OSM2VISSIM TOOL TO IMPORT OPEN STREET MAP (OSM) INTO PTV VISSIM

An open-source OSM conversion tool for seamless automatized generation of microscopic traffic simulation for PTV VISSIM

Authors: Levente Tőkés and Tamás Tettamanti

Technical report for the open-source code of https://github.com/bmetrafficlab/OSM2VISSIM

Budapest University of Technology and Economics
Dept. of Control for Transportation and Vehicle Systems,
Fac. of Transportation Eng. and Vehicle Eng.

MŰEGYETEM 1782

Budapest, 2023

Contents

1 Purpose of the program

OpenStreetMap (OSM) is a free, editable map of the whole world being developed by volunteers and released under an open-content license. Due to this open-source nature of OSM, it is more and more important for software tools dealing with map data to have the capability of smooth importing. Software tools for modeling or simulating road traffic represent a group of programs for which proper maps are of crucial importance, i.e. traffic is simulated on a traffic network graph.

SUMO traffic simulator [2] has an efficient tool to import OSM, called OSMWebWizard providing an easy solution to start with SUMO based on a selection of an OSM excerpt together with randomized traffic demand. The main capability of OSMWebWizard are:

- demand generation,
- importing public transport,
- creating signal programs.

Similarly to SUMO traffic simulator's OSMWebWizard we developed an open-source conversion tool for PTV VISSIM microscopic simulation suite to help and support VISSIM users. According to the above, the main aim of the developed OSM2VISSIM tool is to provide the capability of:

- importing OSM map with all properties into PTV VISSIM, i.e. automatically generate VISSIM project file based on the OSM data with proper geometry and traffic light data;
- generating traffic simulation scenario with customized traffic demand.

2 Background

In this section, the applied map format and traffic simulation environment are briefly summarized used for the developed converter tool.

2.1 OpenStreetMap (OSM)

OSM is a collaborative map format that can be edited by any registered user, the data was entered into the system manually (by GPS survey), but currently, the map has several business and government sources. Existing data is stored in a PostGIS database according to the Mercator projection, and several additions have been made to the base map, which can be accessed by additional servers. The topological structure contains:

- node: spatial position with a coordinate;
- way: a series of vertices that represent either a polyline or a polygon;
- relation: groups of vertices or paths that have some properties;
- tag: descriptive data for nodes, ways, or relations.

2.2 PTV VISSIM and VISSIM COM programming

VISSIM is a microscopic traffic simulation tool based on the individual behavior of the vehicles. The simulator applies the Wiedemann car-following model [4]. VISSIM is widely used for diverse problems by traffic engineers in practice as well as by researchers for developments related to road traffic. VISSIM offers a user-friendly graphical interface (GUI) through which one can design the geometry of any type of road networks and set up simulations in a simple way. However, for several problems, the GUI is not satisfying. This is the case, for example, when the user aims to access and manipulate VISSIM objects during the simulation dynamically. To this end, an additional interface is offered by the manufacturer based on the COM (Component Object Model) technology enabling interprocess communication between software [1]. The VISSIM COM interface defines a hierarchical model in which the functions and parameters of the simulator originally provided by the GUI can be handled by programming as COM objects. Through VISSIM COM the user is able to manipulate the attributes of most of the internal objects dynamically [3].

3 Overview of the program

In this part, the main features of the developed program are introduced.

3.1 Starting the program

When starting the program, a GUI asks for the name of the OSM file. The user only has to provide the name of the file without the *.osm* extension. The source file has to be in the same folder where the program is, otherwise the program cannot find it.

After entering the filename and clicking on the *Set* button the window closes and creates a VISSIM input file (*.inpx*) with the name of the OSM file containing only one line: $< network >\backslash n< /network >$. Without this step the program is not working because at starting VISSIM needs an input file to save the data in.

To access and manipulate VISSIM objects, or in this case add new ones, a COM interface is required so the program and the simulation can communicate with each other to execute commands. In our development the applied programming language is Python but any other languages can be used as long as they can handle COM objects. As a first step the *win32com.client* must be imported. In this paper, the module has been imported as "com". After this the connection to the COM server can be created with the following command:

```
Vissim = com.gencache.EnsureDispatch("Vissim.Vissim")
```

In case of multiple VISSIM versions on the computer the version must be specified by adding the version number as well (i.e. "Vissim.Vissim.XXXX"). Once the cache has been generated, it is faster to call Dispatch which also creates the connection to VISSIM:

```
Vissim = com.Dispatch("Vissim.Vissim")
```

The next step is naming the input file that has been created earlier in the program. For this the path to the file has to be given either directly or with the help of the os.path.join() and os.getcwd() commands. The latter one is more practical since all the files are being created in the current working directory and the user does not have to change the path in the program.

After the input file is named it must be loaded into VISSIM with the Vissim.LoadNet() command which takes 2 arguments. The first one is the input file, the second one is a boolean variable which in the case of "True" value allows users to read network elements additionally. The default value of the later variable is "False" but for the program to work it needs to be set to "True".

3.2 Building the network

For creating links in the network VISSIM requires the coordinates of the links' points. To get these parameters from the OSM file the features of the file must be first exported into JavaScript Object Notation (JSON) format. A JSON file is a text-based, human-readable file that stores simple data structures and objects. A basic unit of JSON is a *key:value* pair which is similar to the Python dictionaries. To access these data a driver is required from the *ogr* module which is called OSM in this case. This driver can open and read OSM files and get the required layer and features in it. The first layer the program needs is the "lines" layer which contains the "way" elements of the OSM file including the roads and sidewalks. The features of the layer are saved in a list. The commands for these steps are:

```
driver = ogr.GetDriverByName('OSM')
data_source = driver.Open(osm_file)
layer = data_source.GetLayer('lines')
features=[x for x in layer]
```

The following steps are happening in a cycle that goes through all the features in the selected layer. First, the feature is exported into JSON:

```
data = feature.ExportToJson(as_object=True)
```

In the next step some of the feature's attributes are read. The most important ones are *coordinates*, *highway* and *other_tags*. The first one is needed for creating the links, the second one helps to assort the features that are in the layer but are irrelevant to the simulation, the last one contains extra information that might be needed, like the number of lanes.

To find the relevant features a list of key values is given for the *highway* key. If the value of *highway* is not in the list the program skips the feature. The list contains the following elements: *steps; footway; cycleway; motorway; trunk; primary; secondary; tertiary; unclassified; residental; motorway_link; trunk_link; primary_link; secondary_link; tertiary_link; service; track; bus_guideway; escape; raceway; road; busway; living_street; residential.*

For adding links to the network coordinates and the width and number of lanes are needed. The program starts with determining the last one. In OSM there are 2 keys containing information about it, *lanes* and *turn:lanes*, furthermore both of them can include information about the forward and backward directions (e.g. *turn:lanes:forward*). The program checks if the current object contains the key with the smallest group and if not, then checks the next key and so on. Checking the keys one by one is necessary because some OSM objects do not contain all of them which can lead to errors. The order of these keys is the following:

turn:lanes:forward/backward $\longrightarrow$ *lanes:forward/backward* $\longrightarrow$ *lanes*

Furthermore, if the object does not hold any of these keys the program checks if it contains the key *oneway*. If it does then the object is handled like a link with one lane but if it does not then a backward direction is given with one lane.

In OSM no information is given about the width of the lanes. That is why all the footways are 1 meter while the cycle ways 1.5 meters wide. Most of the car ways are 3.5 meters wide except when a backward direction is generated as explained in the last paragraph.

The last step before creating a link is generating its coordinates. The *way* elements of the exported OSM file are ordered lists of nodes, and each node has its own coordinates. This information can be extracted as explained above. The problem is that OSM is based on the WGS-84 coordinate system, while VISSIM uses the Web Mercator, so a transformation must be executed. For this the following equations are used:

$Latitude = ln(tan\frac{\pi}{4} + \frac{Latitude\ in\ radian}{2})) \times Earth\ radius$
$Longitude = Longitude\ in\ radian \times Earth\ radius$

In Python:

```
def Convert_to_mercator(lat, lon):
    RADIUS = 6378137.0
    lat_mer = math.log(math.tan(math.pi / 4 + math.radians(lat) / 2)) * RADIUS
    lon_mer = math.radians(lon) * RADIUS
    return [lon_mer, lat_mer]
```

The last step creating the link itself, which can be done with the following command:

```
Vissim.Net.Links.AddLink(Key, WktLinestring, LaneWidths)
```

where *Key* is the key of the generated link, *WktLinestring* contains the coordinates of the points of the link and *LaneWidths* is a list with the width of each lane. In this program the key is the ID of the *way* element in OSM. The linestring is created with the *shapely.geometry.Linestring(list of coordinates)* command. The length of the list given as *LaneWidths* is equal to the number of lanes, which was already defined as explained above, just as the value of width. Currently the widths for one link are all equal. The final command:

```
Vissim.Net.Links.AddLink(way_id, str(sg.LineString(mercator)), widths)
```

where *mercator* is a list of the transformed coordinates.

The opposite direction (if exists) is created with a specific command, which needs the original link and the number of lanes (see above) as inputs. For each link generated like this, the attribute *Name* is set to the opposite key of the original link. The command with the variables:

```
Vissim.Net.Links.GenerateOppositeDirection(Vissim.Net.Links.ItemByKey(way_id), lanes_backward)
```

For later use the links used by vehicles, bikes and pedestrians are sorted in 3 different lists. Also for later use, the name of the links and that of their opposite direction, which has a common point, are placed into a list.

3.3 Traffic signal heads

The placing of the signal head starts with accessing the XML-formatted OSM file. Through a cycle it looks for *node* objects that have an attribute with the value *traffic_signals*. If it finds one, it saves the object's ID, and another cycle looks for *way* objects that has a reference with the value of this ID. These are the links controlled

by the traffic light. In the case of a two-way street, the selected link is the one that's ending is closer to the traffic light.

The next thing to do is determine the ID of its signal controller. In general, a new signal controller is created with an ID equal to the number of signal controllers, but there is an exception. For each traffic light the program searches for other signal heads within 30 meters. In case it finds one, the ID of the new signal controller is equal to the ID of the found signal controller. Then, the signal heads' coordinates are saved in a dictionary.

The final parameter the program needs to place the signal heads is their position. The first step is finding the one point of the link that is closest to the traffic light. The program saves the index (X) of this point and creates a new link with the first X points of the original link. The length of this new link gives the position of the signal head on the original link. The only problem this way is that the closest point is often inside the junction the traffic light is controlling, so 5 meters is subtracted from the position. It is not a calculated number, it is only based on trying out different values. In the case of a link with multiple lanes, a signal head is placed on each lane.

Now, that the position and the signal controller's ID are determined, the signal heads can be placed. The most important commands are the following:

```
Vissim.Net.SignalControllers.AddSignalController(SC_ID)
Vissim.Net.SignalControllers.ItemByKey(SC_ID).SGs.AddSignalGroup(len(Vissim.Net.
    SignalControllers.ItemByKey(SC_ID).SGs)+1)
Vissim.Net.SignalHeads.AddSignalHead(SH_ID, lane, Links[-1].AttValue('Length2D')-5)
```

Links[-1] refers to the new link that helps to determine the signal head's position. At the and of the cycle this link is removed from the network:

```
Vissim.Net.Links.RemoveLink(Links[-1])
```

The main steps of placing the signal heads are:

1. Searching for *traffic light* objects in the OSM file.
2. Searching for the *way* objects that have the *traffic light* object's ID as reference.
3. Searching for other *traffic light* objects within 30 meters.
4. Determining the signal controller's ID.
5. Searching for the link's point closest to the *traffic light* object's coordinates.
6. Creating a link that ends at the closest point.
7. Placing the signal heads 5 meters before the new link's end.
8. Deleting the new link.

3.4 Connectors

For creating connectors between links the program follows the rule that all directions are allowed except in case of restrictions, which can be found as *relation* objects in the OSM file. There are two types of restrictions. The first one forbids certain directions (e.g. *no_right_turn*, while the other one allows only one (e.g. *only_straight_on*). These objects contain 3 keys, *from*, *via* and *to* with the value of other objects' ID. The *from* and *to* keys refer to the way objects that the restriction is for. For later use, these are saved in different lists based on the type of restriction.

The next step is starting a cycle that goes through the links. The program first checks if the link (henceforth: Link1) is part of any *only* type restriction and saves it in a boolean variable named *has_rest*. Another cycle starts going through the links (henceforth: Link2). If the value of *has_rest* is *True*, the program checks if the [Link1, Link2] list can be found in the list containing the *only* type restrictions. In case the list does not contain it Link2 is skipped. Link2 is also skipped if [Link1, Link2] is not in the list that contains the link with common points saved earlier while building the network.

If skipping is not necessary a third cycle is started which examines the points of Link2. The program checks if any of the points is within 3 or 5 meters (depending on the length of Link1 and Link2) of the first or last point of Link1. If the first point is, the program handles Link1 as the *to* object and Link2 as the *from* object

and vice versa if the last point is. However, Link2 is skipped if the [Link1,Link2] pair can be found in the list containing the *no_turn* type restrictions.

The method of placing the connector is the same in the two cases and similar to the method of placing the signal heads (see 3.3). If the Link1 is the *from* object, a cycle searches for the point of Link2 that is closest to the last point of Link1. After it is found a new link is created with the points of Link2 but ends at the closest point. The length of this new link gives the position of the connector's ending point. This position might be altered depending on the length of Link1 and Link2 and the role of Link1 (*from/to*). The reason for this change is partly aesthetics but a more important reason is that in case the first or last point of Link1 and the closest point have the same coordinates the length of the created connector is 0. The new link created in the process is deleted at the end of the cycle.

The last part before placing the connector is choosing the number of its lanes and the lanes where the connector starts and ends. For the first problem, the rule is that the number of lanes is equal to the minimum number of lanes between Link1 and Link2:

```
lanes=min(way_1.AttValue('NumLanes'),way_2.AttValue('NumLanes'))
```

The rule for starting and ending lanes is the same. If the connector has only 1 lane, it starts and ends in the outer lanes while in the case of multiple lanes, it starts and ends in the inner lanes.

Finally, the command to create a connector:

```
Vissim.Net.Links.AddConnector(Key, FromLane, FromPos, ToLane, ToPos, NumberOfLanes,
    WktLinestring)
```

The steps of creating connectors are:

1. Searching for and saving restrictions.
2. Choosing Link1.
3. Searching for restrictions containing Link1.
4. Choosing Link2.
5. Checking if Link1 and Link2 have restrictions or common point.
6. Searching for close points.
7. Searching for the closest point.
8. Creating a new link ending at the closest point.
9. Determining the number of lanes and the starting and ending lanes.
10. Creating the connector.
11. Deleting the link created in step 8.

3.5 Program ending

The last step of creating the input file is saving it. By default, the name the file is saved by is always the name of the original OSM file. The saving is executed with the following command:

```
Vissim.SaveNetAs(os.path.join(os.getcwd(), str(osm_file)+".inpx"))
```

It is important to note that while the program is running (i.e. before saving) the network cannot be edited because it may cause program and VISSIM crash.

4 A working example

To help understand the process of the program a minimal example is presented in this section. The OSM file named *example* can be found in the *Working example* folder. The coordinates of the example OSM map are 47.51576°, 19.05958° (a junction in Budapest, Hungary) as shown in Fig. 1. Do not forget that the exported/-

Figure 1: Example junction in OSM (47.51576°, 19.05958°)

downloaded OSM file must be in the same folder as the program. After pasting the OSM file into the working directory the program can be started. First, the GUI opens asking for the name of the OSM file. Do not put the extension (*.osm*) on the end of the file's name, only the name of the file must be given as shown in Fig. 2. After entering the file name and clicking on the *Set* button the window closes and VISSIM opens.

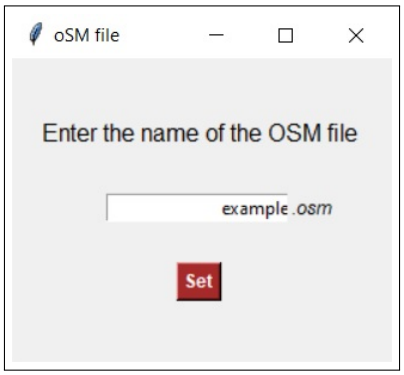

Figure 2: GUI for the name of the OSM file

After starting the *Start page* of VISSIM should be closed (top right corner). At first the world map is shown but by clicking on *Show entire network* VISSIM should zoom to the new road network (Fig. 3). The program starts building the network automatically so it can be observed as the links are placed.

Before going on placing the signal head the network should look like in Fig. 4 without traffic lights and/or connectors. During the process of generating signal heads new links can be seen that only appear for a moment before getting deleted. These links are only used temporally for technical reason, to determine the position of the signal heads (see Section 3.3).

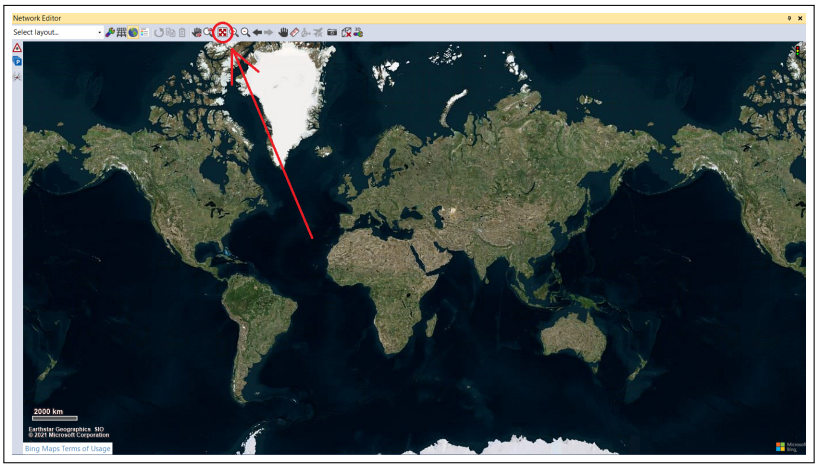

Figure 3: Click on *Show entire network* in VISSIM

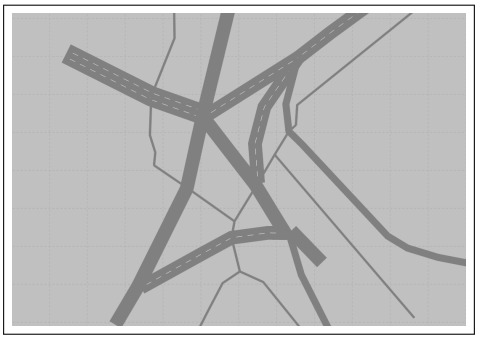

Figure 4: Example network before generating signal heads

Fig. 5 depicts the creation of Link-8 to find the position of the signal head placed on the sidewalk before the crossing. The signal head is placed 5 meters before the end of Link-8. In the next step Link-8 is deleted and the program searches for the next traffic light in an iterative loop. If all signal heads are generated the network should look as shown by Fig. 6.

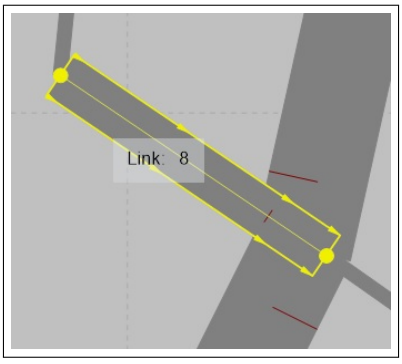

Figure 5: Example for placing a signal head

The last part is creating the connectors between links. If *Toggle wireframe* is on pink lines should mark the connectors but, as seen in Fig. 7, they are missing. The program continues by creating the connectors for cycle ways but in the example junction there are none so it goes right to car ways. Just like in case of the traffic lights there are links appearing for a second determining the position of the connector's starting and ending position. By the end of the program the network should look like in Fig. 8.

If the file is saved it means the program has ended and the network might be edited.

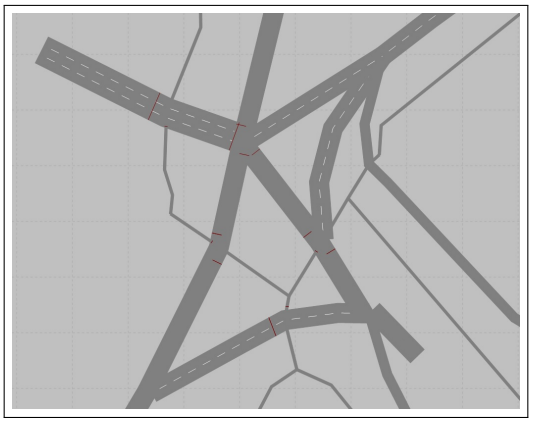

Figure 6: Example network after generating the signal heads

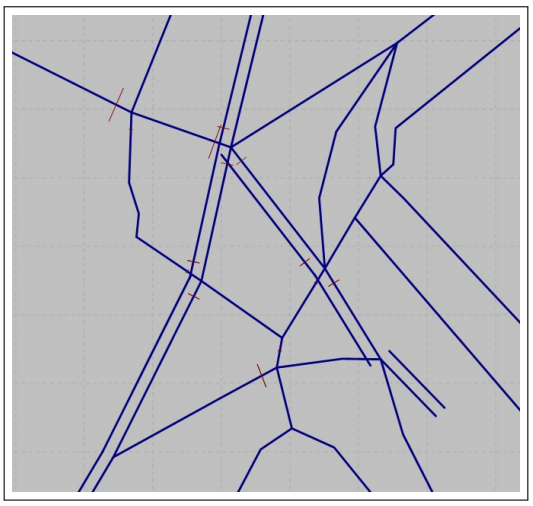

Figure 7: Example network without connectors

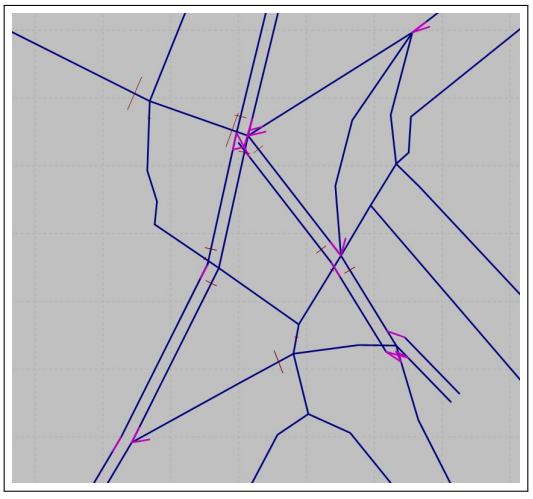

Figure 8: Example network with connectors

Finally, it is important to note the capability of the program to handle roundabouts. This is not presented by the above working example, but you can consider an example (*Esztergom.osm*) in the *Examples* folder.

5 Limitations

Hereby, the known limitation of the program are presented.

5.1 False direction problem

Although the program is capable of building the road network, placing signal heads and creating connectors, there are limitations that are basically the consequences of the OSM's imperfection.

During the network creation, the order of link (edge) points is the same as that of the coordinates in the OSM file but this order does not always give a continuous direction for the given link. As an example consider a part of the *BME.osm* file (see *Examples* folder under https://github.com/bmetrafficlab/OSM2VISSIM) depicted by Fig. 9 after conversion.

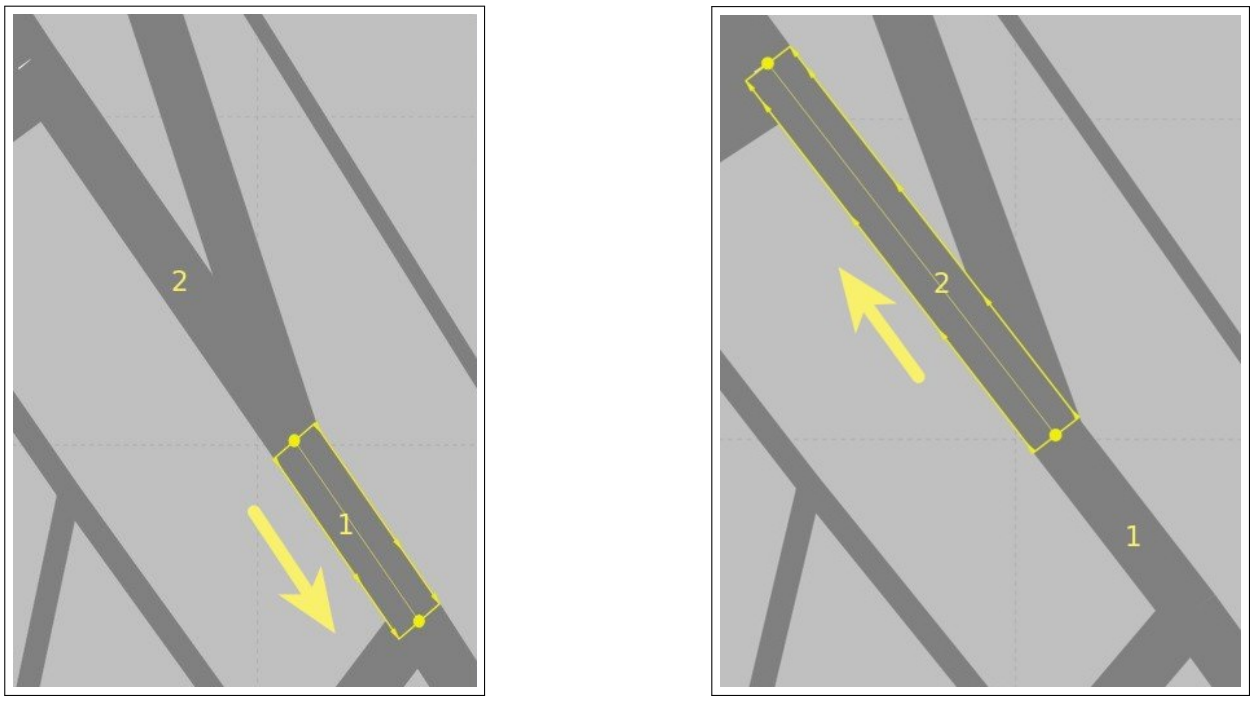

Figure 9: Links with opposite direction before the manual inversion of the directions and connectors

The two links are successive edges in the network but their directions are opposite due to the OSM's imperfection. After manually generating the opposite directions and creating the corresponding connectors, the continuity problem can be resolved. Nevertheless, after this operation the final network might look fallen apart in VISSIM, and so need manual correction.

Although the issue, presented above, generally does not cause any complication regarding the program's operation and it is rather an aesthetic issue, it might still cause some errors in very specific circumstances. In these special cases, it is suggested to manually remove the problematic edge from the OSM file before the execution of the program.

5.2 Identical lane width

Another limitation regarding the link generation is the lane width setting. Since usually there is no information about the lane width in the OSM files, the lane width parameter of the links is set to 3.5 meters identically. A possible solution to this issue might be to automatically choose the lane width based on the road category that can be extracted from the OSM file's *highway* attribute (this solution is not yet coded in the program).

5.3 No gradient modeling

Although VISSIM is capable of height modeling, our program only works in 2D. Fortunately, OSM data contain restrictions that forbid connections between *way* objects that are not on the same height, i.e., even if they cross each other in 2D there will be no connection created between the 2 links.

5.4 Issues of the traffic signal heads

Although for placing the signal heads a working solution was presented (see Subsection 3.3), the generated positions might be not perfect. During the tests in some of the example files a few signal heads have been placed inside the junction area. Raising the distance from the link's end may cause too long distance at other junctions. Therefore, in these special cases, for now, the problem can only be fixed with additional manual editing.

In some other cases, the signal heads for pedestrian crossings are duplicated. The reason for this is the OSM data structure. When searching for signal head objects the program looks for any object that has an attribute that's value is *traffic_signals*. There are two types of these objects, one of them being for junctions and the other one for crossings. In most cases, only one of these belongs to a given link. However, in some other cases, two objects with the same attribute value *traffic_signals* are saved for a crossing. As an example Fig. 10 is shown where *crossing* attribute has the value *traffic_signals* too. Due to this redundancy, duplicated signal heads are generated in VISSIM. This redundancy can be handled by additional manual editing in VISSIM.

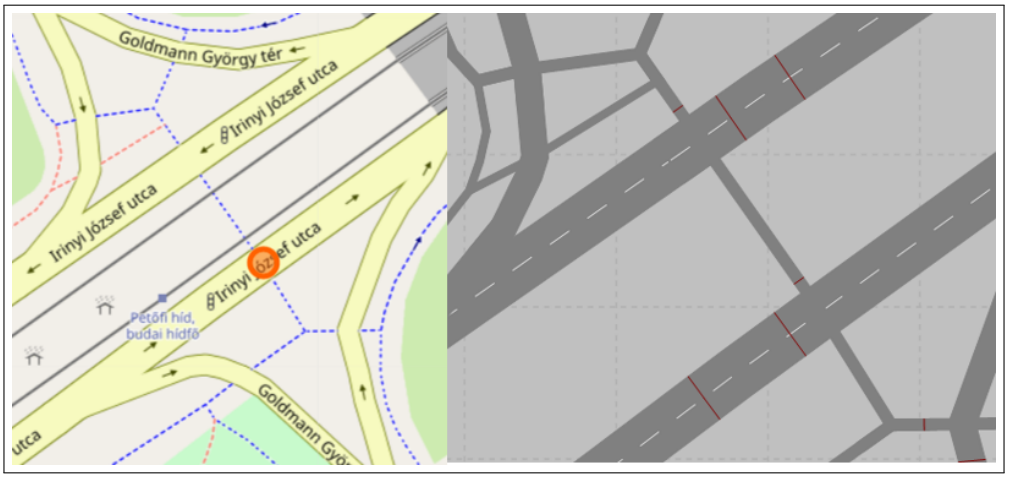

Figure 10: Duplicated signal heads in VISSIM

5.5 Issue on creating connectors

Creating connectors is one of the most critical parts of the program due to the restrictions and specific circumstances that are not revealed until testing for the first time. Generally, the program creates all connectors but it might happen that a connector will be missing in the produced network.

Another issue, when creating connectors, is the relatively high processing time. Due to the loop based iteration in this part of the program the whole process can last for several minutes depending on the size of the network. Although the program only checks the links with common points, since it examines every links' points, the period of the program grows nearly exponentially.

5.6 Manual editing for traffic signal programs and vehicle inputs

Finally, it is noted that manual editing is needed before running the simulation. On the one hand, signal programs must be created for each of the signal controllers. On the other hand, vehicle inputs must be placed on the relevant links asn defined as they are not generated automatically.

The simulation can be run without these manual settings, but warnings will be produced by VISSIM.

References

[1] D Box. *Essential COM*. Addison-Wesley, London, 1988. ISBN 0-201-63446-5.

[2] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[3] T. Tettamanti and M.T. Horváth. *A practical manual for Vissim-COM programming in Matlab and Python – 5th edition for Vissim version 2020 and 2021*, 5th edition, 2021.

[4] R Wiedemann. Simulation des straßenverkehrsflusses. *Schriftenreihe des Instituts für Verkehrswesen der Universität Karlsruhe*, 8, 1974.

6 Appendix: the full code of the OSM2VISSIM program

The code is available at github.com/bmetrafficlab/OSM2VISSIM.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 20 14:27:03 2023

@author: tokes
"""

import win32com.client as com
import os
import math
from osgeo import ogr
import shapely.geometry as sg
import xml.etree.ElementTree as ET
import tkinter as tk


file_window=tk.Tk(className='OSM file')

file_canvas=tk.Canvas(file_window, width=250, height=200)
file_canvas.pack()

file_label=tk.Label(file_window, text='Enter the name of the OSM file', font=('helvetica', 12)
    )
file_canvas.create_window(125, 50, window=file_label)

file_entry=tk.Entry(file_window, width=20, justify="right")
file_canvas.create_window(125, 100, window=file_entry)

osm_label=tk.Label(file_window, text='.osm', font=('helvetica', 9, 'italic'))
file_canvas.create_window(200, 100, window=osm_label)

def Set_osm_file():

    global osm_file

    osm_file=file_entry.get()

    file_window.destroy()

file_button=tk.Button(file_window, text='Set', command=Set_osm_file, font=('helvetica', 9, '
    bold'), bg='brown', fg='white')
file_canvas.create_window(125, 150, window=file_button)

file_window.mainloop()

#Create base .inpx XML file
# f= open(str(osm_file)+".osm.inpx","w+")
f= open(osm_file+".osm.inpx","w+")
f.write("<network>\n</network>")
f.close()

# Connnecting COM server
Vissim= com.Dispatch("Vissim.Vissim")

#Get Vissim version for XML file

Filename = os.path.join(os.getcwd(),osm_file+".osm.inpx") # in Current Working Directory
Vissim.LoadNet(Filename, True)

# ----Base data------
# This base data were added in order to solve fulfill and solve empty file issue
Vissim.Net.VehicleClasses.AddVehicleClass(1)
#Vissim.Net.VehicleTypes.AddVehicleType(1)
##Vissim.Net.TimeDistributions.AddTimeDistributionNormal(1)
##Vissim.Net.DrivingBehaviors.AddDrivingBehavior(1)
##Vissim.Net.LinkBehaviorTypes.AddLinkBehaviorType(1)
```

```
#Vissim.Net.LinkBehaviorTypes.ItemByKey(1).VehClassDrivBehav.AddVehClassDrivingBehavior(Vissim
    .Net.VehicleClasses.ItemByKey(30))
#Vissim.Net.DisplayTypes.AddDisplayType(1)
##Vissim.Net.Levels.AddLevel(1)
# --------Links---------------
# Input parameters to add links

driver=ogr.GetDriverByName('OSM')
data_source = driver.Open(osm_file+'.osm')

tree = ET.parse(osm_file+'.osm')
root = tree.getroot()

no_turn_restrictions=[]
only_restrictions=[]

SH_ID=1

layer = data_source.GetLayer('lines')

features=[x for x in layer]

short_links=0
Links=[]
link_key=1
carways=[]
footways=[]
cycleways=[]
all_cycleway_coords={}
all_carway_coords={}
carways_with_crossing=[]
cycleways_with_crossing=[]

print('Creating links')

for feature in features:

    data=feature.ExportToJson(as_object=True)

    way_id=int(data['properties']['osm_id'])
    coords=data['geometry']['coordinates']
    mercator=[]
    other_tags=data['properties']['other_tags']
    name=data['properties']['name']
    highway=data['properties']['highway']
    highways=['steps','footway','cycleway','motorway', 'trunk', 'primary', 'secondary', '
    tertiary', 'unclassified', 'residental', 'motoway_link', 'trunk_link', 'primary_link', '
    secondary_link', 'tertiary_link', 'service', 'track', 'bus_guideway', 'escape', 'raceway',
     'road', 'busway', 'living_street', 'residential']

    if highway in highways:

        if other_tags and ('"lanes:forward"' in other_tags or '"lanes:backward"' in other_tags
     or '"turn:lanes:forward"' in other_tags or '"turn:lanes:backward"' in other_tags):

            if '"turn:lanes:forward"' in other_tags:

                feat=[x for x in other_tags.split(',') if '"turn:lanes:forward"' in x][0]
                lanes_forward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

                try:

                    feat=[x for x in other_tags.split(',') if '"lanes"' in x][0]
                    lanes_backward=int(feat[feat.rfind('>')+2:feat.rfind('"')])-lanes_forward

                except:

                    try:
```

```
                    feat=[x for x in other_tags.split(',') if '"turn:lanes:backward"' in x
][0]
                    lanes_backward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

                except:

                    lanes_backward=0


        elif '"turn:lanes:backward"' in other_tags:

            feat=[x for x in other_tags.split(',') if '"turn:lanes:backward"' in x][0]
            lanes_backward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

            try:

                feat=[x for x in other_tags.split(',') if '"lanes"' in x][0]
                lanes_forward=int(feat[feat.rfind('>')+2:feat.rfind('"')])-lanes_backward

            except:

                try:

                    feat=[x for x in other_tags.split(',') if '"turn:lanes:forward"' in x
][0]
                    lanes_forward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

                except:

                    lanes_forward=0

        elif '"lanes:forward"' in other_tags:

            feat=[x for x in other_tags.split(',') if '"lanes:forward"' in x][0]
            lanes_forward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

            try:

                feat=[x for x in other_tags.split(',') if '"lanes"' in x][0]
                lanes_backward=int(feat[feat.rfind('>')+2:feat.rfind('"')])-lanes_forward

            except:

                try:

                    feat=[x for x in other_tags.split(',') if '"lanes:backward"' in x][0]
                    lanes_backward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

                except:

                    lanes_backward=0

        elif '"lanes:backward"' in other_tags:

            feat=[x for x in other_tags.split(',') if '"lanes:backward"' in x][0]
            lanes_backward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

            try:

                feat=[x for x in other_tags.split(',') if '"lanes"' in x][0]
                lanes_forward=int(feat[feat.rfind('>')+2:feat.rfind('"')])-lanes_backward

            except:

                try:

                    feat=[x for x in other_tags.split(',') if '"lanes:forward"' in x][0]
                    lanes_forward=len(feat[feat.rfind('>')+2:feat.rfind('"')].split(';'))

                except:
```

```
                lanes_forward=1

        lanes=0

    else:

        try:

            feat=[x for x in other_tags.split(',') if '"lanes"' in x][0]
            lanes=int(feat[feat.rfind('>')+2:feat.rfind('"')])

        except:

            lanes=1

        lanes_forward=0
        lanes_backward=0

    def Convert_to_mercator(lat, lon):

        RADIUS = 6378137.0
        lat_mer=math.log(math.tan(math.pi / 4 + math.radians(lat) / 2)) * RADIUS
        lon_mer=math.radians(lon) * RADIUS

        return [lon_mer, lat_mer]

    for k in range(len(coords)):

        mercator.append(Convert_to_mercator(coords[k][1], coords[k][0]))

    if highway=='cycleway':

        width=1.5

    elif highway=='footway' or highway=='steps':

        width=1

    elif other_tags and '"oneway"' in other_tags:

        feat=[x for x in other_tags.split(',') if '"oneway"' in x][0]
        oneway=feat[feat.rfind('>')+2:feat.rfind('"')].split(';')[0]

        if '"junction"' in other_tags:

            oneway='yes'

        if oneway=='yes':

            width=3.5

        else:

            lanes_forward=1
            lanes_backward=1

            if lanes==1:

                width=2.75

            else:

                width=3.5

            lanes=0

    elif other_tags and not('"oneway"' in other_tags):

        if '"junction"' in other_tags:
```

```
            try:

                feat=[x for x in other_tags.split(',') if '"lanes"' in x][0]
                lanes=int(feat[feat.rfind('>')+2:feat.rfind('"')])

                width=3.5

                lanes_forward=0
                lanes_backward=0

            except:

                width=3.5

        else:

            lanes_forward=1
            lanes_backward=1

            if lanes==1:

                width=2.75

            else:

                width=3.5

            lanes=0

    else:

        width=3.5

    widths=[]

    if width==1.5:

        key_list=list(all_cycleway_coords.keys())
        value_list=list(all_cycleway_coords.values())

        for way_1_coords in coords:

            for way_2_coords in value_list:

                if way_1_coords in way_2_coords:

                    cycleways_with_crossing.append([str(way_id), str(key_list[value_list.
index(way_2_coords)])])
                    cycleways_with_crossing.append(['-'+str(way_id), str(key_list[
value_list.index(way_2_coords)])])
                    cycleways_with_crossing.append([str(way_id), '-'+str(key_list[
value_list.index(way_2_coords)])])
                    cycleways_with_crossing.append(['-'+str(way_id), '-'+str(key_list[
value_list.index(way_2_coords)])])

        all_cycleway_coords[way_id]=coords

    elif width==3.5 or width==2.75:

        key_list=list(all_carway_coords.keys())
        value_list=list(all_carway_coords.values())

        for way_1_coords in coords:

            for way_2_coords in value_list:

                if way_1_coords in way_2_coords:
```

```
                        carways_with_crossing.append([str(way_id), str(key_list[value_list.
    index(way_2_coords)])])
                        carways_with_crossing.append(['-'+str(way_id), str(key_list[value_list
    .index(way_2_coords)])])
                        carways_with_crossing.append([str(way_id), '-'+str(key_list[value_list
    .index(way_2_coords)])])
                        carways_with_crossing.append(['-'+str(way_id), '-'+str(key_list[
    value_list.index(way_2_coords)])])

            all_carway_coords[way_id]=coords


        if lanes_forward==0:

            for k in range(lanes):

                widths.append(width)

        else:

            for k in range(lanes_forward):

                widths.append(width)

        Links.append(
            Vissim.Net.Links.AddLink(way_id, str(sg.LineString(mercator)), widths)
            )

        Links[-1].SetAttValue('Name', way_id)

        if width==3.5 or width==2.75:

            carways.append(Links[-1])

        elif width==1.5:

            cycleways.append(Links[-1])

        else:

            footways.append(Links[-1])

        if lanes_forward!=0:

            Links.append(
                Vissim.Net.Links.GenerateOppositeDirection(Vissim.Net.Links.ItemByKey(way_id),
     lanes_backward)
                )

            Links[-1].SetAttValue('Name', -way_id)

            if width==3.5 or width==2.75:

                carways.append(Links[-1])

            elif width==2:

                cycleways.append(Links[-1])

            else:

                footways.append(Links[-1])

            link_key=link_key+1

        link_key=link_key+1

print('Creating signal heads')

lamp_coords={}
```

```
for nodes in root:
    if nodes.tag=='node':
        for node in nodes:
            if node.attrib['v']=='traffic_signals':
                lamp_id=nodes.attrib['id']
                for ways in root:
                    if ways.tag=='way':
                        for way in ways:
                            if 'ref' in way.attrib and way.attrib['ref']==lamp_id:
                                SC_near=False
                                way_id=ways.attrib['id']
                                for coords in lamp_coords:
                                    if math.dist(Convert_to_mercator(float(nodes.attrib['lat'
    ]), float(nodes.attrib['lon'])), lamp_coords[coords])<30:
                                        SC_ID=list(lamp_coords).index(coords)+1
                                        SC_near=True
                                        break
                                if not(SC_near):
                                    SC_ID=len(Vissim.Net.SignalControllers)+1
                                    Vissim.Net.SignalControllers.AddSignalController(SC_ID)
                                Vissim.Net.SignalControllers.ItemByKey(SC_ID).SGs.
    AddSignalGroup(len(Vissim.Net.SignalControllers.ItemByKey(SC_ID).SGs)+1)
                                link_found=False
                                for link in Vissim.Net.Links:
                                    if link.AttValue('Name')==way_id:
                                        # if link.Lanes.GetAll()[0].AttValue('Width')==3.5 and
     node.attrib['k']=='crossing':
                                        #     continue
                                        linkpolypts_1=link.LinkPolyPts.GetAll()
                                        link_found=True
                                        break
                                if not(link_found):
                                    continue
                                for link in Vissim.Net.Links:
                                    if link.AttValue('Name')=='-'+way_id:
                                        linkpolypts_2=link.LinkPolyPts.GetAll()
```

```
                                        oneway=False
                                        break
                                    else:
                                        oneway=True
                                link_start_1=[linkpolypts_1[-1].AttValue('X'), linkpolypts_1
[-1].AttValue('Y')]
                                try:
                                    link_start_2=[linkpolypts_2[-1].AttValue('X'),
linkpolypts_2[-1].AttValue('Y')]
                                except:
                                    pass
                                if oneway:
                                    link_name=way_id
                                else:
                                    if math.dist(Convert_to_mercator(float(nodes.attrib['lat'
]), float(nodes.attrib['lon'])), link_start_1)<math.dist(Convert_to_mercator(float(nodes.
attrib['lat']), float(nodes.attrib['lon'])), link_start_2):
                                        link_name=way_id
                                    else:
                                        link_name='-'+way_id
                                for link in Vissim.Net.Links:
                                    try:
                                        if link.AttValue('Name')==link_name:
                                                polypoints=link.LinkPolyPts.GetAll()
                                                for point_index in range(len(polypoints)):
                                                    if point_index!=0:
                                                        if math.dist(Convert_to_mercator(float
(nodes.attrib['lat']), float(nodes.attrib['lon'])), [polypoints[point_index].AttValue('X')
, polypoints[point_index].AttValue('Y')])<math.dist(Convert_to_mercator(float(nodes.attrib
['lat']), float(nodes.attrib['lon'])), [polypoints[point_index-1].AttValue('X'),
polypoints[point_index-1].AttValue('Y')]):
                                                            closest_point=polypoints[
point_index]
                                                    else:
                                                        pass
                                                points_for_link=polypoints[0:polypoints.index(
closest_point)+1]
                                                coords_for_link=[]
                                                for point in range(len(points_for_link)):
                                                    coords_for_link.append([points_for_link[
point].AttValue('X'), points_for_link[point].AttValue('Y')])
```

```
                                                Links.append(
                                                    Vissim.Net.Links.AddLink(0, str(sg.
    LineString(coords_for_link)), [3.5])
                                                    )

                                                for lane in link.Lanes:

                                                    Vissim.Net.SignalHeads.AddSignalHead(SH_ID
    , lane, Links[-1].AttValue('Length2D')-5)

                                                    Vissim.Net.SignalHeads.ItemByKey(SH_ID).
    SetAttValue('SG', Vissim.Net.SignalControllers.ItemByKey(SC_ID).SGs.ItemByKey(len(Vissim.
    Net.SignalControllers.ItemByKey(SC_ID).SGs)))
                                                    SH_ID=SH_ID+1

                                                Vissim.Net.Links.RemoveLink(Links[-1])

                                        if not(lamp_id in lamp_coords) and not(SC_near):

                                            lamp_coords[nodes.attrib['id']]=
    Convert_to_mercator(float(nodes.attrib['lat']), float(nodes.attrib['lon']))

                                    except:

                                        pass

for relations in root:

    if relations.tag=='relation':

        for relation in relations:

            if 'k' in relation.attrib:

                if relation.attrib['k']=='restriction' and (relation.attrib['v']=='
    no_right_turn' or relation.attrib['v']=='no_left_turn'):

                    for ways in relations:

                        if 'role' in ways.attrib:

                            if ways.attrib['role']=='from':

                                from_way_id=int(ways.attrib['ref'])

                            if ways.attrib['role']=='to':

                                to_way_id=int(ways.attrib['ref'])

                        else:

                            continue

                    no_turn_restrictions.append([from_way_id, to_way_id])
                    no_turn_restrictions.append([-from_way_id, to_way_id])
                    no_turn_restrictions.append([from_way_id, -to_way_id])
                    no_turn_restrictions.append([-from_way_id, -to_way_id])

                if relation.attrib['k']=='restriction' and (relation.attrib['v']=='
    only_straight_on' or relation.attrib['v']=='only_right_turn' or relation.attrib['v']=='
    only_left_turn'):

                    for ways in relations:

                        if 'role' in ways.attrib:

                            if ways.attrib['role']=='from':

                                from_way_id=int(ways.attrib['ref'])
```

```
                            if ways.attrib['role']=='to':
                                to_way_id=int(ways.attrib['ref'])
                        else:
                            continue
                    only_restrictions.append([from_way_id, to_way_id])
                    only_restrictions.append([-from_way_id, to_way_id])
                    only_restrictions.append([from_way_id, -to_way_id])
                    only_restrictions.append([-from_way_id, -to_way_id])

def Create_connectors(way_type):
    if way_type==carways:
        way_2_list=carways_with_crossing
        print('Creating carway connectors')
    elif way_type==cycleways:
        way_2_list=cycleways_with_crossing
        print('Creating cycleway connectors')
    serial_number=0
    percentages=[]
    for way_1 in way_type:
        percentage=serial_number/len(way_type)*100
        if int(percentage)%10==0 and not(int(percentage) in percentages):
            percentages.append(int(percentage))
            print("%.2f" % int(percentage)+"%")
        serial_number=serial_number+1
        if only_restrictions!=[]:
            for restriction in only_restrictions:
                if restriction[0]==int(way_1.AttValue('Name')):
                    has_rest=True
                    break
                else:
                    has_rest=False
        else:
            has_rest=False
        linkpolypts_1=way_1.LinkPolyPts.GetAll()
        coord_x_1=linkpolypts_1[-1].AttValue('X')
        coord_y_1=linkpolypts_1[-1].AttValue('Y')
        way_1_start=[linkpolypts_1[0].AttValue('X'), linkpolypts_1[0].AttValue('Y')]
```

```
    for way_2 in way_type:

        if way_2.AttValue('Length2D')<=5 or way_1.AttValue('Length2D')<=5:

            dist=3

        else:

            dist=5

        if [way_1.AttValue('Name'), way_2.AttValue('Name')] in way_2_list or [way_2.
AttValue('Name'), way_1.AttValue('Name')] in way_2_list:

            pass

        else:

            continue

        if has_rest and not([int(way_1.AttValue('Name')),int(way_2.AttValue('Name'))] in
only_restrictions):

            continue

        for linkpoly_2_index in range(len(way_2.LinkPolyPts.GetAll())):

            linkpoly_2=way_2.LinkPolyPts.GetAll()[linkpoly_2_index]

            coord_x_2=linkpoly_2.AttValue('X')
            coord_y_2=linkpoly_2.AttValue('Y')

            if linkpoly_2_index!=len(way_2.LinkPolyPts.GetAll())-1 and way_1!=way_2 and
int(way_1.AttValue('Name'))!=-int(way_2.AttValue('Name')) and not([int(way_1.AttValue('
Name')),int(way_2.AttValue('Name'))] in no_turn_restrictions) and math.dist([coord_x_1,
coord_y_1], [coord_x_2, coord_y_2])<dist:

                polypoints=way_2.LinkPolyPts.GetAll()

                points_for_link=polypoints[0:linkpoly_2_index+1]

                coords_for_link=[]

                for point in range(len(points_for_link)):

                    coords_for_link.append([points_for_link[point].AttValue('X'),
points_for_link[point].AttValue('Y')])

                if linkpoly_2_index!=0:

                    Links.append(
                        Vissim.Net.Links.AddLink(0, str(sg.LineString(coords_for_link)),
[3.5])
                        )

                lanes=min(way_1.AttValue('NumLanes'), way_2.AttValue('NumLanes'))

                if lanes==1:

                    lane_connection_1=1
                    lane_connection_2=1

                else:

                    lane_connection_1=way_1.AttValue('NumLanes')-lanes+1
                    lane_connection_2=way_2.AttValue('NumLanes')-lanes+1

                if linkpoly_2_index!=0 and linkpoly_2_index!=len(way_2.LinkPolyPts.GetAll
())-2:

                    connect_pos=Links[-1].AttValue('Length2D')+5
```

```
                elif linkpoly_2_index!=0:

                    connect_pos=Links[-1].AttValue('Length2D')+1

                else:

                    if way_2.AttValue('Length2D')<3:

                        connect_pos=1

                    elif way_2.AttValue('Length2D')<5:

                        connect_pos=2

                    else:

                        connect_pos=5

                Vissim.Net.Links.AddConnector(0,way_1.Lanes.ItemByKey(lane_connection_1),
way_1.AttValue('Length2D'), way_2.Lanes.ItemByKey(lane_connection_2), connect_pos, lanes,
'LINESTRING EMPTY')

                if linkpoly_2_index!=0:

                    Vissim.Net.Links.RemoveLink(Links[-1])

                break

            if linkpoly_2_index!=len(way_2.LinkPolyPts.GetAll())-1 and linkpoly_2_index!=
len(way_2.LinkPolyPts.GetAll())-2 and linkpoly_2_index!=1 and linkpoly_2_index!=0 and
way_1!=way_2 and int(way_1.AttValue('Name'))!=-int(way_2.AttValue('Name')) and not([int(
way_2.AttValue('Name')),int(way_1.AttValue('Name'))] in no_turn_restrictions) and math.
dist(way_1_start, [coord_x_2, coord_y_2])<dist:

                polypoints=way_2.LinkPolyPts.GetAll()

                points_for_link=polypoints[0:linkpoly_2_index+1]

                coords_for_link=[]

                for point in range(len(points_for_link)):

                    coords_for_link.append([points_for_link[point].AttValue('X'),
points_for_link[point].AttValue('Y')])

                if linkpoly_2_index!=1:

                    Links.append(
                        Vissim.Net.Links.AddLink(0, str(sg.LineString(coords_for_link)),
[3.5])
                        )

                lanes=min(way_1.AttValue('NumLanes'), way_2.AttValue('NumLanes'))

                if lanes==1:

                    lane_connection_1=1
                    lane_connection_2=1

                else:

                    lane_connection_1=way_1.AttValue('NumLanes')-lanes+1
                    lane_connection_2=way_2.AttValue('NumLanes')-lanes+1

                connect_pos=Links[-1].AttValue('Length2D')-3

                Vissim.Net.Links.AddConnector(0, way_2.Lanes.ItemByKey(lane_connection_2),
 connect_pos, way_1.Lanes.ItemByKey(lane_connection_1), 3, lanes, 'LINESTRING EMPTY')
```

```
                    if linkpoly_2_index!=0:
                        Vissim.Net.Links.RemoveLink(Links[-1])
                    break
Create_connectors(cycleways)
print('100.0%')
Create_connectors(carways)
print('100.0%')
Vissim.SaveNetAs(os.path.join(os.getcwd(), osm_file+".inpx"))
```