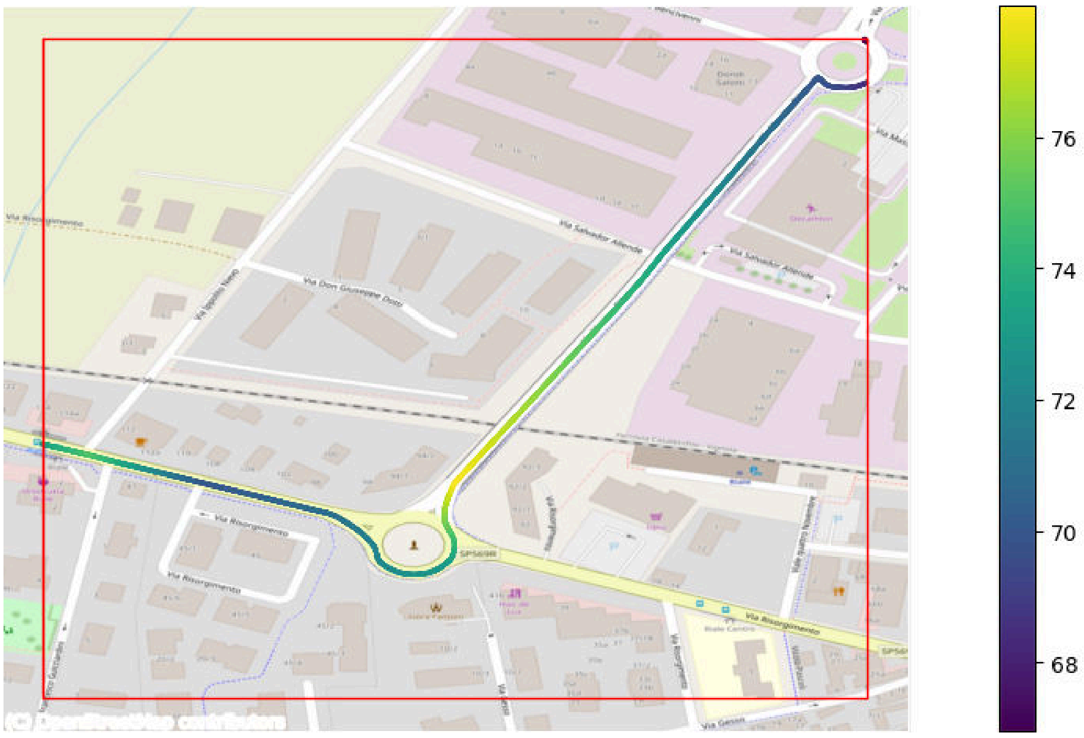


Figure 3.13: Example of elevation interpolation (represented in meters)

3.5 Support metadata

Other than the metadata used during the curve labelling process in interpolation, we also include other data in the output table to be used in following processing steps as required. In particular, we include:

- The **speed limit** of each passed road, necessary to simulate the speed of the vehicle. As mentioned in Chapter 2, OSM data often doesn't include the `maxspeed` tag when it is equal to the legal standard of that road type. Thus, we fill missing speed data depending on standard legal maximum speed values.
- The **curvature** of the route at each point (see below).
- The **label** of each road, not only distinguishing line and curve tracts but also marking roundabouts, which are treated differently in the forces calculation.
- The index in the table of the first sample of the line/curve segment each sample belongs to.

In general, the package allows to easily include additional OSM way attributes in the output. For example:

```
sampled_gdf = interp_dir.get_points(
    [...],
    gdf_columns=['base_idx', 'junction', 'speed_kph'],
)
calc_curvature(sampled_gdf)
interp_gdf_to_csv(
    sampled_gdf,
    [...],
    extra_cols=['speed_kph', 'curvature'],
)
```

If one wanted to include the OSM Way `highway` attribute representing the road type, this would be the required code:

```
sampled_gdf = interp_dir.get_points(
    [...],
    gdf_columns=['base_idx', 'junction', 'speed_kph', 'highway'],
)
calc_curvature(sampled_gdf)
interp_gdf_to_csv(
    sampled_gdf,
    [...],
    extra_cols=['speed_kph', 'curvature', 'highway'],
)
```

Curvature is defined as $1/R$, where R is the radius of curvature, the radius of the circle tangent to the current point of a continuous curve. The radius of curvature for a parametrical curve is:

$$R = \left| \frac{(\dot{x}^2 + \dot{y}^2)^{3/2}}{\dot{x}\ddot{y} - \dot{y}\ddot{x}} \right|$$

Since we have a discrete set of points, we approximated the curvature using the discrete gradient as the derivative, and using the index in the list of points as the parameter, providing a "good enough" result from a practical standpoint for this usecase. Curvature is measured here on the 2D curve of points, not including the elevation value.

3.6 Usage

We will give a final overview of the usage of the `route2vel` package as part of the Acc2Twin pipeline. First, the routing service OSRM must be started with the **run_osrm** script, then in the program the configuration files of the package should be loaded or initialized as follows:

```
import route2vel

route2vel.load_config(".")
```

`route2vel.load_config` will either create or load the *json* file containing the package's config in the current folder. Then, as mentioned, the `RichDirection` and `InterpolatingDirection` classes are the main access point of the package, instantiated as in the following:

```
route_dir = route2vel.find_route_osrm
  ["Bologna", "Modena"],
  load_graph=True, load_graph_name = "test_graph"
)
interp_dir = route2vel.interp_from_route(route_dir)
```

The first function will call the routing service, and return a RichDirection object. It wraps a `routingpy` Direction object, and includes additional functions to get the list of OSM nodes and additional data. The second function tags the route into lines and curves as previously explained, and offers functions to sample points. After that, interpolation can be executed in one of the following ways:

```
sampling_distance = 5
sampled_gdf = interp_dir.get_points_with_density(
  sampling_distance, return_gdf=True, in_meters=True,
  gdf_columns=['base_idx', 'junction', 'speed_kph'],
)
# or one of the following
num_points = 100
sampled_gdf = interp_dir.get_points_with_num(num_points, ...)
t = np.array([0, 0.2, 0.5, 0.9])
sampled_gdf = interp_dir.get_points(t, ...)
```

As a final step, the resulting CSV can be created with the `interp_gdf_to_csv` function.

```
from route2vel.postprocess import calc_curvature, interp_gdf_to_csv
calc_curvature(sampled_gdf)
interp_gdf_to_csv(
  sampled_gdf, "output.csv"
  # copies OSM way data and other columns into the final CSV
  extra_cols=['speed_kph', 'curvature']
)
```

The function `calc_curvature` allows to add a curvature column to the final data. `interp_gdf_to_csv` also allows additional parameters to further modify the final output, such as `separate_roundabout` to tag roundabouts differently from curves.

3.7 Processing steps following our work

As our work was a collaboration with the Department of Industrial Engineering, the steps done on their side following our part of the processing will be quickly overviewed for context.

Starting from our processed data, consisting of a series of geographical points with the metadata detailed in the previous section, the route is split in additional groups depending on the direction of linear segments, and additional categorizing is done on the start and end points of each final segment.

Depending on maximum speed and curvature, the target speed (not necessarily equal to the maximum speed) of each segment is computed, and together with the rate of acceleration and deceleration between each segment, first using only latitude and longitude and then considering elevation.

Starting from the knowledge of the target speed and the acceleration/deceleration rates, the kinematics of the vehicle is reconstructed as a function of time. As a last step, the final output is represented by the tangential and normal acceleration components, used in the packaging Digital Twin to simulate the damage the payload may endure during the transport.

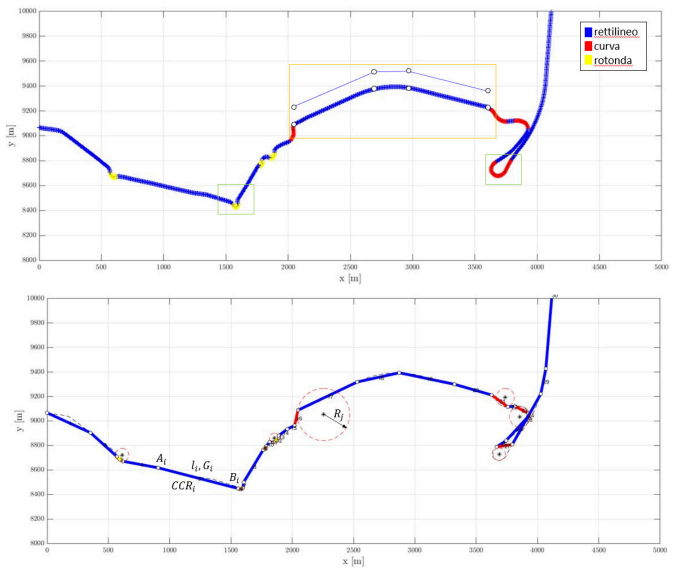


Figure 3.14: Example of further processing from our data.
Courtesy of Roberto Di Leva (UNIBO). Top: categorizing of different segments, bottom: curvature calculations.

3.8 Experimental results

As a ground truth for comparision wasn't available, we couldn't rigorously test the accuracy of the interpolation other than comparing our results with satellite images visually (see figure 3.15).

We provide a simple testing Web-UI (see figure 3.16), which can be launched through the **launch-webui.ps1** and **launch-webui.sh** scripts as previously written, that allows to specify two locations either by name, address or coordinates, a point density for sampling, and a output CSV file path. The application then runs the route navigation, and interpolates points in the route at a specified density, saving the results in a CSV file that also includes metadata requested by our colleagues from the Department of Industrial Engineering. The CSV file is then used in further processing to calculate forces undergone by the vehicle payload, and then evaluate an optimal packaging amount. See table 3.1

Lat (m)	**Lon (m)**	**Ele (m)**	**Is curve**	**Start index**	**Max speed**	**Curvature ($\mathbf{m^{-1}}$)**
366.71	339.36	70.01	line	2796	50.0	1.451×10^{-7}
366.77	339.46	70.00	line	2796	50.0	1.727×10^{-1}
366.82	339.56	70.00	line	2796	50.0	6.504×10^{-1}
366.88	339.64	70.00	roundabout	5676	90.0	1.172
366.94	339.70	69.99	roundabout	5676	90.0	1.203
367.01	339.76	69.99	roundabout	5676	90.0	1.159

Table 3.1: Extract from sample output CSV file

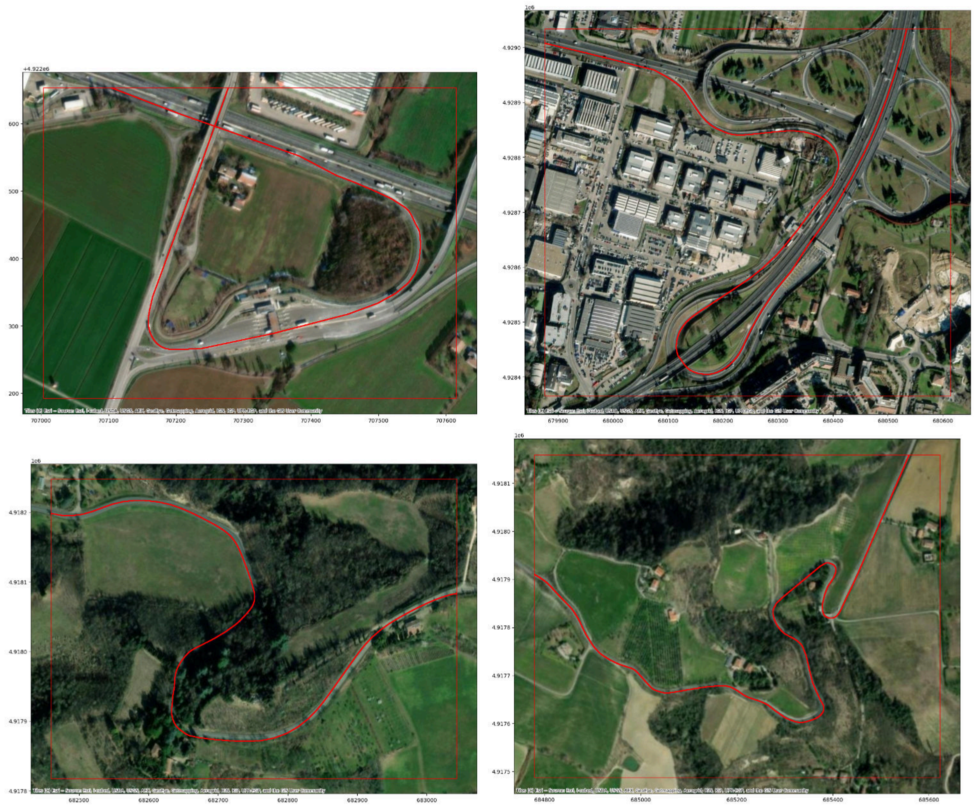
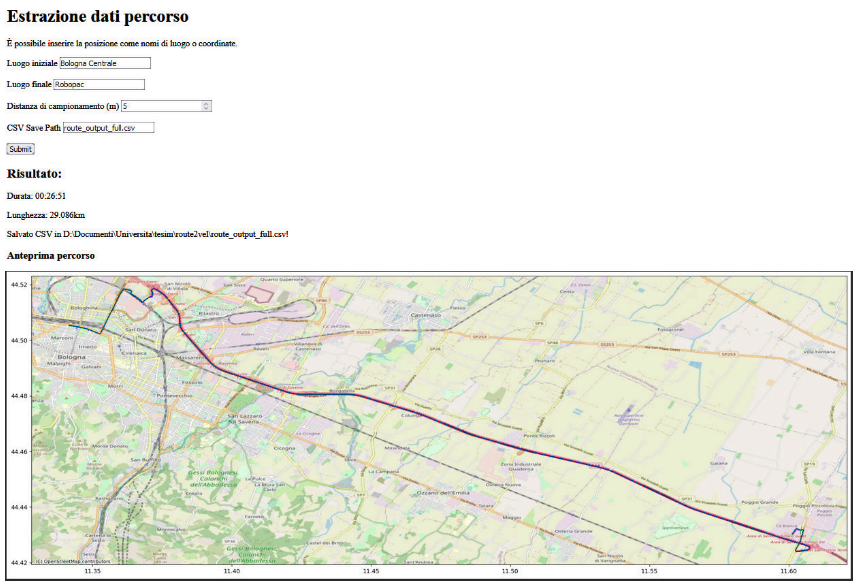


Figure 3.15: Examples of route interpolation compared to satellite images from ESRI World Imagery
Top images taken from route from Zola Predosa, BO to Robopac, bottom images taken in route from (near) San Lazzaro, BO to Sasso Marconi, BO



Estrazione dati percorso

È possibile inserire la posizione come nomi di luogo o coordinate.

Luogo iniziale Bologna Centrale

Luogo finale Robopac

Distanza di campionamento (m) 5

CSV Save Path route_output_full.csv

Risultato:

Durata: 00:26:51

Lunghezza: 29.086km

Salvato CSV in D:\Documenti\Universita\tesim\route2vel\route_output_full.csv!

Anteprima percorso

Figure 3.16: Sample of the testing Web-UI
From top to bottom: start and destination inputs, sample point distance, output file path; then, results containing route duration, result file path, and a route preview

for an extract of the output of our step of the pipeline.

Chapter 4

ParallelTwin

As previously written in 2.3, the Eclipse SUMO software was chosen for the multi-vehicle simulation work associated with this thesis, to act as a Digital Twin of real traffic for the purpose of analyzing it and environmental phenomena of interest in future research, such as pollutants emitted by vehicles per area and traffic bottlenecks.

One notable issue with it from testing, however, is its lack of parallelization: the simulation runs in one process and mostly one thread, leaving the CPU under-utilized.

To make a better usage of the system resources, we developed a wrapping software to SUMO, ParallelTwin [48], overhauling an existing simpler implementation by Phillip Taylor, ParallelSumo [49], with the purpose of using more processes to run the simulation: reducing the overall running time, while striving to maintain same or similar levels of accuracy in the simulation results.

4.1 Goals

We aimed for three goals when developing ParallelTwin.

1. Node computation balance. To have a good performance, the load of each process running the simulation should be as balanced as possible, avoiding bottlenecks. We measured this using the amount of vehicles each process simulates over time, as they are the bulk of the simulation workload.

2. Avoiding SUMO modification. To make the application lightweight and especially forward-compatible, it only uses the existing SUMO software (mainly through its LibSUMO interface), without modifying its source code as other implementations of SUMO parallelization did.

3. Reduced partition overlap. To keep a small error from the ground-truth (single-process) output, the street network partitioning should keep the number of connecting roads between partitions low, as having frequent interactions between partitions may lead to edge-cases and general poor performance.

1 and 3 depend on the partitioning algorithm and parameters used in dividing the road network used in the simulation, and depending on the method used could be partially mutually incompatible. Also, an important point is that 2 requires having a static partitioning, as changing the road network mid-simulation is not possible in base SUMO, as would be the case in a hypothetical dynamic partitioning

scenario.

For the work during this thesis, the main focus was on improving the partitioning logic and the high-level communications between partitions; as such, the performance of the inter-process communication itself had a lower priority, as will be noticeable in the test results, since it mostly depends on lower-level implementation. We left the improvement of the communications implementation as a future goal.

4.2 General overview

The software is structured in three C++ executables, *ParallelTwin*, *ParallelTwin-Partition*, and *ParallelTwin-Partition-Gui*, and various Python scripts, mainly centered around *createParts.py*.

- **ParallelTwin**: main executable, which runs the partitioning script *createParts.py*, and then runs a process of *ParallelTwin-Partition* for each partition, with a GUI depending on parameters.
- **ParallelTwin-Partition[-Gui]**: the executable file running the simulation for each partition, also referred in this thesis as the *partition manager*. The two executables have the same parameter and purposes, with *-Gui* also opening the SUMO interface to visualize the running simulation. It requires being run by ParallelTwin, and cannot be executed standalone.
- **createParts.py**: run before the simulation to partition the SUMO road network and route files, and generate metadata used in the partition handling, with parameters to configure the partitioning logic and optionally generating graphs for the resulting split. The vehicles in the route files are kept only in the partitiong containing their starting point.

The software can be launched with the included script `./launch.sh`; it runs ParallelTwin and sets up the Python environment, as it also needs to run the partitioning script. An example run command is

```
./launch.sh -c simulation.sumocfg -N 4 --gui
```

to launch the software with 4 partitions, and the GUI enabled. A **--help** argument is provided to list available options. A full documentation of available options for the programs used in this thesis is provided at the end of this document in Appendix A.

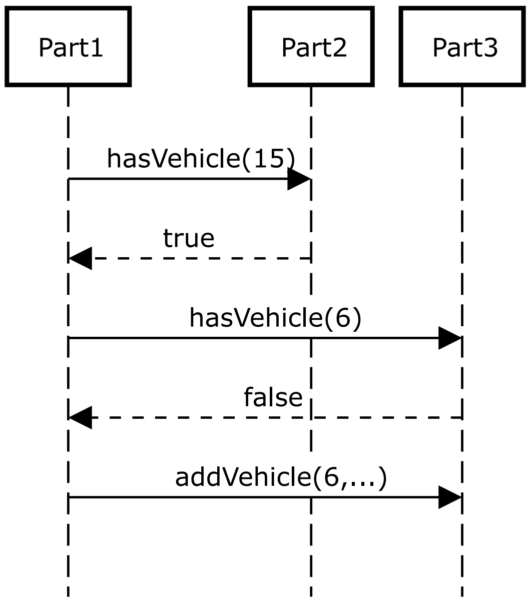
4.3 Inter-process interaction

In making a parallel application, one of the main concerns is choosing between using threads or processes, and how the parallel instances should interact between shared memory or message-passing. As we are working with SUMO, using a process for each simulation instance is required: for obvious reasons if launching the *sumo* and *sumo-gui* binaries, and similarly if using LibSUMO, as its provided functions are static and execute one simulation per process. Thus, this requires some sort of message-passing to be used between the different processes, as shared memory is not available in this case.

The required message kinds between simulation instances, in the current simulation handling logic, are the following:

- **addVehicle**: used to add an outgoing vehicle to another partition, passing its related data (ID, route, type, current position, velocity).
- **hasVehicle**: used before adding vehicles, to check if that vehicle was already added to that partition.

SUMO, as previously mentioned, provides a native communication interface via TraCI, using TCP sockets. Thus, both the original ParallelSumo program and our first implementation relied on it, with the latter not needing additional processes or threads but simply having a single control process communicating with each SUMO instance as a TraCI client, sending messages to exchange vehicles.



Simple interaction diagram between ParallelTwin partition managers

Partition 1 is currently checking whether vehicles in its shared edges should be moved to other partitions

This method, though, proved to have a high overhead, both because of the overhead of using TCP sockets, and because of what kind of messages were required to be passed: as TraCI lacks a *hasVehicle* function in its API, larger amounts of data needed to be passed in each interaction before adding vehicles to run a similar check, by using `getLastStepVehicleIDs` to get the list of all vehicles in the edge of interest and then searching for the current vehicle in the client side, which meant passing a list of strings for every boundary edge in each iteration. Both of these issues resulted in a large enough overhead such that the overall running time of the simulation consistently increased with the amount of partitions.

Thus, we chose to use LibSUMO to run and interface with each simulation instance, using a dedicated message-passing protocol for inter-process communication. This allowed to define the required interaction kinds when not available, running the more data-heavy functions locally without passing larger amounts of data in messages.

As a message-passing protocol, we chose ZeroMQ for this experimental implementation, as it allows for a more robust and simpler communication, with a high performance even if not as high as other, more specialized, alternatives like MPI. As previously mentioned (4.1), in this phase of work the performance of the message-passing protocol itself was at a lower priority than that of the simulation; future improvements will focus on using higher-performance protocols such as MPI.

ZeroMQ

ZeroMQ [50] is an open source high-performance message-passing library, offering both asynchronous and synchronous messaging, aimed for distributed or concurrent applications. It supports a wide array messaging patterns, and various transports: in particular, for this thesis the main transports were the specialized *in-process* and *inter-process* transports, rather than the more expensive TCP and UDP, and the simpler request-reply pattern was enough to handle the current requirements for communications between partitions.

ZeroMQ is named after its design principle of minimalism, for example having no brokers or similar middlewares between endpoints. Various bindings for common languages are provided, wrapping around the low-level *LibZMQ* library. In our case, we used the C++ binding **cppzmq**, being header only and thus simple to include in the build process.

The workflow for ZMQ uses the classical socket pattern: after creating a context which handles the IO threads, sockets are instanced with a specified type, in our case, using *request/reply* sockets between partitions and *pair* sockets inside a partition's threads for interaction handling. Then, sockets are used to send messages containing binary data in their payload.

4.4 Road graph partitioning

The program, as mentioned, splits the workload between simulation instances by partitioning the road network, and the routes and vehicles accordingly. The script that implements this functionality is *createParts.py*, which itself imports functionality from other files:

- *convertToMetis.py*: implements the SUMO network partitioning with the METIS algorithm
- *partroutes.py*: cuts route files according to a partitioned SUMO network file, acting similarly to SUMO's *cutRoutes.py*.
- *partitiondatagen.py*: generates the metadata that the partition managers use during the simulation to handle partition interactions (see section 4.4.4).
- Assorted files for utility (*prefixing.py*, *sumobin.py*) and output generation (*sumo2png.py*).

Given a SUMO network file and M SUMO route files, the output of partitioning for N parts is N network files, and N route files, plus N metadata files to be used during the simulation. The network files are split using the METIS algorithm (see section 4.4.2), with configurable weights, and then the route files are processed to only contain the route segments included in the relevant partition, and the vehicles starting in it. Additionally, a `--png` parameter is available to output an image representing the partitions, as well.

Initially the partitioning process used SUMO's *cutRoutes.py* tool explained in 2.3.1 to split the traffic demand files to the area of interest of each partition. The output wasn't usable for ParallelTwin out of the box, though, as it still included every vehicle in each partition's route file, simply offsetting the departure time in case a vehicle didn't start in the edges of a partition.

As our program handles the transfer of vehicles between partitions, this was not desired behavior and so we needed to filter out vehicles in the resulting partitioned traffic demand files, only keeping those with the earliest starting point (presumably the ones that started in the partition), adding additional overhead.

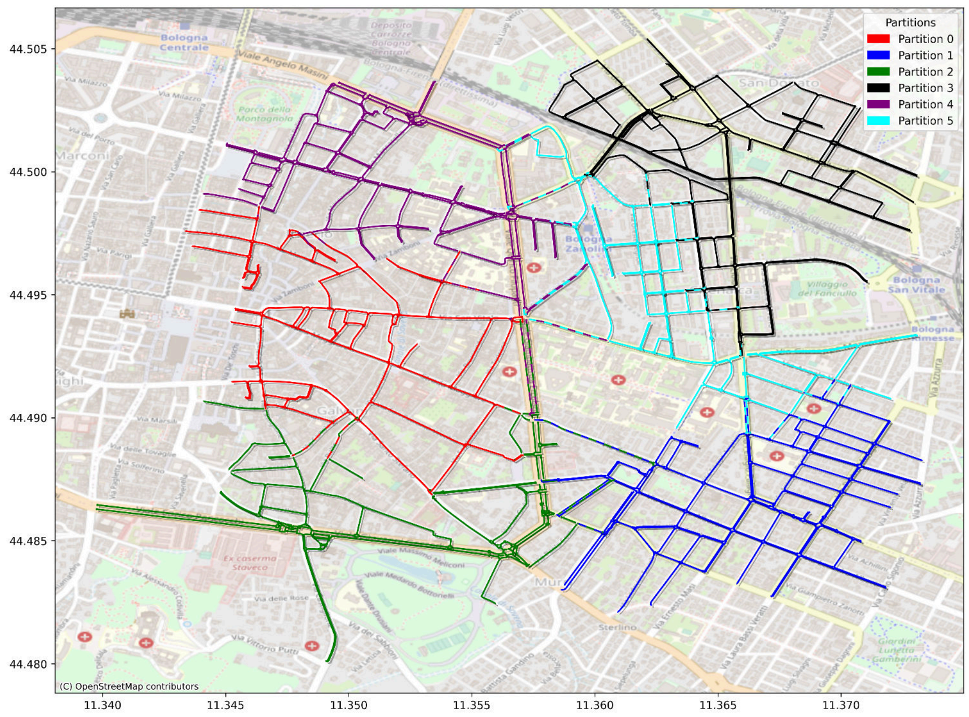


Figure 4.1: Representation of a sample partitioning output

For this reason, *partroutes.py* was developed to produce similar results, without keeping vehicles that do not start in the relevant partition.

This still has some issues, particularly in the edge case in which a vehicle starts in an edge shared between two partitions. This is currently handled by checking all traffic files after their generation, and arbitrarily removing all duplicate instances of a vehicle after the first.

4.4.1 Usage

The script is run by ParallelTwin automatically, and it can be manually run (a utility script *run-with-env.sh* is included to automatically setup the Python environment with the required packages). To pass arguments to the partitioning script when launching ParallelTwin, one can use the “pipe” (**--**) keyword, as such:

```
./launch.sh -c simulation.sumocfg -N 4 -- --png --verbose
```

4.4.2 METIS algorithm

ParallelTwin uses the METIS algorithm by George Karypis and Vipin Kumar [51] [52] to partition the SUMO road network. In particular, we use the multilevel k-way paradigm, which allows to ensure the partitions are spatially contiguous.

The METIS algorithm is a multilevel k-way graph partitioning algorithm. Generally, in this kind of algorithm the graph $G = (V, E)$ is coarsened to to a small number of vertices, a k-way partitioning of the smaller graph is computed, and it is then projected to the original graph by refining the partitioning at

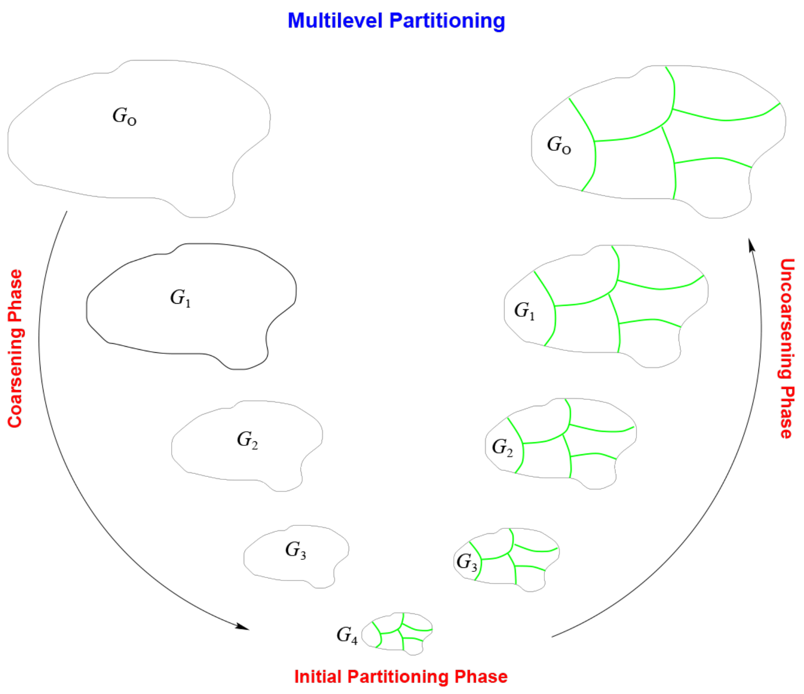


Figure 4.2: Multilevel partitioning. Source: METIS manual [2]

each level of detail (see figure).

An important part of the algorithm is the meaning of the node and edge weights. The algorithm strives to maintain a similar total vertex weight on each partition, as close as possible to W/k, where W is the vertex weight of the original graph. Edge weights are used for one of the possible optimization targets of METIS, which is having a minimum number of edge-cuts, meaning edges that connect vertices of different partitions: the edge-cuts are the sum of the weights of the edges connecting different partitions. Having a low edge-cut number reduces the communication cost for applications that partition a graph to then split the workload on more processors (such as ours), which was also one of the goals of our development as previously stated.

In ParallelTwin, METIS is executed through the Python package `pymetis` in the *convertToMetis.py* script. The package mainly includes a function `part_graph`, that works similarly to the `gpmetis` (graph partitioning) program included in the original METIS software package.

To be able to be processed with METIS, the SUMO road network file is preprocessed in a neighbors-list format, as used in the *xadj* and *adjncy* parameter in the `gpmetis` program[1]. The edge weights are passed in the same manner.

The `part_graph` function outputs a list of integers representing the partition ID of each node of the graph, which we process into a list of edge IDs belonging to each partition.

4.4.3 Partitioning weights

As previously written, the METIS algorithm allows to specify both node and edge weights, aiming to keep a similar total node weight value between the different partitions and avoiding cutting higher-weight

[1]From METIS manual at page 13

edges. We tested various weighting methods, which can be used at runtime using the `-w` (for edge weights) and `-W` (for node weights) arguments. The arguments can be combined, for instance:

```
./launch.sh -c input.sumocfg -N 4 -- -w osm
    -w route-num -W connections
```

Combined weighting methods are added together. We tested the following weighting methods:

Method	Kind	Description
osm	Edge	Give higher weight to main roads as identified by OSM metadata
route-num	Edge	Weight equals the amount of routes passing through the edge
connections	Node	Weight equals the amount of edges connecting to the node
connexp	Node	Same as *connections*, but with an exponent to increase the difference between nodes

4.4.4 Partition processing and route cutting

The route files in SUMO contain a series of XML elements representing routes, simulated actors that can access the simulation (mainly vehicles), actor types, and other kinds of data [53]. The routes can be both explicit *route* elements, containing a list of edge IDs the route passes through, and implicit *trip* elements, which contain a starting and ending route that is navigated at runtime by SUMO. Also, routes can be both defined separately from their vehicles, which reference them via a *route* attribute, as child elements of their vehicles, and in the case of trips with the vehicle (a *trip* contains both route and vehicle data).

For our usecase, we need routes defined separately from their vehicles, as a partition where a vehicle passes through after it started in a different partition will not contain the vehicle in its route file, but must contain its relevant route segment; and the routes must be explicit, as the cutting process needs to be aware of every edge the route passes through.

So, the route files are processed as follows:

1. The implicit trips are converted into explicit routes, using the SUMO binary tool **duarouter**. This step also merges the route files into a singular route file, if there were more input route files specified in the *.sumocfg*.

2. Routes defined as child elements of vehicles are moved out, adding a *route* attribute to those vehicles.

3. For each partition:

    a. The list of edges output by the METIS partitioning is used with the SUMO binary tool **netconvert** to generate a partitioned SUMO network file from the input network.

    b. Cut the previously created route file such that the routes only contain edges within the partition, and vehicles starting in the partition, using *partroutes.py*.

    c. Create a .sumocfg file for each partition pointing to the new network and route files.

4. Consistency is verified: for example, vehicles starting in an edge shared between partitions are reduced to a single copy.

5. Partition metadata is generated using *partitiondatagen.py*.

Step 3 is run in parallel for each partition, as the work on the partition files is independent from the others.

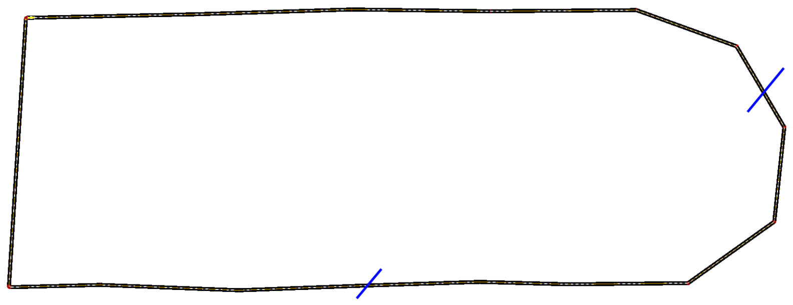


Figure 4.3: Minimal example of a route passing multiple times from a partition. Regardless of the partitioning cut, the circular route will start and end in the same partition.

A particular edge case that is handled both during partitioning and at runtime is that of routes that pass multiple times from a partition (see figure 4.3 for an example). In that case, the route is split into multiple segments in that partition, following the behavior of SUMO's original *cutRoutes.py* script, and at runtime the vehicle will track the last traveled segment when it is inserted again in that partition.

partitiondatagen.py generates a JSON file for each partition, containing metadata used at runtime for knowledge about other partitions. This file contains:

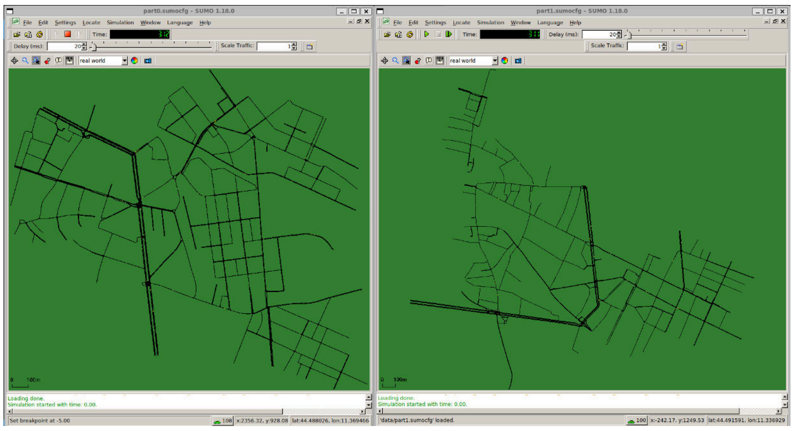
- A list of *border edges* (edges shared with other partitions), including their contained lanes and the edge's partitions.
- A list of neighboring partitions.
- For each neighbor partition, a list of routes that lead into it.
- For each border edge, a list of routes that end into it.

Some of this information can be retrieved with LibSUMO from the local partition, but it was chosen to be included in this format as the cost of adding it while generating the other required metadata was lower than the overhead of using LibSUMO at runtime.

4.5 Simulation runtime

Overall, the software after partitioning works as following. After the simulation starts, each partition manager alternates running steps of the simulation with checking the edges of the road graph shared between partitions for outgoing vehicles. Vehicle routes are pre-fixed, and have metadata to mark those that continue in other partitions. Only vehicles with routes continuing to the partition sharing the edge are sent to that partition.

Note that since the vehicle is created in the new partition as soon as it enters the shared edge, the vehicles will be "mirrored" between partitions on that edge, until the vehicle exits the previous partition.


part0.sumocfg - SUMO 1.18.0
part1.sumocfg - SUMO 1.18.0
File Edit Settings Locate Simulation Window Language Help
Time:
Delay (ms):
Scale Traffic:
real world
0 100m

```
Loading done.
Simulation started with time: 0.00.
Set breakpoint at -5.00    108  x:2356.32, y:928.08  lat:44.488026, lon:11.369466
```

```
Loading done.
Simulation started with time: 0.00.
'data/part1.sumocfg' loaded.    100  x:-242.17, y:1249.53  lat:44.491591, lon:11.336929
```

Figure 4.4: SUMO GUI with more partitions open

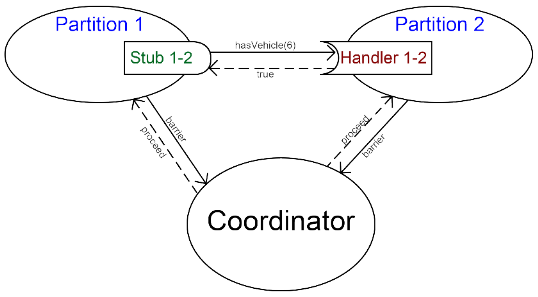
This is intentional, and has the positive effect of reducing the loss of traffic data used in the other vehicles' simulation: as previously written in 2.3.3, SUMO normally tracks the following and preceding vehicles in the vehicle acceleration behavior to have a more accurate rendition of its movement; since partitioning the simulation in more instances removes this knowledge for vehicles near the edge, keeping the vehicles in the previous partition as long as possible reduces this effect, and helps in maintaining accuracy.

Currently partitions wait on a barrier for every partition to be done with a simulation step before continuing to the next, to avoid possible loss of accuracy and concurrency issues. The overall simulation ends with conditions analogous to the one in the input SUMO simulation config: either a fixed end time, or once all partitions are devoid of vehicles and do not have any future vehicle departures.

4.6 Software structure

The software is comprised of four main classes, and various support files. The main classes are:

- **ParallelSim**: the class acting as the coordinator of the system; it runs the partitioning script, then generates the partition manager processes by running the *ParallelTwin-Partition* binary. During the simulation, partitions communicate with it to synchronize after each step, acting as a barrier through messaging.
- **PartitionManager**: one instance for each partition (thus one instance per process), handles the simulation itself using LibSUMO, and the management of outgoing vehicles.



Representation of interaction example between system components

- **PartitionStub**: N instances per partition, offers an abstraction for messaging to a specific partition with an interface equivalent to the corresponding functions of that PartitionManager. Currently offers *addVehicle* and *hasVehicle*, plus other functions used in debugging.
- **NeighborHandler**: N instances per partition, the peer to each client PartitionStub, resolving its requests. Read operations are resolved immediately, while write operations (addVehicle) are queued and resolved before the start of the next partition step to avoid concurrency issues.

Support files include *args.hpp*, which uses the *argparse* library for C++ to implement the execution arguments, and various utility files. Additionally, the *TinyXML2* library is used to process SUMO xml files.

Each process contains more threads: NeighborHandler uses an internal thread for each instance to allow for concurrent listening to the requests of other partitions, thus also using a separate ZeroMQ context for each. Additionally, ZeroMQ contexts contain one IO thread to handle messages. The thread in each NeighborHandler communicates with the main thread also by use of specialized ZeroMQ sockets (*inproc* transport).

4.7 Test results

To test the performance of the parallelization, we ran ParallelTwin through two types of tests: overall duration of the simulation (and gain compared to the single-core monolithic case), and balance between the partitions in workload.

For testing purposes, the *ParallelTwin* binary offers various utility arguments:

- `--pin-to-cpu`: pins each partition process to a specific CPU, to ensure tests remain representative and are not influenced by the operating system
- `--log-handled-vehicles`: outputs a CSV with the amount of vehicles included in each partition at each simulation step.
- `--log-msg-num`: outputs a CSV with the amounts of messages sent and received for each partition at each simulation step.

Scripts were also included to gather test results of each partition (*gather-msgcounts.py*, *gather-stepvehicles.py*, and *gather-times.py*) and to launch the test configurations and gather their data (*performance-measures-launch.sh* and *test_partitions.py*).

4.7.1 Partitions workload balance

We tested every combination of our devised weighting methods mentioned earlier in section 4.4.3. For each test, we considered the vehicle counts and simulation times. Our objective was to achieve balanced vehicle numbers and simulation times across partitions at each time step. To quantify this, we utilized the standard deviation of vehicle numbers and simulation times between partitions. Since vehicle numbers are recorded at each time step, we calculated the average standard deviation across time steps for vehicles between each partition. Given the vehicle count over time matrix v_{ti}, where i is the partition number, N partitions, and a simulation duration of T, the average standard deviation score $\overline{\sigma_v}$ is calculated as follows:

$$\sigma_{v_t} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v_{ti} - \overline{v_t})^2}$$

$$\overline{\sigma_v} = \frac{1}{T}\sum_{t=1}^{T}\sigma_{v_t}$$

Additional scores were calculated as the maximum difference between vehicle numbers of each partition (averaged across time) and maximum simulation time.

The tests were done with four partitions, and using a simulation of a section of the city center of Bologna generated using SUMO's OpenStreetMap extraction tool (see figure 4.5), to have a realistic map.

Specifically, this simulation was generated with these steps:

- Using the OSM Web Wizard, the eastern half of Bologna's city center was selected (exact parameters are not available in the OWW).
- The generated traffic file was removed, and a new one was generated with SUMO *randomTrips*, with the following parameters: `-n <file> -r osm.passenger.trips.xml -e 3600 --fringe-factor 20 --random-depart`
- Duarouter was used on the resulting file.

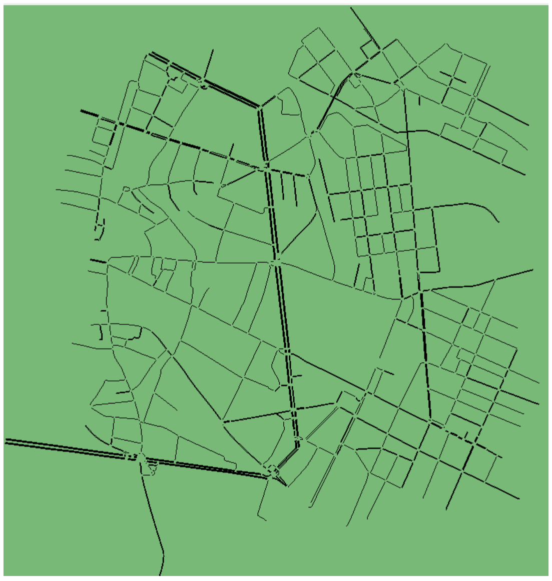


Figure 4.5: Area used in the partitioning tests

Weights used (Edges and Vertices)	**V.num** σ	**V.num max diff.**	**sim.** t σ	**sim.** t **max**	**comm.** t σ	comm. t max
E: V:	19.08	41.77	0.68	3.23	1.06	5.29
E: V: connections	19.28	40.11	0.71	3.02	1.60	6.04
E: V: connexp	17.31	37.66	0.52	2.94	2.15	7.14
E: V: connections,connexp	17.31	37.66	0.49	3.07	2.06	7.24
E: route-num V:	18.56	39.32	0.66	2.95	1.47	5.94
E: route-num V: connections	10.50	23.27	0.35	2.48	0.87	6.40
E: route-num V: connexp	15.75	35.82	0.53	2.60	1.27	7.36
E: route-num V: connections,connexp	15.75	35.82	0.49	2.41	1.23	7.20
E: osm V:	16.82	35.88	0.84	3.24	1.49	6.17
E: osm V: connections	18.36	39.84	0.72	2.90	1.49	4.67
E: osm V: connexp	18.17	39.37	0.68	3.06	1.43	7.65
E: osm V: connections,connexp	18.17	39.37	0.69	3.21	1.44	7.79
E: route-num,osm V:	16.82	35.88	0.80	3.16	1.48	6.15
E: route-num,osm V: connections	18.36	39.84	0.71	2.91	1.51	4.71
E: route-num,osm V: connexp	18.17	39.37	0.72	3.55	1.43	7.92
E: route-num,osm V: connections,connexp	18.17	39.37	0.68	3.08	1.38	7.58

Table 4.1: Partition test results.
Green are best (lowest) results, red are worst. Note that for vehicle number, results are averaged across the time duration.
σ: standard deviation.
t: elapsed time

As seen in the table, the best combination of weights was using *route-num* weighting for edges, and *connections* weighting for nodes (figure 4.6), having both the smallest deviation in simulation times and vehicle counts, and the smallest maximum vehicle count difference (figures 4.7 and 4.8).

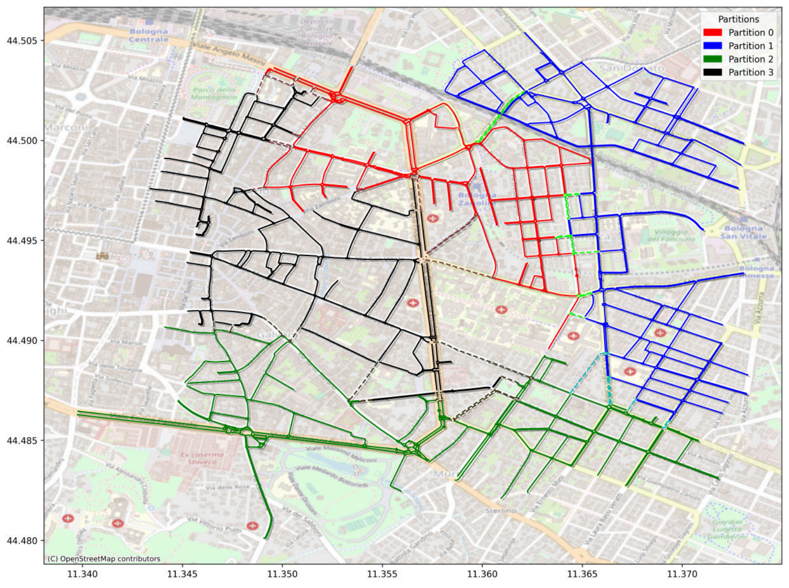


*Figure 4.6: Visualization of best partitioning scheme (*route-num + connections*) during tests*

Communication times, which also includes execution of vehicle transfer and generally LibSumo functions, were also measured, but were not considered for the purpose of this test as they mainly depend on the implementation and chosen framework for inter-process communication, rather than the partitioning logic. It can still be useful to note how the weighting chosen as best also has the lowest deviation in communication times; and how inter-process communication is the most taxing part of the

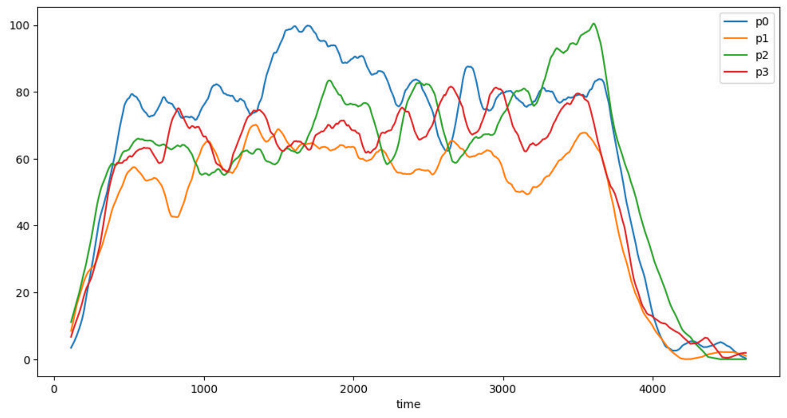


*Figure 4.7: Vehicle counts across simulation time steps in best partitioning scheme (*route-num + connections*) from tests*

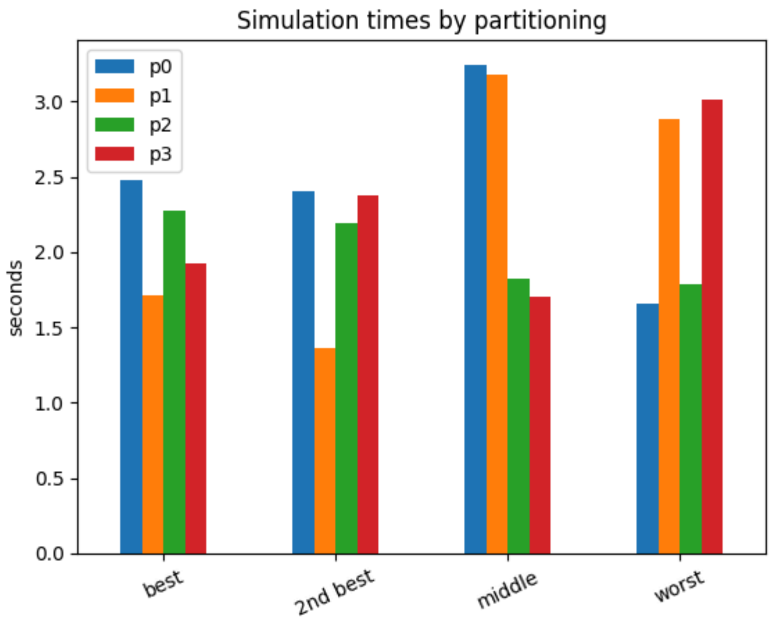


Figure 4.8: Simulation times across partitions from select partitioning schemes.
Best: route-num + connections
2nd best: route-num + connections + connexp
Middle: standalone osm
2nd best: standalone connections

current version of the software, taking more than half of the total elapsed time. This is to be expected with the lower focus placed on the communications performance as previously mentioned, and can be improved with future developments (for example, by using MPI, and switching to an observer pattern when applicable).

4.7.2 Simulation times

We then tested the simulation times, to ensure an improving performance with a growing number of partitions in a multi-core CPU. The test was done on a 12-core CPU (AMD Ryzen 5 3600X), varying the amount of partitions between 1, 2, 4, 6 and 12, and testing multiple road networks.

We tested on both real road networks obtained from OpenStreetMap, and sample large road networks generated with SUMO's `netgenerate` program, using two traffic configurations for each. We made a simple script, *randomSimulation*, to run netgenerate and randomTrips in sequence with both traffic configurations.

In more detail, the test road network were generated as such:

- **bologna-sim**: road network generated with the OSM Web Wizard with the same parameters as the simulation in 4.7.1
- **bologna-metropolitan-area**: road network generated with the OSM Web Wizard by including a large area that covers Bologna's center and its surrounding city sectors (again, precise parameters are not available in the OSM WW)
- **spider0**: generated with netgenerate from the following parameters:
  `--spider --spider.omit-center --spider.circle-number 3 --spider.arm-number 5`
- **spider1**: generated with netgenerate from the following parameters:
  `--spider --spider.omit-center --spider.circle-number 15 --spider.arm-number 8`
- **spider2**: generated with netgenerate from the following parameters:
  `--spider --spider.omit-center --spider.circle-number 50 --spider.arm-number 9`
- **grid_large**: generated with netgenerate from the following parameters:
  `--grid --grid.number 100 --grid.length 20`

The netgenerate road networks are not representative of real irregular road networks, but work well for stress/performance testing.

After generation, randomTrips was executed twice for each network, generating medium-intensity and low-intensity traffic files with these parameters:

```
python "$SUMO_HOME/tools/randomTrips.py" -n "$NETFILE" -r "$ROUFILE" -e 3600 --fringe-factor 20 --ra
python "$SUMO_HOME/tools/randomTrips.py" -n "$NETFILE" -r "$ROUFILE2" -e 3600 --fringe-factor 20 --r
```

`--period` is the average time period between each departure (in seconds). The end time of 3600 is in seconds, and represents a duration of 1 hour in simulation-time.

The characteristics of each simulation are overviewed in table 4.2; the low traffic simulations have on average a 10 times lower vehicle count due to the higher depart period.

Each configuration was simulated 10 times, with the results then averaged across the various repeats.

	nodes	edges	vehicles
cfg			
bologna-metropolitan-area/osm.sumocfg	17255	36517	3546
bologna-metropolitan-area/osm_lowtraffic.sumocfg	17255	36517	345
bologna-sim/osm.sumocfg	473	852	2757
bologna-sim/osm_lowtraffic.sumocfg	473	852	343
grid_large.sumocfg	10000	39600	3600
grid_large_lowtraffic.sumocfg	10000	39600	360
spider0.sumocfg	15	50	3600
spider0_lowtraffic.sumocfg	15	50	360
spider1.sumocfg	120	464	3600
spider1_lowtraffic.sumocfg	120	464	360
spider2.sumocfg	450	1782	3600
spider2_lowtraffic.sumocfg	450	1782	360

Table 4.2: Properties of the simulation configurations used in the elapsed times test

The target ideal result is each configuration resulting in a lower overall time with the increase of partition numbers, thus implying the gain from partitioning to be higher than the loss from inter-process communication.

Bologna-sim

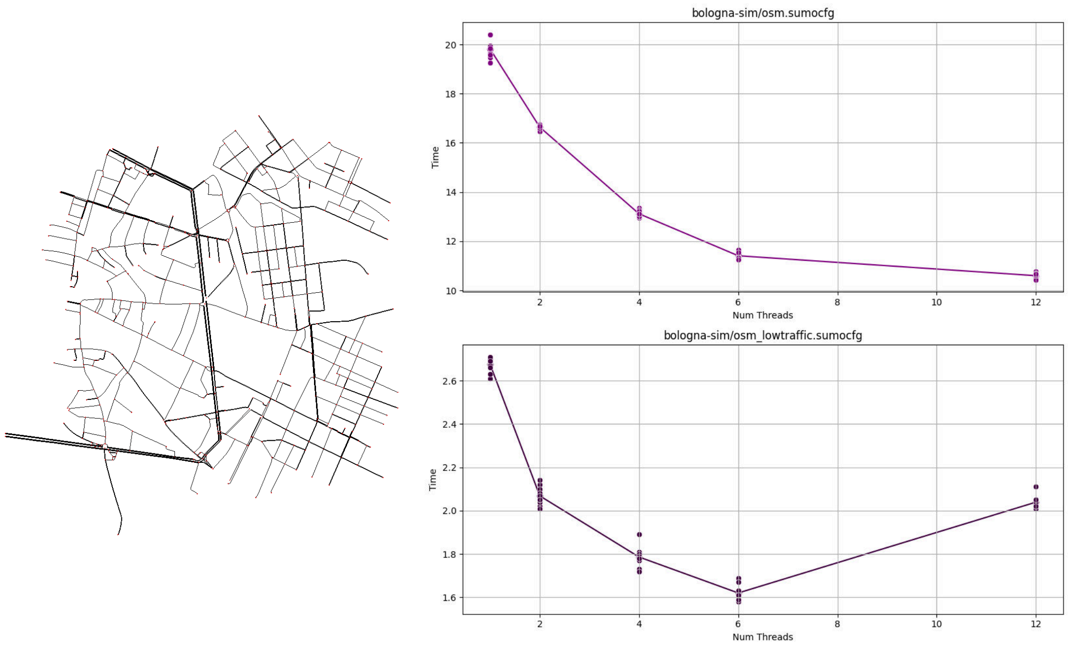


Figure 4.9: Map and results of elapsed time testing with the bologna-sim simulation. On left: road network used for the simulation. On top right: simulation time results in seconds by partition number on normal configuration. On bottom right: results on low traffic configuration.

With the simulation used in the previous tests, results were satisfactory, with the elapsed time lowering on increasing the amount of partitions, except from the maximum number of threads not giving a gain in time with a low traffic amount.

It can be inferred that this is caused by the increased overhead in communications with more processes being higher than the gain in the handling of simulation workload with a lower overall workload (amount of vehicles).

Bologna-metropolitan-area

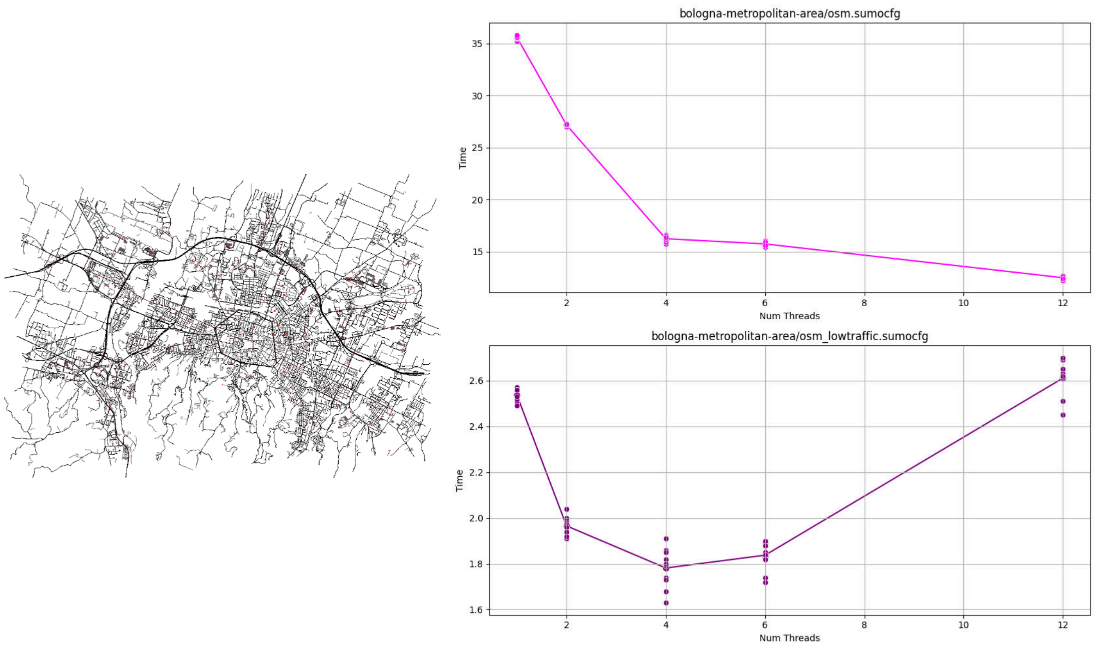


Figure 4.10: Map and results of elapsed time testing with the bologna-metropolitan-area simulation. On left: road network used for the simulation. On top right: simulation time results in seconds by partition number on normal configuration. On bottom right: results on low traffic configuration.

With the larger simulation of the metropolitan area of Bologna, results were still satisfactory with a ”normal” (one vehicle per second) traffic amount, but less stable with a lower amount of traffic. Despite CPU pinning, parallelization induces an inherent random effect on results, that was especially notable with four partitions in this instance.

The loss due to inter-process communication overhead in this case was worse, with the optimal peak at four partitions on average, despite the variance in results, and worse simulation times with 12 partitions than 1.

Spider0, spider1, spider2

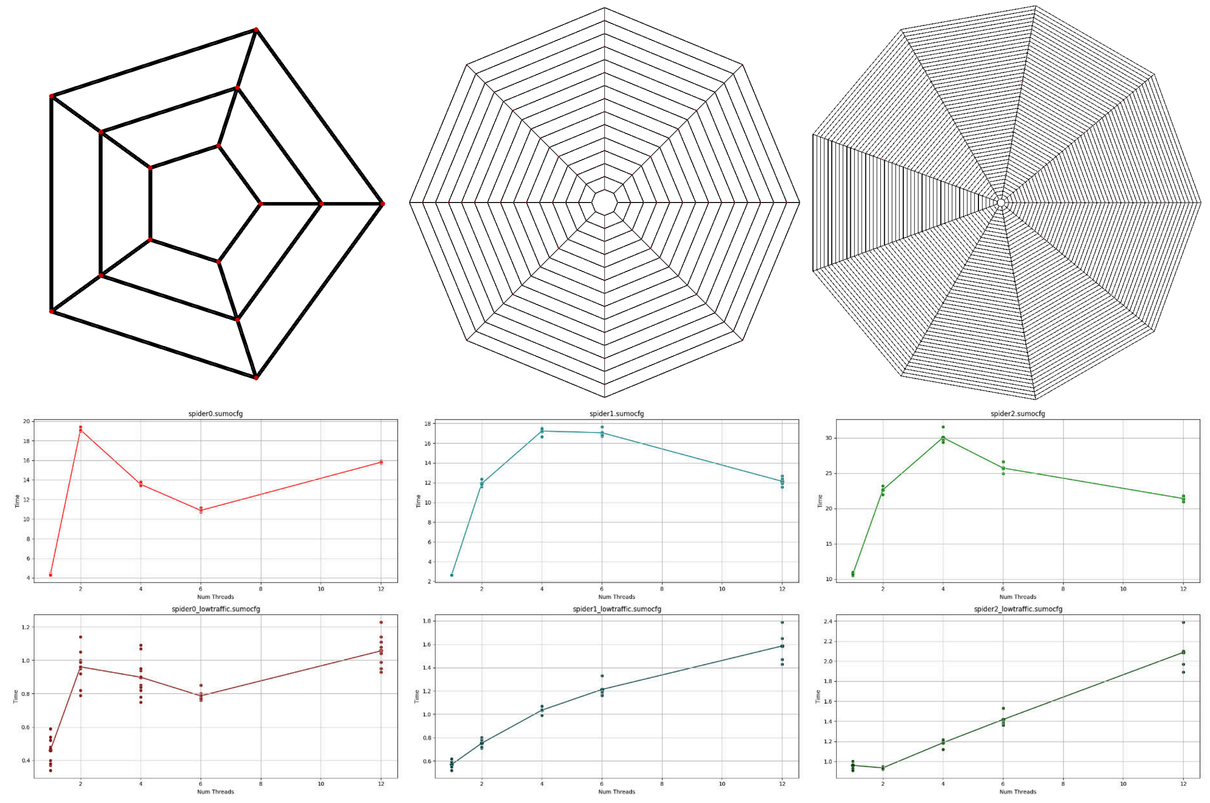


Figure 4.11: Map and results of elapsed time testing with various SUMO netgenerate spider configurations.
On top: map layouts, on center: simulation time results in seconds by partition number on normal configuration, on bottom: results on low traffic simulations. From left to right: spider0, spider1, spider2. Simulations were generated using SUMO's netgenerate tool, with the `--spider` generation mode.

Using an artificial, regular web-shaped road network generated with SUMO's *netgenerate* tool with the `--spider` generation settings, results were well below expectations, with the monolithic one-partition case performing best in all cases sans one (where the second-lowest partition number was instead the optimal amount), for both the high-traffic and low-traffic cases.

Surprisingly, the small network spider0 (which is orders of magnitude smaller than an ordinary SUMO simulation) wasn't the worst performer, which was expected due to the small network size making the gain from partitioning unnecessary and overshadowed by the loss from inter-process communications.

The reason of the low performance with these road networks can be amounted to regular networks being unable to be partitioned in such a way to reduce boundary edges and vehicle crossings between partitions, which are the main factors for higher communication volumes.

Grid_large

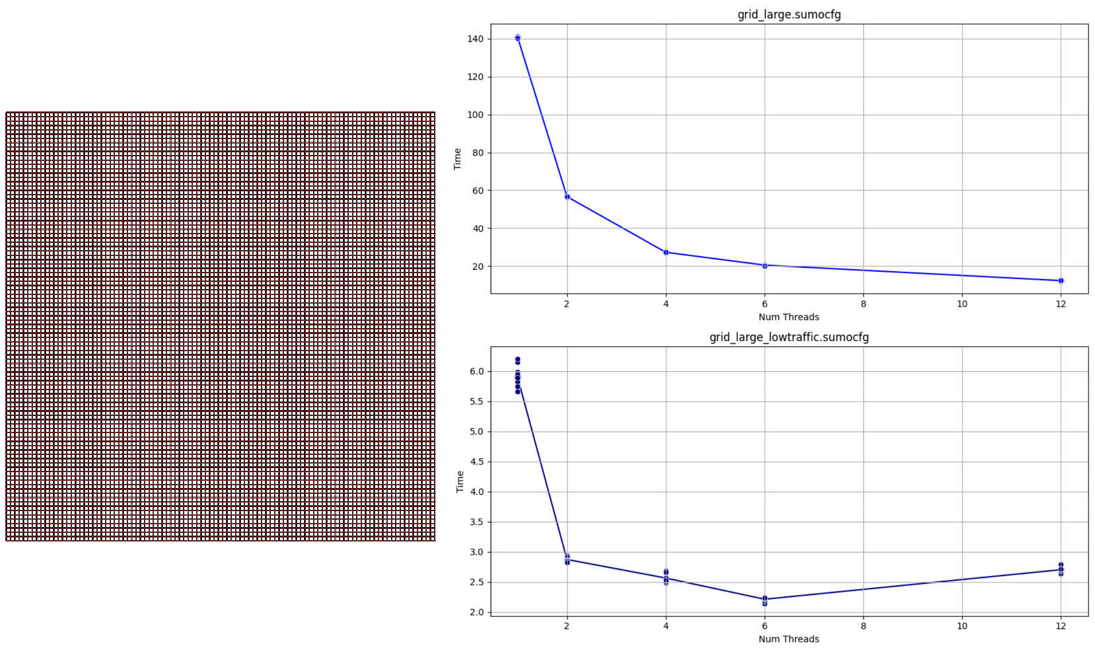


Figure 4.12: Map and results of elapsed time testing with the grid_large simulation. On left: road network used for the simulation. On top right: simulation time results in seconds by partition number on normal configuration. On bottom right: results on low traffic configuration. Simulations were generated using SUMO's netgenerate tool, with the `--grid` *generation mode.*

Using another artificial, regular road network, in this case generated with SUMO's *netgenerate* tool with the `--grid` generation settings, and a larger size than all *spider* configurations, led to far better results than with the previous configurations, with the elapsed times consistently lowering with increased partition numbers.

This can be inferred to be caused by the large size of the simulation emphasizing the gain from the partitioning and thus reducing the relative loss from communications.

Overall results

With normal simulations and a medium-high traffic amount results were satisfactory, with the simulation elapsed time lowering with increased partition numbers, sometimes reaching a peak after which elapsed time increased with amount of partitions.

Results seem to have a higher variance with a lower traffic density, and seem to be less than satisfactory when using small or non-realistic regularly-shaped simulations, as they do not allow for an efficient partitioning that reduces the contact points between partitions. This isn't much pressing of an issue, as the priority is on the efficiency of simulation of real networks.

Despite this, as mentioned in section 4.7.1, communication times are still far from optimal, usually taking more than half of the total elapsed time. Thus, optimizing this aspect is still an important future goal.

Chapter 5

Conclusions and future developments

In conclusion, this thesis presents an examination of *Digital Twin* (DT) technology and its potential in the context of urban development, with a focus on the optimization of traffic analysis in smart cities and transport packaging optimization for waste reduction. We detailed the work on two particular use cases of this technology, **Acc2Twin** and **ParallelTwin**.

Acc2Twin is a pipeline to use a transport pallet DT to optimize packaging plastic film usage, in which we created **route2vel**, a Python package that led to the successful creation of a continuous road network topology critical to simulating real-world transport conditions, starting from *OpenStreetMap* data. Meanwhile, the ParallelTwin use-case highlights the advantages and disadvantages of parallel simulations in optimizing vehicular traffic analysis, by developing a parallel solution to the *SUMO* traffic simulation software package, with a focus on efficient road network partitioning to obtain a balanced workload between nodes.

The results from our work in both study cases were mostly satisfactory: route2vel allows to interpolate and sample points at an arbitrary precision from road data, and was used successfully as part of the Acc2Twin pipeline to aid in computing the forces the transported payload will endure.

The results of the SUMO wrapper for the ParallelTwin case were also satisfactory for the goal of balancing the simulation load between partitions, being able to find a balanced partitioning, but not as satisfactory in regards to overall elapsed time and optimization. The focus was mainly on investigating optimal partitioning schemes, and devising a working implementation of SUMO parallelization, and thus less on the optimization on the inter-process communication. The performance of the final program is not optimal on that regard, with often more than half of the elapsed time being spent on the inter-partition interactions. Thus, optimizing the inter-process communication remains as an important future goal.

Future goals

Acc2Twin

The following possible future developments apply only to our Python package (*route2vel*), not the whole Acc2Twin project.

- Using a local Docker container for the elevation data service is possible, and could be easily done by changing the elevation loading code. This would allow for far better performance, instead of using the public API used in development.
- Ideally, both the route calculation and metadata should use the same source (*osm.pbf* files), instead of using local files with OSRM and the Overpass API for OSM metadata through the `osmnx` package.
- A proper interface could be built, either improving on the current development Web-UI or building a GUI from the ground up. In either case, the goal should be user-friendliness to allow for easier data processing for non-developers.
- After consulting with the end clients, a specialized driving profile for trucks and other transport vehicles should be created to be used in OSM, as mentioned in Section 3.3.

ParallelTwin

- As already mentioned, the actual inter-process communication should be optimized. This can be in two main approaches: switching to different IPC frameworks, such as MPI, and spending further time on optimizing the communication patterns between partitions: for example, by reducing the amount of *hasVehicle* messages by using an Observer pattern.
- Further optimization could be reached by the usage of dynamic partitioning: this would be a far more complex endeavor, requiring alteration of SUMO to allow for dynamic modification of the road network at runtime, and an efficient runtime logic to avoid excessive overhead in changing the road network.
- The partitioning process also takes up a considerable amount of time currently. While much of this is during the execution of SUMO binaries executed during partitioning, which are out of our control, and the partitioning operations can generally be considered an ”offline” process whose optimization is less relevant, some avenues for improvement are possible: for example, switching to more optimized languages other than Python.

Appendix A

Appendix - ParallelTwin Usage

In this appendix, the usage and parameters of the developed software will be detailed.

ParallelTwin [48] must be launched through the *launch.sh* script, which automatically locally setups the required Python environment. Alternatively the Python environment can be manually activated and then the program can be manually launched via the *ParallelTwin* executable.

The program uses a *data* subfolder to store partitioning and runtime data in, and an *output* folder to place outputs, including ones specified by the user.

Parameters

The following are the required and optional parameters available for ParallelTwin.

Required Parameters			
-c	**--cfg**	string	SUMO simulation config path

Optional Parameters			
-h	**--help**	-	Prints help message and exits
-v	**--version**	-	Prints version information and exits
-N	**--num-threads**	int	Number of partitions to run the program with, defaults to 4.
-	**--part-threads**	int	Threads used while partitioning (will be capped to partition amount), defaults to 8.

-	**--remote-port**	N (int)	First remote port that will be used to host TraCI servers in the partitions. Each partition will host it at N + *partition number*, where partition number starts from 0. If not set, no TraCI server will be started. Note that doing this will made each partition wait for a client to take control and manually allow them to proceed.
-	**--gui**	-	Displays SUMO gui. Note that this launches one GUI application per thread.
-	**--skip-part**	-	Skip partitioning. If partitioning data wasn't generated in previous runs of this program, it will error.
-	**--keep-poly**	-	Keep poly data if present in the original sumocfg, which is normally used to display additional shapes in the GUI. Disabled by default for performance.
-	**--pin-to-cpu**	-	Force each partition to run on one CPU only; will not work with higher values than the amount of available CPUs.
-	**--log-handled-vehicles**	-	Print a text file in the data folder with the number of handles vehicles at each simulation step for each partition.
-	**--log-msg-num**	-	Print a text file with the number of messages passed in each iteration in each partition.
-	**--data-dir**	string	Data directory to store working files in, defaults to "data".
-	**--verbose**	-	Print additional output.

Outputs

ParallelTwin places all of its outputs in the *output* folder. By default it only outputs a log file for each partition in the output folder (as per SUMO's `--log` option), as *partN_log.txt*, and in the current version also two CSV files containing simulation times in each partition, and communication times in each partition respectively (this may be changed to a parameter in future developments).

Additional outputs can be specified using standard SUMO parameters, either in the simulation *sumocfg* file or in the command line. ParallelTwin will add SUMO's `--output-prefix`[1] with the value *partN_*, to differentiate between outputs of different partitions, and will automatically adjust outputs defined in the sumocfg file to be placed into the output folder instead. Absolute paths in outputs are not supported.

For example, a sumocfg containing this entry:

```
<output>
    <emission-output value="results/emission_output.xml"/>
</output>
```

[1]See https://sumo.dlr.de/docs/Simulation/Output/index.html#separating_outputs_of_repeated_runs

```
</output>
```

will create N files named *partK_emission_output.xml* in the *output/results* folder.

Partitioning Configuration

As mentioned, ParallelTwin launches the *createParts.py* script to run the partitioning of the road network. It can also be run standalone, by either using the installed Python environment or directly using *run-with-env.sh* (recommended).

```
./run-with-env.sh python scripts/createParts.py [...]
```

The execution of *createParts.py* by ParallelTwin can be configured by using the keyword `--`, for example:

```
./launch.sh -c assets/simpleNet.net.xml -N 4 -- --png
```

will pass the `--png` argument to *createParts.py*. The arguments are the following:

Required Parameters			
-N	**–num-parts**	int	Number of partitions to create
-c	**–cfg-file**	string	Path to the SUMO simulation sumocfg file

Optional Parameters			
-h	**–help**	-	Show an help message and exit
-	**–data-folder**	string	Folder to store output in (will be used by ParallelTwin). Defaults to `data`.
-	**–keep-poly**	-	Keep poly files from the sumocfg, disabled by default for performance
-	**–no-metis**	-	Partition network using grid (carried over from ParallelSumo, currently untested and unsupported)
-w	**–weight-fun**	[route-num,osm ...]	One or more edge weighting methods to use, use with no values to avoid using any weight. Defaults to `route-num`, can be repeated to combine more weights (will be summed).

-W	**–node-weight**	[connections,connexp ...]	One or more node weighting methods to use for nodes, use with no values to avoid using any weight. Defaults to `connections`, can be repeated to combine more weights (will be summed).
-T	**–threads**	int	Threads to use for processing the partitioning of the network, will be capped to partition number, defaults to 8 or num-parts.
-	**–use-cut-routes**	-	Use cutRoutes.py with postprocessing instead of our custom script to cut the routes (likely slower, but might handle different edge cases)
-t	**–timing**	-	Measure the timing for the whole process and output it
-	**–png**	-	Output network images for each partition; can be slow
-	**–quick-png**	-	Remove some image details to output network images faster, needs –quick-png to also be set
-	**–force**	-	Regenerate partitioning even if data folder already contains partition data matching these settings
-	**–filter-vehs**	string[,string,string...]	Only keep vehicles with these ids in simulation. Meant for testing
-v	**–verbose**	-	Additional output

Bibliography

[1] "A taxonomy of digital twins," 08 2020.

[2] Metis on github. [Online]. Available: https://github.com/KarypisLab/METIS/tree/master

[3] (2023) Packaging waste statistics. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Packaging_waste_statistics

[4] (2023) Carbon pollution from transportation. [Online]. Available: https://www.epa.gov/transportation-air-pollution-and-climate-change/carbon-pollution-transportation

[5] C. Reche, A. Tobias, and M. Viana, "Vehicular traffic in urban areas: Health burden and influence of sustainable urban planning and mobility," *Atmosphere*, vol. 13, no. 4, 2022. [Online]. Available: https://www.mdpi.com/2073-4433/13/4/598

[6] O. Andrisano, I. Bartolini, P. Bellavista, A. Boeri, L. Bononi, A. Borghetti, A. Brath, G. E. Corazza, A. Corradi, S. de Miranda, F. Fava, L. Foschini, G. Leoni, D. Longo, M. Milano, F. Napolitano, C. A. Nucci, G. Pasolini, M. Patella, T. Salmon Cinotti, D. Tarchi, F. Ubertini, and D. Vigo, "The need of multidisciplinary approaches and engineering tools for the development and implementation of the smart city paradigm," *Proceedings of the IEEE*, vol. 106, no. 4, pp. 738–760, 2018.

[7] M. Enders and N. Hoßbach, "Dimensions of digital twin applications - a literature review," 08 2019.

[8] Digital twin market size, share & trends analysis report by solution (component, process), by deployment (cloud, on-premise), by enterprise size, by application, by end-use, by region, and segment forecasts, 2023 - 2030. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/digital-twin-market

[9] Eu destination earth. [Online]. Available: https://digital-strategy.ec.europa.eu/en/policies/destination-earth

[10] B. S. Center. (2023, 3) Barcelona tests with a digital twin developed by bsc if it is a 15-minute city. Accessed on 11/2023. [Online]. Available: https://www.bsc.es/news/bsc-news/barcelona-tests-digital-twin-developed-bsc-if-it-15-minute-city

[11] I. G. Andy Walker. (2023, 5) Singapore's digital twin - from science fiction to hi-tech reality. Accessed on 11/2023. [Online]. Available: https://infra.global/singapores-digital-twin-from-science-fiction-to-hi-tech-reality/

[12] A. B. Aarni Heiskanen. (2019, 4) Singapore's digital twin - from science fiction to hi-tech reality. Accessed on 11/2023. [Online]. Available: https://aec-business.com/helsinki-is-building-a-digital-twin-of-the-city/

[13] Change2twin website. [Online]. Available: https://www.change2twin.eu

[14] M. Grieves and J. Vickers, *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*, 08 2017, pp. 85–113.

[15] A. Fuller, Z. Fan, C. Day, and C. Barlow, "Digital twin: Enabling technologies, challenges and open research," *IEEE Access*, vol. 8, pp. 108 952–108 971, 2020.

[16] B. Korth, C. Schwede, and M. Zajac, "Simulation-ready digital twin for realtime management of logistics systems," 12 2018, pp. 4194–4201.

[17] McKinsey. What is digital twin technology? [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-digital-twin-technology

[18] Precog. The types of digital twins used in process manufacturing. [Online]. Available: https://precog.co/blog/types-of-digital-twins/

[19] IBM. What is a digital twin? [Online]. Available: https://www.ibm.com/topics/what-is-a-digital-twin#Types+of+digital+twins

[20] "Digital twin enabling technology catalogue," Change2Twin, 05 2021, see https://www.change2twin.eu/about/deliverables/ for related materials. [Online]. Available: https://www.change2twin.eu/wp-content/uploads/2021/11/D1.3-Digital-Twin-Enabling-Technology-Catalogue.pdf

[21] Openstreetmap website. [Online]. Available: https://www.openstreetmap.org

[22] Openstreetmap foundation website. [Online]. Available: https://osmfoundation.org

[23] Openstreetmap wiki - beginner's guide. [Online]. Available: https://wiki.openstreetmap.org/wiki/Beginners_Guide_1.3

[24] B. Ciepłuch, R. Jacob, P. Mooney, and A. C. Winstanley, "Comparison of the accuracy of openstreetmap for ireland with google maps and bing maps," *Proceedings of the Ninth International Symposium on Spatial Accuracy Assessment in Natural Resuorces and Enviromental Sciences 20-23rd July 2010*, p. 337, July 2010. [Online]. Available: https://mural.maynoothuniversity.ie/2476/

[25] G. Salvucci and L. Salvati, "Official statistics, building censuses, and openstreetmap completeness in italy," *ISPRS International Journal of Geo-Information*, vol. 11, no. 1, 2022. [Online]. Available: https://www.mdpi.com/2220-9964/11/1/29

[26] Osm taginfo. [Online]. Available: https://taginfo.openstreetmap.org

[27] Users - osm stats. Accessed on 28/11/2023. [Online]. Available: https://osmstats.neis-one.org/?item=members

[28] Contributors on osm wiki. Accessed on 28/11/2023. [Online]. Available: https://wiki.openstreetmap.org/wiki/Contributors

[29] C. Barrington-Leigh and A. Millard-Ball, “The world’s user-generated road map is more than 80% complete,” *PLoS ONE*, vol. 12, no. 8, p. e0180698, 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0180698

[30] Osm apps webtool. Accessed on 28/11/2023. [Online]. Available: https://osm-apps.zottelig.ch/#

[31] Uber data attribution. Accessed on 28/11/2023. [Online]. Available: https://www.uber.com/legal/it/document/?country=united-states&lang=en&name=data-provider-attribution

[32] “How lyft discovered openstreetmap is the freshest map for rideshare.” [Online]. Available: https://eng.lyft.com/how-lyft-discovered-openstreetmap-is-the-freshest-map-for-rideshare-a7a41bf92ec

[33] Snapchat map. Accessed on 28/11/2023. [Online]. Available: https://map.snapchat.com/about

[34] “Pokémon go’s maps now look a lot different.” [Online]. Available: https://www.polygon.com/2017/12/4/16725748/pokemon-go-map-changes-openstreetmap

[35] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, “Microscopic traffic simulation using sumo,” in *The 21st IEEE International Conference on Intelligent Transportation Systems.* IEEE, 2018. [Online]. Available: https://elib.dlr.de/124092/

[36] Y. Wang, H. Yin, H. Chen, T. Wo, J. Xu, and K. Zheng, “Origin-destination matrix prediction via graph convolution: A new perspective of passenger demand modeling,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1227–1235. [Online]. Available: https://doi.org/10.1145/3292500.3330877

[37] H. Yang and J. Zhou, “Optimal traffic counting locations for origin–destination matrix estimation,” *Transportation Research Part B: Methodological*, vol. 32, no. 2, pp. 109–126, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0191261597000167

[38] Public origin and destination tables - uk office for national statistics. [Online]. Available: https://www.ons.gov.uk/census/2011census/2011censusdata/originanddestinationdata/publicoriginanddestinationtables

[39] Rilevazione flusso veicoli tramite spire - anno 2023. [Online]. Available: https://opendata.comune.bologna.it/explore/dataset/rilevazione-flusso-veicoli-tramite-spire-anno-2023

[40] Comptage routier - données trafic issues des capteurs permanents. Accessed on 29/11/2023. [Online]. Available: https://opendata.paris.fr/explore/dataset/comptages-routiers-permanents/

table/?disjunctive.libelle&disjunctive.etat_trafic&disjunctive.libelle_nd_amont&disjunctive.libelle_nd_aval&sort=t_1h

[41] Output - sumo documentation. [Online]. Available: https://sumo.dlr.de/docs/Simulation/Output/index.html

[42] Visualization - sumo documentation. [Online]. Available: https://sumo.dlr.de/docs/Tools/Visualization.html

[43] Traci - sumo documentation. [Online]. Available: https://sumo.dlr.de/docs/TraCI.html

[44] Robopac - change2twin project. [Online]. Available: https://www.change2twin.eu/success-stories/pilot-experiments-in-change2twin/robopac/

[45] Digital twin - robopac. [Online]. Available: https://www.robopac.com/it/news/digital-twin

[46] Route2vel github. [Online]. Available: https://github.com/filloax/route2vel

[47] J. Xia, N. Yokoya, B. Adriano, and C. Broni-Bediako, "Openearthmap: A benchmark dataset for global high-resolution land cover mapping," 2022.

[48] Parallelsumo github. [Online]. Available: https://github.com/filloax/Parallel-Sumo

[49] P. Tay. Parallel-sumo. [Online]. Available: https://github.com/philliptay/Parallel-Sumo

[50] Zeromq website. [Online]. Available: https://zeromq.org/

[51] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[52] ——, "Multilevelk-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731597914040

[53] Sumo documentation. [Online]. Available: https://sumo.dlr.de/docs/index.html