SUMO User Conference 2022
https://doi.org/10.52825/scp.v3i.109
© Authors. This work is licensed under a Creative Commons Attribution 3.0 DE License
Published: 29 Sept. 2022

Building a real-world traffic micro-simulation scenario from scratch with SUMO

Maria Laura Clemente[1][https://orcid.org/0000-0002-5952-2810]
[1] CRS4, Italy
clem@crs4.it

Abstract

Simulation of Urban Mobility (SUMO) is a powerful traffic simulation program which can work at different scales, from sub-microscopic to macroscopic. Depending on the available input dataset, it is possible to build lots of different configurations, changing routing and car-follow algorithms, and many parameters. Building a basic SUMO scenario is a multi-step activity involving the followings: preparing the transportation network, traffic definition, setting a routing algorithm, and running the simulation. The aim of the present work is to show a detailed real case study explaining how to build a complete scenario and run simulations, starting from the preparation of the network from Open Street Map. The last part of the present paper is about how to use the SUMO output files with MongoDB in order to keep track of significant information resulting from each simulation.

1 Introduction

Transport planning is based on three pillars: demand analysis (the number of movements performed by people, either walking or using a vehicle), offer analysis (the transportation network and public transport services), and the traffic assignment with the interaction models to get a configuration of equilibrium (or a sequence of equilibrium, in the dynamic models) between the demand and the offer. In this big picture, it's understandable that a reliable demand computation is the basis for a good simulation; for this purpose, two approaches have commonly been used: the more traditional one is the trip-based, also called *four-step model*, which puts the trips of each vehicle at the core; the second one, more recent, is called *activity-based* because it starts from the analysis of activities performed by the users to understand how they will move during the day (McNally, 2000; McNally et al. 2007). Hybrid approaches are possible as well.

Many platforms for traffic simulation are available and they can be grouped by main features: commercial versus free and open-source, scale of simulation (if performing macro meso or micro simulation), routing algorithms (if trip based or activity based), if altimetry is supported, possibility of online interaction, documentation availability, etc.

An updated and comprehensive comparison of all the software available for traffic simulation is provided in (Ullah et al., 2021). The most popular commercial tools for macro-simulation are Cube Voyager, Visum and OmniTRANS, while VISSIM is very common for the micro-simulations. Talking about free and open-source solutions for micro-simulation, Matsim (https://www.matsim.org/) and SUMO (https://sumo.dlr.de) are the most widely used.

The present activity was carried out with SUMO (Alvarez Lopez et al., 2018) for many reasons. First, it is free and open source (under the Eclipse Public License EPL v.2), is highly portable, and is continuously improved by the German Aerospace Center and a huge community of users. Another important feature is that it allows to manage a big area micro-simulation, in fact it has been used for whole cities (Maiorov et al., 2019; Bachechi et al., 2019): Bologna, Brunswick, Dublin, Ingolstadt, Luxembourg, Monaco, Stuttgart, Turin, Cologne (https://sumo.dlr.de/docs/Data/Scenarios.html).

A simulation built with SUMO is continuous in space (vehicles can be in any position on the street) and discrete over time (the mobility model uses uniform time steps), supports multi-modality, is highly configurable, and extendible (it's also possible to add new algorithms). Depending on the traffic dataset, SUMO allows to build different types of micro-simulations, from the classic Origin Destination Matrices to data from sensors (Po et al. 2019).

SUMO provides command line tools specific to each of the steps required to build a whole simulation. A first confusion can arise from the name, because sumo is the name of the whole platform but also the name of the last step, which is the one to run the simulation; there is also the command sumo-gui which differs in that it also shows the graphical visualization of the running simulation. The sumo platform has a solid and efficient base of algorithms written in C++, but it includes also a great number of useful Python scripts to facilitate many operations, working as a sort of wrapper over the C++ core.

All the files required in input to a simulation are in xml format. So, all the procedures related to the input file preparation have the target of translating the available information into a set of input xml files suitable to be understood by the SUMO world.

The present paper describes one of the many possible ways to build a sumo micro-simulation, referring to a real scenario, the corridor SS195 to Cagliari (Italy), for which some Origin Destination Matrices were available; the available input dataset drove the choice toward a classical 4-step model. So, this paper shall not be considered as a complete tutorial of this powerful program, for which the official website is the best reference (http://sumo.dlr.de/userdoc/), but it shows a detailed roadmap of the steps required to get a real-world simulation up and running, enriched with practical tips to overcome tricky obstacles along the way, learned by experience.

Having said that, the following set of files makes up what is usually called a 'scenario':

- Information about the transportation network (in short, net),
- Information about the Traffic Assignment Zones (TAZ),
- Information about the Origin and Destination Matrices (ODM).

These compound the smallest set of input to run a real-world simulation, which could be extended with many further information about: buildings, types of vehicles, traffic lights, Public Transport Services and bus stops, and so on.

A SUMO simulation can have many different configurations, based on a great number of parameters, some of which are particularly important, such as:

- *route-choice-method (*input of the dynamic User Assignment for route building, default is Gawron*),* further information at https://sumo.dlr.de/docs/Demand/Dynamic_User_Assignment.html
- *routing-algorithm*, the famous Dijkstra is set by default, but other possibilities are alpha star (astar), Contraction Hierarchies (CH), and CHWrapper, as explained at https://sumo.dlr.de/docs/Simulation/Routing.html

- car-following-models (input of sumo/sumo-gui), explained in detail at https://sumo.dlr.de/docs/Car-Following-Models.html

Providing details about the choice of these and the many other parameters is out of the scope of the present paper; it's important to know that they are the core of a traffic micro-simulation and each of them involves further parameters that must be tuned during a careful process of model calibration (Krajzewicz et al., 2002; Aminia et al., 2019), when a sort of ground truth dataset is available. The official SUMO documentation gives many explanations about them. Being an open source, an interesting and evolving research activity consists in further analysis of custom-developed algorithms. In this presentation we limit to provide an example of sequence of command lines applied to the real-world case study. To help beginners, SUMO provides default values for all these parameters.

This paper doesn't cover the interactive "Traffic Control Interface" (TraCI), provided by SUMO software to allow accessing to a running traffic simulation in order to retrieve values of simulated objects and change their behavior "on-line".

The last part of the present paper proposes a way to model the output dataset of a simulation using the Mongo Atlas cloud database in order to allow a quick and effective comparison of the results coming from all the simulations related to the same scenario.

2 Network preparation

A traffic micro-simulation starts with the choice of the boundaries of the study area. Working on a road, like the SS195, which collects traffic flows from a big area and brings it to Cagliari, as the most attractive point of that area, cannot be limited only to the road, but needs a careful analysis of the whole area involved; in other words, here it is particularly important to analyze a reasonable extension of the 'catchment area'. On the other hand, it is recommended to avoid exaggerating in this operation because the bigger is the area, the more time consuming will result, as side effect, working on it.

2.1 Downloading the map from Open Street Map

Once the proper area has been carefully chosen, the next step consists in downloading it from Open Street Map (openstreetmap.org), in short OSM.

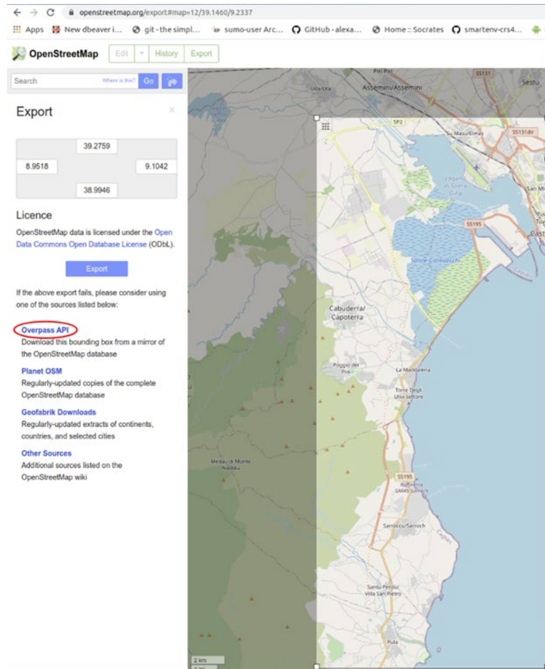

Figure 1: Downloading the map from OSM

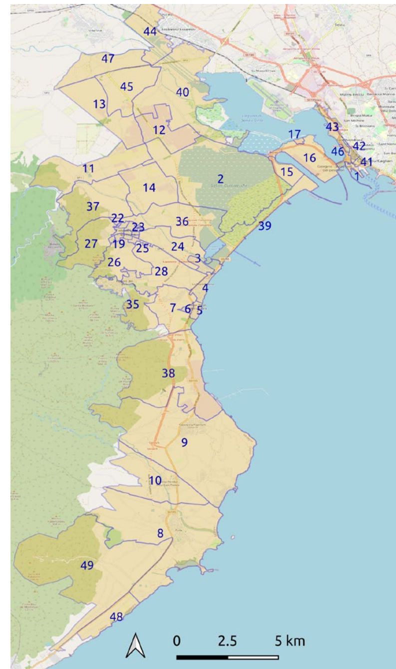

Figure 2: The Traffic Assignment Zones

Figure 1 shows the area required for the SS195 road. But when the area is as big as this one, OpenStreetMap API answers to the request with the following error message*: "You requested too many nodes (limit is 50000). Either request a smaller area or use planet.osm"*. A work around to this problem is selecting the link 'overpass API' (shown with a red contour in Figure 1) which allows to download the selected bounding box from a mirror of the OpenStreetMap database and in fact the downloading will immediately start.

3 Adding Altimetry to the OSM map with osmosis

Unfortunately, OSM does not carry elevation data. There are some ways to add them (https://sumo.dlr.de/docs/Networks/Elevation.html); for the case study, altitudes have been downloaded from Shuttle Radar Topography Mission (SRTM); in particular, for Cagliari the altimetry data are in the file N39E009.hgt.zip which includes points from N39 to N40 and from E009 to E010.

Once this file is available, *osmosis* must be installed (*sudo apt-get install osmosis*). First of all, it's better to check the validation of the OSM map file in *osmosis*, by reading it (the flag *–write-null* avoids any output building):

```
osmosis --read-xml SS195_map.osm.xml --write-null
```

Then if everything goes well, it's possible to run *osmosis* in order to have the altitudes in the osm map file, giving the local directory of the SRTM file, and specifying to set the tag element *ele* for the altitude, which is the trick to have the file imported in SUMO (through the command *netconvert*, as explained later):

./osmosis --read-xml SS195_map.osm.xml --write-srtm locDir=~/SRTM/ tagName=ele --write-xml SS195_map_ele.osm.xml

With this command, the output file *SS195_map.osm.xml* will have each node element enriched with the extra tag about elevation, as shown in Figure 3.

```
<node id="31483414" version="1" timestamp="2019-07-12T09:51:28Z" changeset="1"     lat="39.2842393"
lon="9.0286643">
    <tag k="ele" v="6.808519999996889"/>
</node>
```

Figure 3: The detail of a node in the OSM file with the elevation extra tag.

4 SUMO

Once the OSM map is ready, it's time to start using SUMO and its command lines (here after, cml) and Python scripts. In fact, as already explained, SUMO is made of tools designed to be used independently from each other. SUMO installation is explained from the official site web https://sumo.dlr.de/docs/Installing/index.html.

The presented simulation has been carried out with sumo installed from source code (https://github.com/eclipse/sumo) on ubuntu version 20.04, in order to be able to keep it frequently updated.

To have SUMO command lines available, it's useful to set up the environment variable SUMO_HOME in the .bash file:

export SUMO_HOME=/mypathtosumo/Sources/sumo_sourcecode/

Each command could also take information (input files and parameters) from a xml configuration file, which maybe would reduce error-proneness; on the other hand, for the case study, all the values for each parameter were defined inside the same shell script and all the command lines put in sequence inside nested loops in order to run all the possible combinations of parameters. This structure allows to run many configurations one after the other and to manage all the parameters involved in all the different configurations from just one file. Each command line used will be described in the following paragraphs.

4.1 Netconvert: converting the OSM map into the xml format input for SUMO

The OSM map file, an xml format, needs to be translated into the SUMO xml format (they are compliant to two different xml schemas). To do this, the SUMO command line required is *netconvert*.

An important flag to add when performing this transformation is *–no-turnarounds*. Without this flag, the vehicles moving in the net, arriving at the end of a road, at the boundary of the map, will make a U conversion, going back along the lane in the opposite direction. The *netconvert* command will be:

netconvert -v –osm.elevation true --osm-files SS195_map_ele.osm --no-turnarounds -o SS195_osm_ele.net.xml

The network file obtained as output running this command is called *SS195_osm_ele.net.xml* and can be used as network for the rest of operations with SUMO. Optionally, in order to get an improved model for the visibility of the roundabouts, two further steps can be done. The first one splits the parts of the network into different files:

netconvert -s SS195_osm_ele.net.xml –plain-output-prefix SS195

This command produces in output: *SS195.edg.xml* (file of only edges), *SS195.nod.xml* (files of only nodes), *SS195.con.xml* (file of only connections), *SS195.tll.xml* (file of the traffic lights), *SS195.typ.xml* (file of the types). Now, calling again the *netconvert* command, giving as input all these files, produces a better network file in output (as anticipated):

netconvert -e SS195.edg.xml -n SS195.nod.xml -x SS195.con.xml -i SS195.tll.xml -t SS195.typ.xml -o SS195.net.xml

The output *SS195.net.xml* is the definitive network file which will be used from now on.

In order to add some environment elements to the bare map, there is another command line provided by SUMO, which is *polyconvert*. It requires in input three files: the OSM map file *SS195_map.osm* (as downloaded from OSM), the output file of *netconvert*, which is *SS195_osm.net.xml*, and also a file that is available in the SUMO code, *osmPolyconvert.typ.xml*, that is necessary to define the types of environments. The complete command line will be:

polyconvert --net-file SS195.net.xml --osm-files SS195_map.osm --type-file $SUMO_HOMEdata/typemap/osmPolyconvert.typ.xml -o SS195.poly.xml

It must be said that there is also a quicker way to perform a starting scenario, using the SUMO tool *osmWebWizard.py* which allows to select the study area. In this case *sumo-gui* will import the net from OSM to run a simulation with randomly generated traffic; this approach can be very useful to quickly build all the required set of files, that can be corrected and integrated afterward.

In any case, the map generated must be carefully checked and corrected through another important tool of SUMO, called *netedit*. Typical mistakes are extra lanes/crossings, missing accelerating/decelerating lanes, etc.; *netedit* allows to correct all of them, by hand and one by one.

Another useful tool to make things easier is *sumopy*, a free open-source python library (https://github.com/schwoz/sumopy) gathering all the steps required to build a sumo simulation in a very comprehensive Graphic User Interface (Schweizer, 2013). *Sumopy* was used to build the real-world scenario of the city of Bologna (Schweizer et al., 2021).

4.2 Polyconvert: building the Traffic Assignment Zones (TAZs)

When input traffic data about the demand is available, they define an Origin/Destination matrix, in terms of number of vehicles moving from each TAZ to the others; to add this information to the scenario, a shapefile of the zones must be defined through a GIS editor, like QGIS (https://www.qgis.org). A good starting point is the official shapefile of the census regions, from where it is possible to group regions into sensible TAZs. Doing this operation, it’s possible that a geographical conversion is required to have all the map layers properly matching in QGIS. Typical systems are *EPSG 3003, 4326,* and *32632*. When a shapefile doesn’t show this information, the web service *epsg.io* comes in help, revealing the current coordinates system. Knowing the exact coordinates system allows to make the proper transformations with QGIS, ending with all the layers matching.

The 49 Traffic Assignment Zones of the real case study are shown in Figure 2. The shapefile of the TAZs built with QGIS must be translated into the format required by SUMO, using also the network

file *SS195_osm.net.xml* as input. The conversion of this file to the proper format requires to call again the *polyconvert* cml, this time using other arguments, and specifying the name used for the TAZ column:

polyconvert -v --shapefile-prefixes TAZs --shapefile.guess-projection true --shapefile.id-column TAZ -n SS195.net.xml -o TAZs.taz.xml

It's important to note, that in order to work properly, each TAZ zone in the XML file, in addition to the boundary definition, must also have at least one edge of the network declared as TAZ source, and another as TAZ sink. As suggested by the name, the TAZ sources are all the edges from where the trip can start, while the TAZ sinks are all the edges in the Zone where the trips can end. This information should be added carefully by hand, directly in the SUMO xml file, which is the output of the *polyconvert* command line. In fact, although there is the python script *edgesInDistricts.py* (one of the available SUMO tools) performing this operation automatically, it adds all the edges in the TAZ as sources and as origins, with no distinctions. For the case study, a textual file with a list of the tazSources and tazSinks for each TAZ were prepared to be added running a custom Python script. An example of the structure of a TAZ xml file is shown in Figure 4.

```
<tazs>
  <taz id = “48” shape=”2995.4312,1572.2758, …” >
    …
    <tazSource id=”2940” weight=”0.07”/>
    <tazSource id=”2946” weight=”0.07”/>
     …
    <tazSink id=”2940” weight=”0.07”/>
    <tazSink id=”2946” weight=”0.07”/>
     …
  </taz>
  …
</tazs>
```

Figure 4: The TAZ file XML format

4.3 Od2trips: importing the traffic demand from OD matrices into individual trips

Figure 5 shows a very basic vehicle definition used for the case study, considering only cars (in SUMO called '*passenger*') and *trucks* (generic name for heavy vehicles). Lots of more details could be added to specify a long list of vehicles and related features, even pedestrians.

```
<additional>
  <vType id=“passenger” vClass=“passenger” length=”4.60” color=”1,1,1”
deepartSpeed=”max” />
  <vType id=“truck” vClass=“truck” length=”10.22” color=”0,1,1” deepartSpeed=”max” />
</additional>
```

Figure 5: A basic XML file for vehicle definition

The Sumo *od2trips* cml imports the traffic demand, as a list of origin, destination, and number of vehicles, into an xml output file made of trips; each trip is defined by an *id* with starting and ending time (included inside the given time-lapse); this command must be run once for each transport mode and for each time-lapse and produces as output an xml file for the input transport mode and time-

lapse. The TAZs file is also required as input for this operation. The following is the command line used in the case study to obtain the xml file of the trips of cars inside the 7-8 a.m. time-lapse:

```
od2trips -v --taz-files TAZs.taz.xml --vtype passenger --prefix car --od-matrix-files SS195_ODM_7_8_cars.od
-o $OUTPUT_DIR/SS195_ODM_7_8_cars.odtrips.xml
```

An OD input file in the *O-format*, is an example of suitable input for the *od2trip* command line:

- any line starting with an asterisk is a comment which is ignored from *od2trip*,
- the first line is an identification code of the format of the file ($OR;D2),
- the second significant line defines the time-lapse,
- the third line is a scaling factor for the number of vehicles,
- from the fourth line to the end of the file, the format is:
    - id of the Origin TAZ,
    - id of the Destination TAZ,
    - number of vehicles moving during the specific time-lapse from the Origin to the Destination TAZ.

A sample of the output of the *od2trip* cml is shown in Figure 6.

```
<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/routes_file.xsd">
        <trip id="61" depart="28800.79" from="0" to="118" fromTaz="1" toTaz="44" departLane="free"
departSpeed="max"/>
        <trip id="7048" depart="28801.27" from="4489" to="118" fromTaz="7" toTaz="44" departLane="free"
departSpeed="max"/>
        <trip id="2917" depart="28801.88" from="56" to="19" fromTaz="40" toTaz="13" departLane="free"
departSpeed="max"/>
        <trip id="268" depart="28802.83" from="3017" to="2879" fromTaz="10" toTaz="8" departLane="free"
departSpeed="max"/>
```

Figure 6: A sample of the Origin-Destination trips file

4.4 Duarouter: from trips to routes

The files output of *od2trips*, one for each transport mode, are the input of the SUMO cml *duarouter*, which for each vehicle builds the complete route, in terms of sequence of edges along with the starting time; this file is the input of the simulation with the *sumo* command (or *sumo-gui*).
Calling *duarouter* an important input parameter is the *route-choice-method*, which can be *Gawron* (DUA, used by default), *logit*, or *lhose*.
An example of the duarouter command line used for the case study is the following:

```
duarouter -v -n SS195.net.xml --route-files cars.odtrips.xml, SS195_trucks.odtrips.xml
--additional-files vtypes.xml --xml-validation never --no-step-log true -o SS195_duarouter.odtrips.rou.xml
```

A sample of the output of *duarouter* is shown in Figure 7. As explained in 4.6, the duarouter cml can be used as first step of the iterative algorithm to approximate the Dynamic User Equilibrium (DUE).

```
<routes xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:noNamespaceSchemaLocation=http://sumo.dlr.de/xsd/routes_file.xsd>
        <vType id=”passenger” length=4.60” minGap=”2.00” maxSpeed=”30.00” speedFactor=”normc(1.00,0.00)”
vClass=”passenger” color=”white” carFollowModel=”IDM” accel=”1.0” decel=”1” tau=”1.4” />
        <vehicle id=”car8370” type=”passenger” depart=”25200.42” departLane=”free” departSpeed=”max”
fromTaz=”47” toTaz=”38”>
```

```
        <route edges="47874#0 198047874#1 198047874#2 198047906#0 198047906#1 -373339455“ />
    </vehicle>
    <vehicle id="car61" type="passenger" depart="25200.79" departLane="free" departSpeed="max"
fromTaz="1" toTaz="43">
        <route edges="27752726#0 27752726#1 116050065 11605029 27752728 666975680 11361107" />
    </vehicle>
    <vehicle id="car7069" type="passenger" depart="25201.27" departLane="free" departSpeed="max"
fromTaz="46" toTaz="41">
        <route edges="138413006 26617614#0 26617614#1 26617616 26617617 113611070#0" />
    </vehicle>
```

Figure 7: A sample of the *duarouter* output

4.5 Sumo cml: running the micro-simulation

The command line *sumo* runs the simulation without the graphical user interface. To better understand this core part, it's useful to know that *sumo* performs a time-discrete simulation, where the default step (parameter *step-length*) is 1 second but can be reduced up to 1µs; the simulation model is space-continuous and the position of each vehicle at each simulation step is defined by the *id* of the lane and the distance from the beginning of that lane. By default, the speed of each vehicle is computed using an extension of the stochastic car-following model developed by Stefan Krauss (Krauß, 1998), which is faster and simpler than others. SUMO provides also several other algorithms for this task: Wiedemann 74 and 99 (the ones used by VISSIM), IDM (very popular), but particularly important is the *KraussPS* version because it uses road slopes in the computation, information obtained from the elevation value of the nodes in the network file.
One of the parameters for *sumo* is the *routing-algorithm*, which can be Dijkstra by default (Dijkstra, 1959), alpha star (in short, astar), CH, CHWrapper.
When a vehicle is stopped in the traffic jam for too much time (which is another configurable parameter), it is automatically moved from its position to another one. It's possible to avoid this, by setting the *time-to-teleport* parameter to -1. The simulation allows to set a particular action to be taken in case of collisions, for instance, send a warning in standard output. The list of all additional files, such as information about the buildings, can be passed to the simulation through the *additional-files* flag.
Another important input is related to the time-lapse, for instance 7-8 A.M., because it must be converted into seconds of simulation:

- *start time 7 A.M. = 3600 s * 7 = 25200 s*
- *end time 8 A.M. = 25200 s + 3600 s = 28800 s*

To highlight the relevance of this aspect, it must be said that if start and end time of the simulation are not set accordingly to the ones specified in the Origin/Destination file, the simulation will not have any vehicle running in the network.
An example of the sumo command line is the following:

```
sumo -v -n SS195.net.xml --route-files SS195.odtrips.rou.xml --duration-log.statistics true -b 25200 -e 28800
--time-to-teleport -1 --collision.action warn  --step-length 0.5 --routing-algorithm dijkstra --time-to-impatience -1
--additional-files SS195.poly.xml
```

4.6 duaIterate.py: the iterative algorithm to approximate the Dynamic User Equilibrium (DUE)

After running *duarouter* the first time, its output (SS195_duarouter.odtrips.rou.xml, related to all the transportation modes, which in this example are cars and trucks) can be given as input to a python script, *duaIterate.py*, which performs Gawron's algorithm (Gawron, 1998) by calling both commands, *duarouter* and *sumo* (already explained in previous paragraphs), iteratively in order to approximate the Dynamic User Equilibrium (DUE), where the number of iterations depends on the scenario. A scheme of this process is shown in Figure 8.

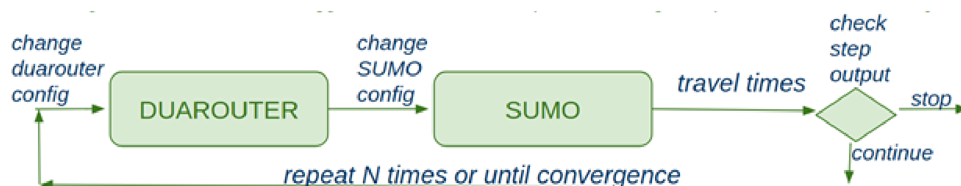

Figure 8: A scheme of how duaIterate.py works.

An important thing to know is that when *duaIterate.py* is run, the input parameters for the *sumo* command are distinguished from the ones for *duarouter* by the prefix *sumo–*.
A key parameter for *duaIterate.py* is the *max-convergence-deviation*: at each iteration the standard deviation related to the average travel time is computed allowing to use convergence as threshold to stop the iterations.
An example of this command for the case study is reported here after:

```
$SUMO_HOME/tools/assign/duaIterate.py --router-verbose -n SS195.net.xml -D TAZs.taz.xml -r SS195.odtrips.rou.xml -l 50 -b 25200 -e 28800 --max-convergence-deviation 0.01 sumo--step-length 0.5 sumo--routing-algorithm dijkstra sumo--vehroute-output vehroute.xml
```

4.7 Running a simulation with sumo-gui

The command sumo-gui, as already mentioned, provides a graphical user interface (GUI) which allows to watch the vehicles moving in the network during a simulation. In the GUI it's possible to set the colours of vehicles and lanes according to various criteria. Effective choices are:

- to be able to watch the simulation, set the delay time to 100, otherwise it runs too fast
- to have an idea of the jammed edges, the colour of vehicles can be set according to speed (red when stopped, etc.)
- to set a constant size for the vehicles, which otherwise are too small.

Figure 9 shows how to achieve this configuration in the sumo GUI.

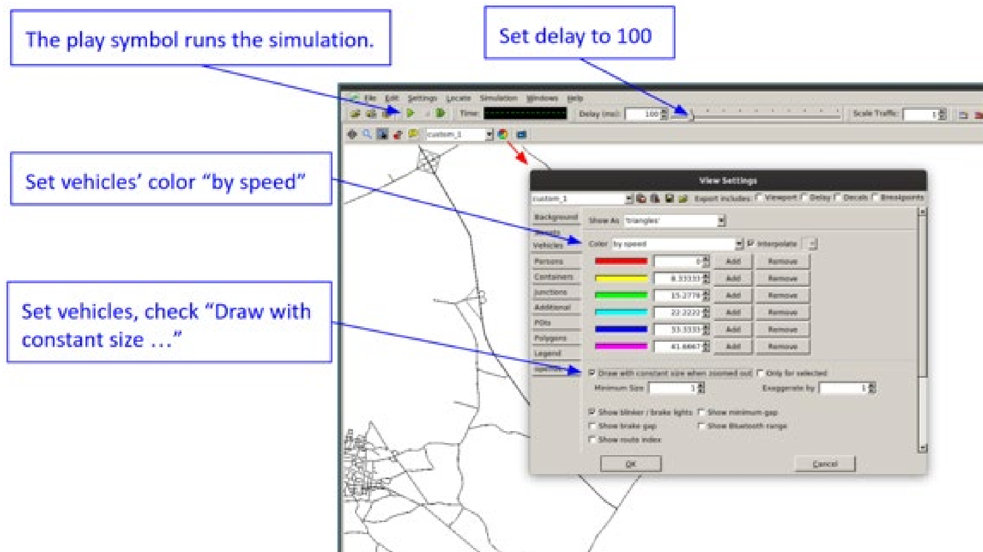

Figure 9: How to get a more readable simulation

During the simulation, while the vehicles are moving, many useful operations are possible. One of these consists in closing an edge, or only a lane, in order to analyse the effects on the traffic flow; another useful feature is following a vehicle in its route (for instance, to follow an emergency vehicle).
Figure 10 shows a particular of the running simulation.

Figure 10: An image of the simulation with sumo-gui

4.8 The xml files output of a simulation

The user can choose how much information get as output of a simulation, which will be written after the simulation ends, by specifying some parameters in the input to the cml:

- *tripinfo-output*, to have detailed information about the trips, such as: departure speed, arrival speed, duration, waiting time, time-loss, etc.
- *vehroute-output*, to have detailed information for each vehicle: vehicle id, departure, route (a list of all the edges), arrival, fromTaz, toTaz, etc.
- *fcd-output*, to have the floating car data, i.e., the coordinates of each vehicle at each instant of the simulation (a very big data file)
- *summary*, to have some general information for each step of the simulation: number of vehicles running, mean speed, mean travel time, etc.

To sum up, the previous parameters can be given as input to run the simulation with the graphical user interface, through the following cml:

sumo-gui -v -n SS195.net.xml --route-files SS195.odtrips.rou.xml --duration-log.statistics true -b 25200 -e 28800 --time-to-teleport -1 --collision.action warn --step-length 0.5 --routing-algorithm dijkstra --time-to-impatience -1 --additional-files SS195.poly.xml --tripinfo-output SS195.tripinfo.out.xml --vehroute-output SS195.vehrou.out.xml --summary SS195_summary.out.xml

SUMO provides many Python scripts to read these output files (in the directory sumo/tools/output/), which can also be used as a starting point to customize a solution accordingly to specific requirements. Particularly useful in this task is the Python library *sumolib*, included in the tools of SUMO.
Figure 11 shows a scheme which summarises the main 5 steps required to build a traffic micro-simulation with SUMO.

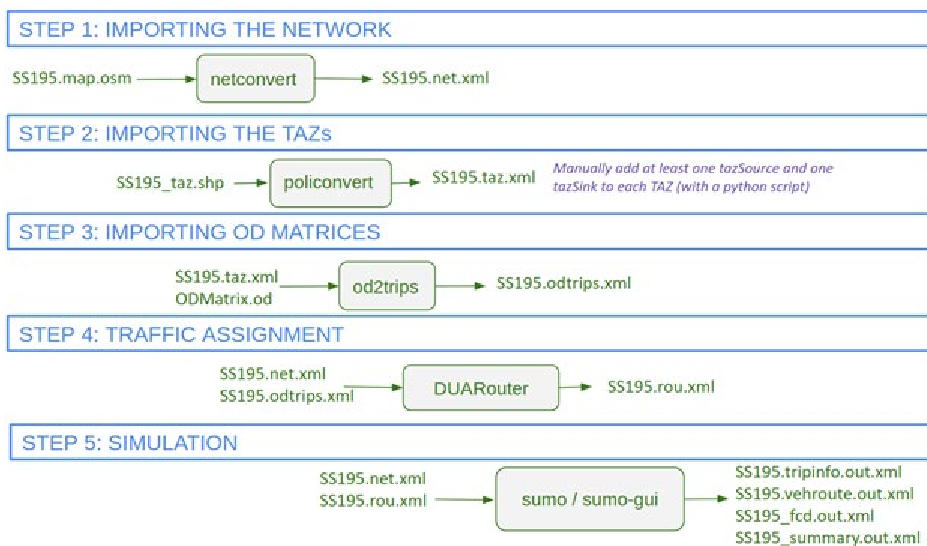

Figure 11: A scheme of the steps to build a SUMO micro-simulation.

5 A way to store information about simulations into MongoDB

A research activity of calibration, changing parameters many times can become difficult to be managed unless a way to record the important information about each run is put in place.

In fact, elaborating statistics and Key Performance Indices (travel time, number of vehicles, average speed, time-loss, etc.) is easier with the help of a database. Figure 12 shows a scheme of input files of a simulation, output files and its parsing through a custom python code to save the related information in a database.

Figure 12: input and output files of a simulation

For the case study MongoDB was used, after a data modeling analysis tailored to the specific application case of simulation recording. A simple combination of the Python libraries *pymongo* and the sumo library *sumolib* allow to save locally or to Atlas (in cloud) a complete set of information about each simulation (as shown in Figure 13).


```
_id: ObjectId("5fc0fe8fd5fbf9210579a38c")
params: Object
    input_dir: "/home/clem/Cagliari2020/osm_map/output/
    network: "/home/clem/Cagliari2020/osm_map/networks/
    aggiunta: "CON_NUOVO_TRATTO"
    fasciaoraria: "7_9"
    stepLength: 0.5
    routingalgorithm: "dijkstra"
    timetoimpatience: -1
    maxnumofiterations: 1
    carfollowingmodel: "Krauss"
output: Array
statistics: Object
    n_iters: 1
    loaded: 10755
    inserted: 10732
    arrived: 7404
    ended: 7404
    halting: 1008
    running: 3328
    waiting: 20
    meanSpeed: 9.09
    meanSpeedRelative: 0.44
    meanTravelTime: 601.37
    teleports: 0
    collisions: 0
    meanWaitingTime: 8.68
```

```
_id: ObjectId("5fc0fe8fd5fbf9210579a38c")
params: Object
    input_dir: "/home/clem/Cagliari2020/osm_map/output/Krauss/C
    network: "/home/clem/Cagliari2020/osm_map/networks/prova_ne
    aggiunta: "CON_NUOVO_TRATTO"
    fasciaoraria: "7_9"
    stepLength: 0.5
    routingalgorithm: "dijkstra"
    timetoimpatience: -1
    maxnumofiterations: 1
    carfollowingmodel: "Krauss"
output: Array
    0: Object
    1: Object
    2: Object
    3: Object
    4: Object
        _id: "car7440"
        type: "passenger"
        fromTaz: Object
            TAZ_id: 46
            name: "Cagliari Nuova Via San Paolo direzione Cagliari
            location: Object
                type: "Point"
                coordinates: Array
                    0: 39.216635
                    1: 9.096704
        toTaz: Object
            TAZ_id: 41
            name: "Cagliari Viale la Plaia zona di cordone"
            location: Object
                type: "Point"
                coordinates: Array
                    0: 39.216119
                    1: 9.104155
        duration: 101
        routeLength: 1507.08
        timeLoss: 6.52
        avgSpeed: 14.92
        depart: 25203.5
        arrival: 25304.5
        departDelay: 0.08
        departSpeed: 25.76
        arrivalSpeed: 13.44
        waitingTime: 0
        stopTime: 0
        speedFactor: 1.03
    5: Object
    6: Object
    7: Object
    8: Object
    9: Object
```


Figure 13: An example of simulation data in Atlas

The python code steps to be implemented are summarized here after:

- step 1: build a simulation dictionary, called *sim*, with all the input parameters;
- step 2: build a TAZ dictionary (reads TAZ names from a csv/txt file);
- step 3: build a dictionary with the statistics from *SS195.summary.out.xml* file and get average simulation performance indicators from last line in *summary.out.xml* file;
- step 4: build a list of information about each vehicle, called *all_vehicles_list*, getting information from output files *SS195_vehroute.out.xml* and *SS195_tripinfo.out.xml*, adding the names of the TAZs from the dictionary built in step 2;
- step 5: the previous steps collect all the data, while this last step stores all the data in the MongoDB Atlas cloud database (as shown in the python method in Figure 14).

```
# step 5: store all simulation info in a new collection of db
def store_in_mongodb(SUMO_DB_NAME, sim, all_vehicles_list, statistics):
    myclient =      pymongo.MongoClient("mongodb+srv://root:PSSWD@cluster0.plnvt.mongodb.net/" +
SUMO_DB_NAME + "?retryWrites=true&w=majority")
    dblist=myclient[SUMO_DB_NAME]
    mycollection=mydb["simulations"]
    simulation = {'params':sim, 'output':all_vehicles_list, 'statistics':statistics}
    sim_id = mycollection.insert_one(simulation)
    print('sim_id = %s' % sim_id.inserted_id)
    return sim_id.inserted_id
```

Figure 14: A python script to store the information about a simulation in the MongoDB Atlas cloud DB.

6 Conclusions and Future work

A real case study was presented as an example of how a micro-simulation scenario can be built with a map from Open Street Map and the traffic simulation platform SUMO. A detailed description was provided, starting from the network preparation, the definition of the Traffic Assignment Zones, and the trips obtained from Origin-Destination Matrices. A micro-simulation was run and some suggestions about how to use the output files and store the important information in a Mongo database.

This activity could be further developed in many directions: extending the study area, collecting further ground truth demand data to work on model calibration, considering public transport services, etc.

The SUMO platform is plenty of tools properly developed to import data collected by sensors or coming from video cameras, even from drones. These data would also allow a more detailed definition of the vehicles that could be used not only to better model the traffic, but also to compute the emission of pollutants. All these developments would provide a useful support to decision makers to prevent events of traffic congestions and environmental risks.

7 Acknowledgements

This work was partially supported by Italian Ministry of University and Research (MIUR), within the Smart City framework (Project Cagliari 2020, ID: PON04A2_00381).

A special thanks to MLabs srl (www.mlabls.eu) for providing the traffic assignment zones and related OD Matrices.

References

Alvarez Lopez, P., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.P., Hilbrich, R., Lücken, L., Rummel, J., Wagner, P., and Wießner, E. (2018). Microscopic Traffic Simulation using SUMO. *IEEE Intelligent Transportation Systems Conference (ITSC)*.

Aminia, S., Tilga, G., Buscha, F. (2019). Calibration of mesoscopic simulation models for urban corridors based on the macroscopic fundamental diagram. *Proceedings of hEART 2019*. 8th Symposium of the European Association for Research in Transportation.

Bachechi, C., Po, L. (2019). Traffic Analysis in a Smart City. *WI'19 Companion*, 2019, Thessaloniki, Greece, ACM.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), pp. 269–271.

Gawron, C. (1998). *An Iterative Algorithm to Determine the Dynamic User Equilibrium in a Traffic Simulation Model*. International Journal of Modern Physics C Vol. 9 (3), pp. 393-407

Krajzewicz, D., Hertkorn, G., Rössel, C., Wagner, P. (2002). An Example of Microscopic Car Models Validation using the open source Traffic Simulation SUMO. *Proceedings of Simulation in Industry, 14th European Simulation Symposium*. SCS European Publishing House (pp. 318-322). Dresden.

Krauß, S. (1998). *Microscopic Modeling of Traffic Flow: Investigation of Collision Free Vehicle Dynamics*. Hauptabteilung Mobilität und Systemtechnik des DLR Köln, ISSN 1434-8454

Maiorov, E.R., Ludan, I.R., Motta, J.D. (2019). *Developing a microscopic city model in SUMO simulation system*. Journal of Physics Conference Series 1368 0420812019.

McNally, M.G. (2000). The activity-based approach. *UCI-ITS-AS-WP-00-4*, Center for Activity Systems Analysis, University of California, Irvine, CA. https://escholarship.org/uc/item/5sv5v9qt

McNally, M. G., Rindt C. (2007). The Activity Based Approach. *Handbook of Transport Modelling*. Emerald Group Publishing Limited. (pp. 55-73). ISBN: 978-0-08-045376-7.

Po, L., Rollo, F., Bachechi, C., Corni, A. (2019). From Sensors Data to Urban Traffic Flow Analysis. *IEEE International Smart Cities Conference ISC2 2019*. Casablanca.

Schweizer, J. (2013). Sumopy: an advanced simulation suite for sumo. *Simulation of Urban MObility User Conference*. Springer, Berlin, Heidelberg.

Schweizer, J., Poliziani, C., Rupi, F., Morgano, D., Magi, M. (2021). Building a Large-Scale Micro-Simulation Transport Scenario Using Big Data. *ISPRS International Journal of Geo-Information*. 10, 165.

Ullah, M.R., Khattak, K.S., Khan, Z.H., Khan, M.A., Minallah, N., Khan, A.N. (2021). Vehicular Traffic Simulation Software: A Systematic Comparative Analysis. *Pakistan Journal of Engineering and Technology*, PakJET. Volume: 04, Number: 01, (pp. 66-78).

Web references

SUMO, Simulation of Urban MObility, http://sumo.dlr.de/userdoc/, accessed on 9 February 2022

Sumo source code https://github.com/eclipse/sumo

SUMO whole cities scenarios https://sumo.dlr.de/docs/Data/Scenarios.html

Sumo docs: Including elevation data in a network
https://sumo.dlr.de/docs/Networks/Elevation.html

sumo car-following-models
https://sumo.dlr.de/docs/Definition_of_Vehicles%2C_Vehicle_Types%2C_and_Routes.html#car-following_models

sumopy documentation
https://sumo.dlr.de/docs/Contributed/SUMOPy.html

sumopy code on github https://github.com/schwoz/sumopy

mongo db
https://mongodb.com