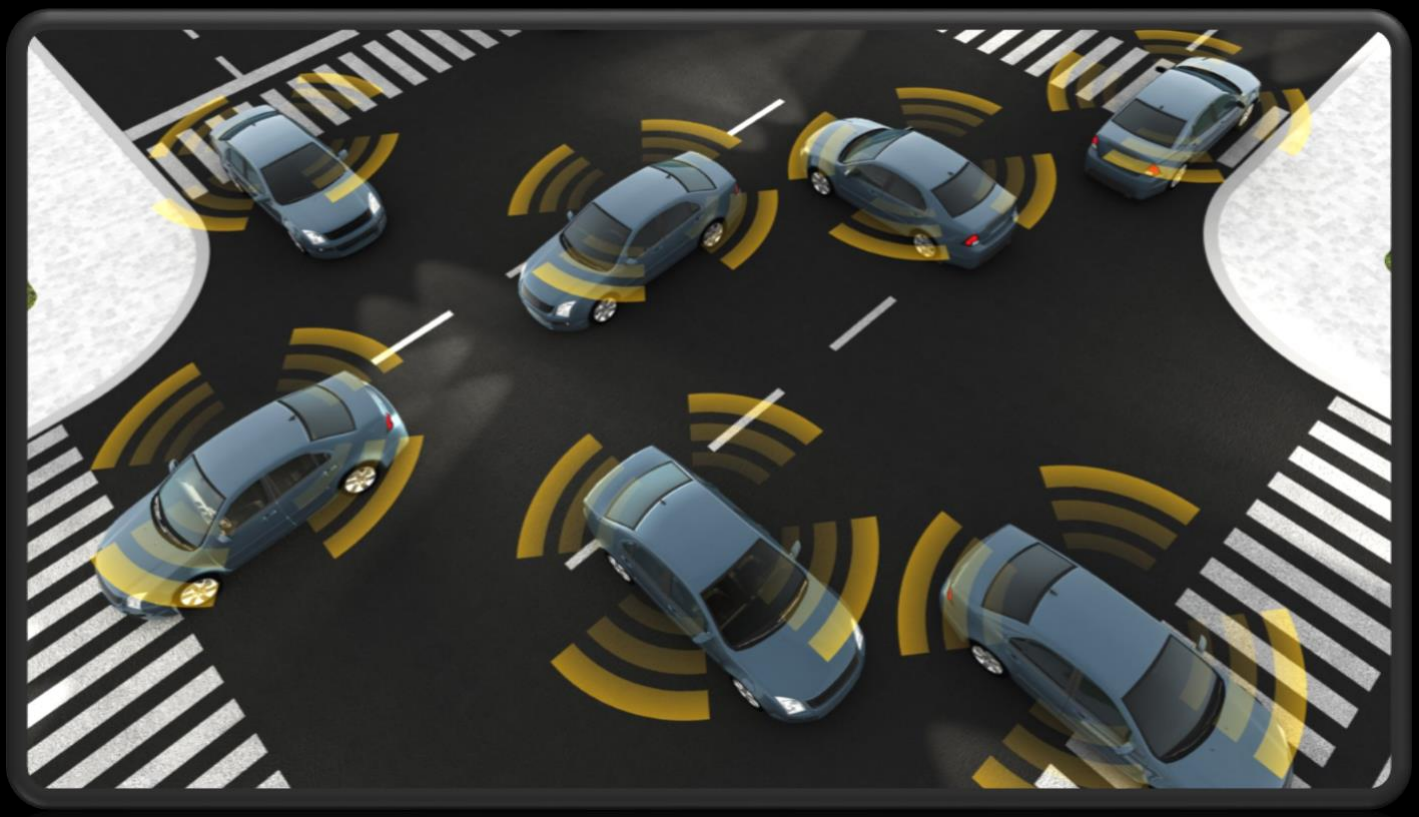

DRIVE:
Digital twin for self-dRiving Intelligent VEhicles

Version 1.0

User manual to accompany DRIVE simulation Framework

August, 5th 2020

Table of Contents

ioannis.mavromatis@toshiba-bril.com

1 Introduction

Digital twin for self-dRiving Intelligent Vehicles (DRIVE) was built to tackle shortcomings of traditional vehicular network simulators and merge the world of Cooperative Intelligent Transportation Systems (C-ITSs) and Artificial Intelligence. The defining feature of DRIVE is a unique architecture allowing for submission of sequential queries to dynamically interact with Intelligent Agents, sharing the "state of the world" and applying decision policies.

DRIVE provides a framework that users can develop, train and optimise Machine Learning-based C-ITS solutions for realistic vehicular scenarios. It can also be used for more traditional communication-related investigation, such as basestation placement optimisation, switch-on-switch-off approaches on the deployed basestations, heterogeneous resource allocation, efficient content dissemination and fetching, Quality-of-Service (QoS) centric optimisation as perceived from the system-level perspective, etc. All the above problems can be tackled from both the indoor user, or the Vehicle-to-Everything (V2X) perspective, or a combination of both.

1.1 DRIVE Architecture

DRIVE is a framework where all decision processes, routes, and interactions are determined at runtime. It can enhance the simulation of large-scale C-ITS city scenarios, where all traffic participants, and the communication infrastructure interact through various means of communication links. The framework is written in MATLAB and is open-sourced to enable the design of extensions by other users. Being designed in MATLAB, can be very easily linked with the existing optimisation and Machine-learning toolboxes provided by MathWorks. In addition, MATLAB provides integration with all common programming languages (Python, C++, etc.), so libraries written in them can be easily utilized within our framework. For example, this link describes how Python libraries can be called from within MATLAB. Our framework is flexible enough to support such features.

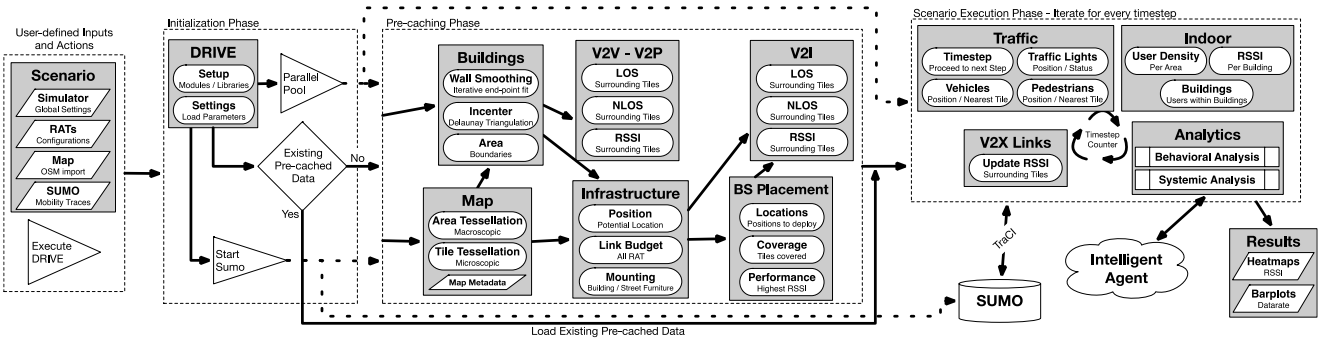

Figure 1 - A high-level framework design of DRIVE, with the steps followed during the initialization, pre-caching, and scenario execution phases. The user inputs several parameters via configuration files, and the framework handles the different interactions later. A pre-caching phase can speed up the consecutive executions, while interactions with mobility traces and intelligent agents can introduce new avenues of realistic C-ITS evaluation.

DRIVE is designed with a two-phase architecture in mind. Initially, it introduces a pre-caching step that can significantly enhance the user experience. During this phase, static information such as building positions, potential basestation placement locations, and communication interactions are calculated on the first run and stored for future usage. DRIVE handles all the file storing/loading to ensure that the execution time is minimised for sequential runs.

Later, a real-time "Scenario Execution Phase" is initiated. During this phase, DRIVE interacts bidirectionally with SUMO traffic generator and the pre-processed map files. A end-user is capable of implementing agents that interact with the environment and device optimal policies. A high-level diagram of the different steps and functionality implemented can be found in Figure 1. Also, more information about our developed framework, and a comparison with existing frameworks in the literature can be found in our accompanying publication (link).

ioannis.mavromatis@toshiba-bril.com

1.2 Key features

DRIVE is designed in a way to reduce the computational complexity of the existing simulation frameworks, still providing a very realistic representation of the real-world. Some of its **key features** are:

- Supports different communication planes in a heterogeneous fashion. It also provides an easy way of adding them in a scenario and modifying their configuration during the simulation time.
- Highly vectorised and parallelised environment, minimising the execution time by utilising several optimisation capabilities from MATLAB.
- Imports buildings and foliage outlines from OpenStreetMap [1] for a realistic city environment.
- Imports the positions of the street furniture either from OpenStreetMap [1] or from SUMO traffic generator [2]. These are later used as potential basestation positions within a scenario.
- Provides a bidirectional connection with SUMO traffic generator [2], via Traffic Control Interface (TraCI) [3];
    - Easily imports vehicular and pedestrian mobility traces, generated via SUMO traffic generator.
    - Modify and manipulate the mobility traces on-the-fly based on decisions taken either by the end-user or by a decision-making algorithm.
- Provides a way of generating indoor users for all the given buildings on the map. These users contribute to the overall network load.
- Tools for visualising the mobility traces and the network performance on the imported city maps are already provide to the end-user.
- Supports different large-scale signal variation models (path-loss and shadowing) for both LOS and NLOS scenarios; more models can be easily added in the framework.
- Easily extendable to support maps and vehicular traces from different sources.
- MATLAB implementation; **Open-source code, free and openly distributed.**

In the rest of the document, the core functionality of DRIVE framework will be described, explaining the different parameters that an end-user can modify to fine-tune a scenario. We will also refer to the key functions developed, to enable the described functionality.

ioannis.mavromatis@toshiba-bril.com

2 Getting Started

2.1 Installation and Supported Operating Systems

The installation of DRIVE is straightforward, and a user can set it up in a few minutes. The steps to be followed are:

1) Install SUMO traffic generator on the host operating system. SUMO can run on all OSs (Windows, Linux, macOS). Instruction on how to install SUMO can be found under this link.
2) Setup *SUMO_HOME* environmental variable in the host OS. Instructions for that can be found under this link.
3) Download or clone TraCI4Matlab from the official GitHub repository and add the directory in the MATLAB path (instruction for that under this link).
4) Add TraCI4Matlab additional dependencies to the MATLAB's *static JAVA path*. To do so:
   - Create a text file containing the path to *traci4matlab.jar*, e.g.:
     TRACI4MATLAB_HOME/traci4matlab.jar
   - *TRACI4MATLAB_HOME* is the path to the root folder of TraCI4Matlab obtained in the previous step.
   - Save this file as *"javaclasspath.txt"* in the preferences directory of MATLAB. You can identify this folder running ***"prefdir"*** command inside MATLAB.
   - Restart MATLAB, so the new static path takes effect.
5) Download or clone DRIVE from the official GitHub Repository.
6) Compile the required external libraries as MEX files. At the moment one external library needs to be compiled, i.e., "Segment Intersection Toolbox".
   - To do that, from the root directory of DRIVE, navigate inside *"./externalToolbox/segment_intersection/"* and run the following commands on MATLAB's command window.

```
mex -v segment_intersect_test.c
mex -v segments_intersect_test.c
mex -v segments_intersect_test_vector.c
```

7) DRIVE framework is ready to be used!

For convenience, all the external functions required for DRIVE come inside the *"./externalToolbox/"* directory. In Section 5, there is a complete list of the external functions used, their licences and the links to the original content. DRIVE is organised in "toolboxes" folders. A function adding all the required toolboxes in the MATLAB path is provided within the framework. This function is called during the initialisation time of the framework.

DRIVE can work on any platform running a recent version of MATLAB. It has been tested with MATLAB 2019a and 2018b, on macOS, Ubuntu and Windows environments without any problems. It has also been tested with SUMO v1.2.0 and v1.6.0, but it is expected to run with any SUMO version >1.0.0.

> ***NOTE:*** The TraCI4Matlab version found on Mathworks seems to be incompatible with some OSs, so we highly advice the end-users to download the GitHub version.

ioannis.mavromatis@toshiba-bril.com

2.2 Folder Structure and Configuration Files

DRIVE's code is structured as a **collection of different toolboxes**. The functions found in there are called during the execution phase. More specifically we have:

- **animate:** Functions to print the different maps and interpolate/animate the vehicles and pedestrians on top of them.
- **basestationToolbox:** A collection of functions that find the potential positions to place a basetation on a map, and all the basestation placement algorithms provided by default with DRIVE.
- **externalToolbox:** All the external libraries required by DRIVE (described in Section 5).
- **loadFiles:** Functions for loading the pre-processed SUMO and map files.
- **mapToolbox:** Functions to parse a given map, concatenate the building polygons, calculate the intersections with buildings, the user areas and map tiles.
- **modeToolbox:** The three operation modes that currently exist in DRIVE.
- **parsingToolbox:** Functions to parse the given SUMO and map files.
- **polygonToobox:** Functions to manipulate the polygons (buildings) on a map.
- **ratToolbox:** All the required functions to calculate the communication interactions between vehicles, pedestrians and the infrastructure network.
- **sumoToobox:** Functions to interact with SUMO.
- **tools:** Various tools supporting all the above functions, and test scripts to ensure the smooth operation of the framework.

DRIVE also requires a number of configuration files to be provided from the end-user, these being:

- ***simSettings.m:*** The main configuration file for DRIVE. It contains all the necessary information to initialise the framework, SUMO and the map files.
- ***./ratToolbox/available/XXXXX.m:*** The configuration files for the communication planes.
- ***./mobilityFiles/sumoFiles/SCENARIOS:*** Folder that contains all the scenario subdirectories. Each scenario should be a separate folder.

All the above files and directories are required from DRIVE when executed. In the following sections, we will describe in more detail each file and the different configuration parameters provided.

2.3 Running the Simulator

Before the execution, the user should download a map from OpenStreetMaps and generate the SUMO mobility traces for this map later. All these files should be saved under the directory *"./mobilityFiles/sumoFiles/"* inside a new folder. In this manual, we will not go into further details about how to download a map and generate the mobility traces. Detailed steps about that, can be found under DRIVE's Wiki page.

Having the above files, the user should modify the settings configuration file accordingly. The configuration file is call ***simSettings.m*** and can be found in the root directory of the framework. The main modifications are the following:

- `SIMULATOR.sumoPath` should point to the *SUMO_HOME* path, as configured during SUMO's installation.
- `SIMULATOR.scenario` should show the required scenario function.
- `SIMULATOR.parallelRun` can be modified to choose parallel or serial execution of the framework.
- `sumo.routeFile` should point to ***map.sumocfg*** SUMO configuration file generated before.
- `map.file` should point to the OpenStreetMap XML downloaded from the OSM's website.

ioannis.mavromatis@toshiba-bril.com

The rest of the configuration parameters can be changed accordingly to the user's requirements and they will be described in more detail in the remaining sections.

Having the above, the user can execute DRIVE, calling ***runSimulator.m*** from the root directory. DRIVE will **automatically load** all the required toolboxes in the MATLAB path (***setupSimulator.m***), load the scenario configuration parameters (***simSettings.m***), the necessary SUMO and OSM files (from *"./mobilityFiles/sumoFiles/SCENARIO_FOLDER"*), the communication plane configuration files (from *"./ratToolbox/available/COMM_TECH.m"*) and the corresponding propagation models (from *"./ratToolbox/pathLossModels/PATHLOSS.m"*).

ioannis.mavromatis@toshiba-bril.com

3 DRIVE in-Depth

3.1 Pre-caching of Scenario Information

DRIVE can handle the loading and processing of the scenario files. The files loaded and processed are the map and mobility trace files provided. Furthermore, DRIVE calculates the potential basestation positions, and the LOS and NLOS communication links based on the user input. A parsed example map file can be found in Figure 2. The core function responsible for loading or saving the map information is *loadMap(map,sumo)*. The different OSM and SUMO files are parsed using *parseOSM(map)* and *parseSUMOMap(map,sumo)* respectively.

- ***First run:*** When the simulator runs for the first time all the necessary files are parsed and the scenario information is calculated. The end-user is asked whether this information will be saves or not. The files are saved under *"./mobilityFiles/preprocessedFiles/"* and a new folder is created with the given scenario name, this being the folder name hosting the source SUMO and OSM files. If an answer is not provided, by default, the files are saved after 10s.
- ***Sequential runs:*** If an already pre-processed scenario exists, DRIVE asks the user whether these files will be loaded or not. If the user provides a positive answer, then the files are loaded from the pre-processed directory. On the other hand, the files are processed again and overwritten on the existing ones. As before, by default DRIVE loads the files are 10s, if an answer is not provided.

Figure 2 - An example of a map imported in DRIVE.

The above functionality can be controlled by the given settings file as well. Modifying the `SIMULATOR.load` parameter a user can:

- Choose to process the scenario files on every execution (`SIMULATOR.load = 0;`).
- Be asked whether to load the pre-processed files (`SIMULATOR.load = 1;`).
- By default, load all the files found in the pre-processed directory (`SIMULATOR.load = 2;`).

3.2 Improving the Execution Time

During the development of DRIVE framework, particular attention was given on parallelising and vectorising several sections of the code to speed-up the execution time. The parallel execution of several functions can be enabled by using "`SIMULATOR.parallelRun = 1;`", found in the ***simSettings.m*** file. If the parallel execution is enabled, the end-user can later specify the number of cores to be allocated for the simulation. For e.g., "`SIMULATOR.parallelWorkers = 8;`" will utilise 8 CPU cores. This can significantly decrease the execution time, especially during the initialisation phase of the scenario where the map is imported, and the LOS/NLOS links are calculated.

ioannis.mavromatis@toshiba-bril.com

3.3 Buildings and Road Polygon Manipulation to Reduce the Execution Time

Several map simplification strategies take place within DRIVE. At first, the building and foliage polygons are merged, and their inner holes are removed (Figure 3). The functions responsible for that are the *mergePolygons(polygons)* and *removeHoles(polygons)*. By that, we can achieve a very realistic performance investigation from the system-level perspective, minimising the computational complexity of the excessive number of polygons.

This functionality can be fine-tuned by the end-user modifying the `map.simplificationTolerance` parameter, used by *simplifyPolygons(polygons,map)* function. This value defines the tolerance threshold (in meters) for simplifying the building outlines. For example, "`map.simplificationTolerance = 10;`" will remove wall sections that are shorter than 10 meters, smoothening the building layout. Removing the excessive corners, and thus the line segments to be tested against all the potential communication links, can significantly reduce the time required to calculate all the LOS and NLOS links for the entire city. Each polygon calculated after the union polygon operation is considered a ***"building block"*** and will be later used for generating the indoor users for each scenario.

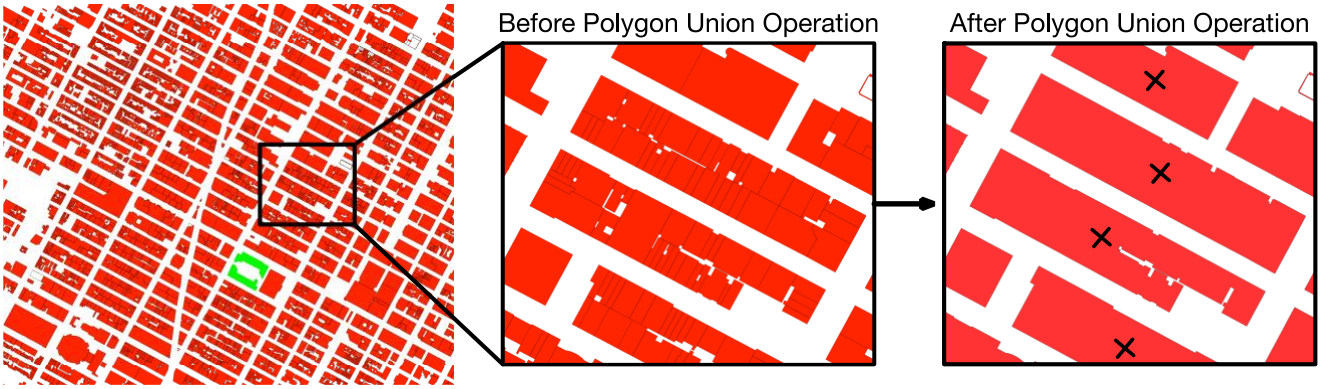

Figure 3 - Example of the union polygon operation, and the simplifications introduced on the map.

Similarly, the road polygons are manipulated as well. OpenStreetMap and SUMO define the roads with different data structures in the corresponding files. DRIVE provides functions to handle both and generates a ***"road polygon"***, later used for various different functions (e.g., the basestation placement, calculating the network performance, etc.). The OSM maps, are parsed using the external OpenStreetMap Functions package, available under this link. A version has been included in our framework and modified accordingly to support different types of buildings and foliage polygons. Later, the "OSM roads" are handled by *roadsPolygon(roadsLine).* This function parses the road lines, given from the OSM file, and generates the road polygons with respect to the road width and type. The "SUMO roads" are directly parsed from SUMO, using TraCI4Matlab framework.

3.4 Map and Tile Tessellation

A tessellation approach for the entire map is followed in a two-way fashion. At first, the user can define the size of the map areas, changing `map.area` parameter in the configuration file (given in meters). This discretises the map into different large areas with equal size, that can be later used to either apply different policies or evaluate different parameters. The areas can have either a hexagonal or a squared shape (Figure 4). This is controlled by the user, modifying `map.areaShape` parameter ("`map.areaShape = 1;`" for hexagon and "`map.areaShape = 2;`" for square). In conjunction with the map area, the user can experiment with different area surfaces. For example:

- "`map.areaShape = 2;`" and "`map.area = 400;`" will produce square areas with sides of length 400m.

ioannis.mavromatis@toshiba-bril.com

- `"map.areaShape = 1;"` and `"map.area = 200;"` will produce hexagon areas with short diagonals of length 400m.

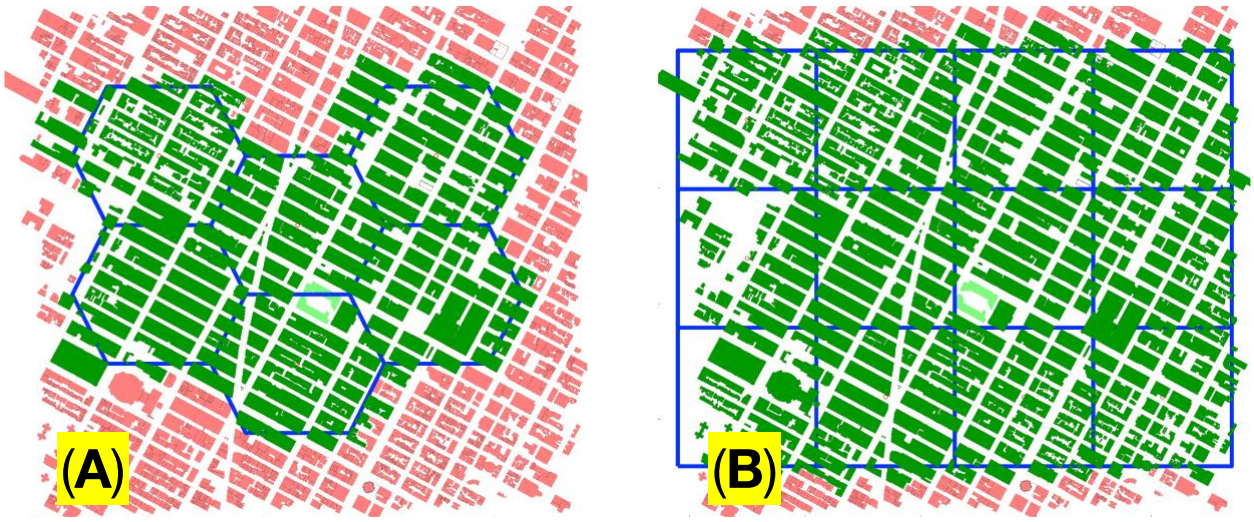

Figure 4 - Examples of hexagonal (A) and squared (B) areas on the city.

Also, the map is tessellated in tiles as well by modifying `map.tileSize` parameter (Figure 5). A tile is considered as a unified area on the map that has exactly the same properties on its entire surface. For DRIVE, all the calculations take place using the incentres of the given tiles to speed up the performance (Figure 3). Using a relatively small tile size <5m (e.g., (`"map.tileSize = 4;"`), a very realistic result can be achieved significantly reducing the computational complexity. More information about this approach can be found in [5].

The core function responsible for the map tessellation is the *mapSmallerAreas(outputMap,map).* Functions *smallerAreasHexagons(outputMap,tile,largeTiles)* and *smallerAreasSquares(output,tile,largeTiles)* manage both the tile and map tessellation. Finally, the *tilesWithinAreas(outputMap)* function finds the tiles that lay within each map area calculated before (either hexagonal or squared). Some of them may not belong to any area and they can be excluded for the rest of the simulation.

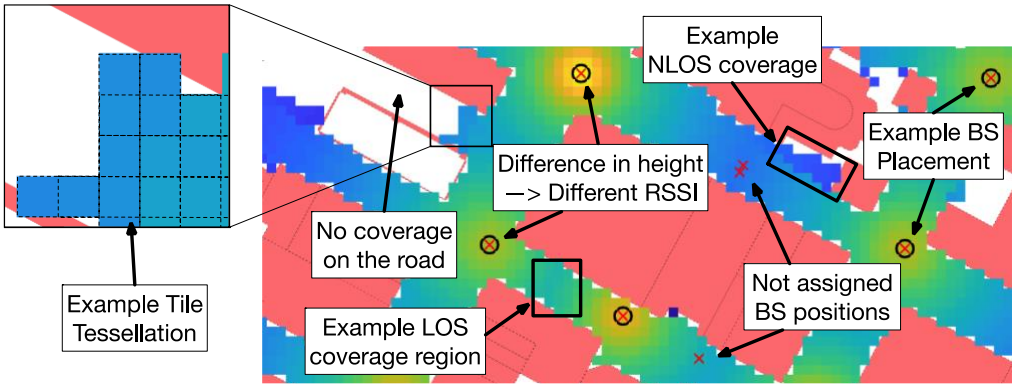

Figure 5 - Example tile tessellation and the operational output. The heatmap shows the calculated RSSI per tile and the effect of the building blockages on the perceived signal strength.

3.5 Generating Indoor User Traffic

DRIVE provides the functionality to introduce a number of users per ***building block***. The maximum number of users for a given area is specified in the configuration file using the variable `map.maxUsersPerBuilding`. If zero is given, the indoor users will not be taken into account. On top of that, the user can define the granularity of the indoor model using

ioannis.mavromatis@toshiba-bril.com

`map.densitySimplification`. This variable defines the area of squared-tiles that the above-mentioned number of users will be applied. For example, if "`map.maxUsersPerBuilding = 50`" and "`map.densitySimplification = 100`", then each squared tile with surface area of 100m$_2$ can have up to 50 users. Based on the number of users per surface area, the number of users per building block is later calculated. The user density model used in DRIVE is described in [4] and is based on real-world experiments. The described functionality can be found in function *usersPerBuilding(outputMap,timeStep,randomValues,coordinates,initialX,initialY,interpX,interpY,map).* The values used for the log-normal distribution of the downlink for urban environments are based in [4] as well.

3.6 3D Environment

DRIVE introduces a 3D city environment. All building polygons inherit their height from OpenStreetMap metadata (Figure 6). If the building height is missing, a value is assigned at random from a range controlled by the end-user. In our framework, there is also a provision to control the randomness with seeds to ensure the reproducibility of the results. The function responsible for parsing the building height is *parse_osm(osm_xml)* and the missing building information are handled by *add3rdDimension(buildings)*. A 2D or a 3D environment can be later utilised during the simulation time, in order to calculate the distance between vehicles or from the vehicles to the basestations (2D and 3D respectively).

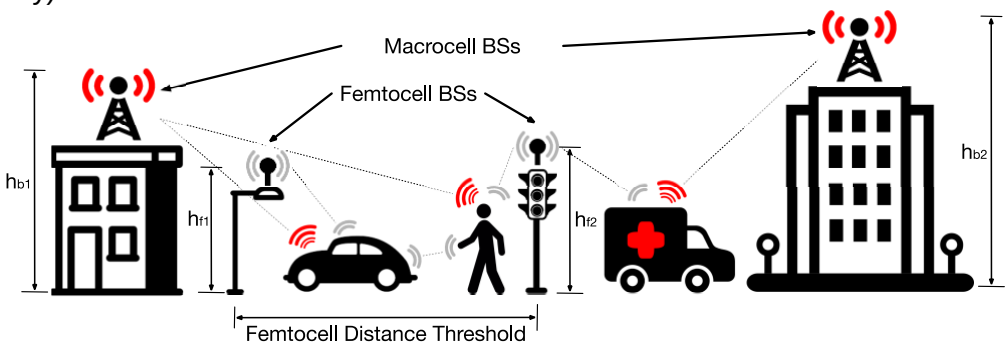

Figure 6 - High-level system representation. The different types of vehicles and the pedestrians, equipped with a number of communication devices, communicate between them and with the basestation placed on top of buildings or street furniture.

3.7 Communication Planes

DRIVE can read configuration files provided by the end-user and automatically generate the different communication planes within each scenario. The configuration files for each plane should be added within the *"./ratToolbox/available/"* directory. On execution, DRIVE iterates for all and adds them in the scenario. A user can add as many configuration files, i.e., communication planes, as required for each scenario.

DRIVE supports two "cell-like" types of communication planes. i.e., macrocell and femtocell. The main difference is the mounting point of the basestations. A user can modify these files accordingly to fine-tune a simulation scenario. The information required are usually the typical information found in all path loss models (e.g., transmission power - `txPower`, carrier frequency - `cFrequency`, noise floor – `noiseFloor` - and noise figure - `noiseFigure`, beamwidth – `beamwidthAngle`, etc.).

Later, some specialised controls are introduced, associated with each communication plane. More specifically, the macrocell configuration file requires the parameter `macroDensity`, that defines the number of basestations per km$_2$. Furthermore, macrocell plane basestations inherit their height from the buildings they are mounted on. On the other hand, the femtocell ones get their height from a uniform random distribution and based on the given `height` parameter. Both planes implement a max transmission distance policy (`maxTXDistance` for macrocell, and `femtoDistanceThreshold` for

ioannis.mavromatis@toshiba-bril.com

the femtocell planes). This means that for any tile being outside of the given “coverage region”, it is assumed that no communication link can be established. An example can be seen in Figure 7. Vehicle C, being outside of the coverage region of Vehicle A, will always be excluded of the communication link calculations to speed up the execution time.

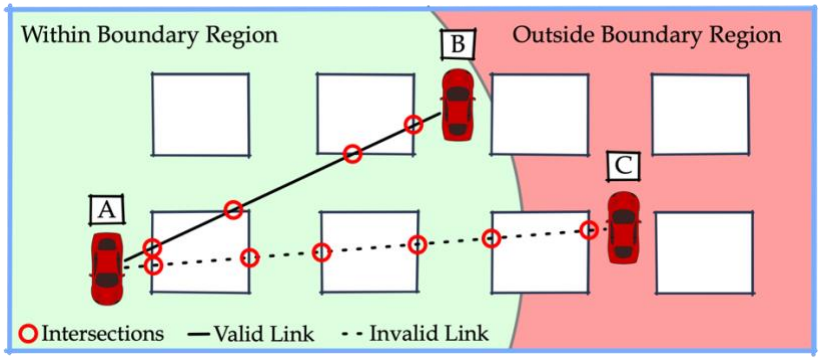

Figure 7 - Coverage region and intersection with building blocks.

Finally, the different datarates, achieved from each Modulation & Coding Scheme can be provided. Associated with a sensitivity threshold (`sensitivityLevels`) and using the calculated signal strength from the path loss models, they can increase the realism of the experimental evaluation.

Within our GitHub repository, we provide two examples of the two different categories, i.e., “lte” referring to the macro-cell one, and “mmWaves” referring to the femto-cell approach. These files can act as templates for the end-user and can be modified accordingly to introduce the required functionality.

ioannis.mavromatis@toshiba-bril.com


```
%% LTE links settings
ratName = 'lte';
ratType = 'macro';
pathLossModel = 'urbanMacro';
cFrequency = 2600000000; % carrier frequency given in Hz
txPower = 15; % given in dBm
noiseFloor = -174; % noise floor
noiseFigure = -6; % noise figure
cBandwidth=20000000; % channel bandwidth in Hz
beamwidthAngle = 120; % default beamwidth for mmWaves - may be changed during the simulation time
mimo = 0; % give 0 for SISO channel and 1 for MIMO 4x4
bsGain = 18; % the antenna gain of the basestation
mobileGain = 0;  % the antenna gain of the mobile station
pathLossExponentLOS = 2.4;
macroDensity = 0.5; % The number of basestations per square km.

% The maximum distance that the LTE signal can still be considered as
% received -- if a very big value is chosen, it will be adapted later given
% the mmWave link budget characteristics
maxTXDistance = 1000;

%% Modulation Schemes for LTE links
% 4 MCS with different data rates at 20MHz channel bandwidth and a MIMO and SISO configuration
% No MIMO                           % MIMO 4x4
% 256-QAM 0.9258 - 97.896Mbps       % 256-QAM 0.9258 - 391.584Mbps
% 64-QAM 0.8525 - 75.376Mbps        % 64-QAM 0.8525 - 301.504Mbps
% 16-QAM 0.6016 - 30.576Mbps        % 16-QAM 0.6016 - 122.304Mbps
% QPSK 0.4385 - 15.84Mbps           % QPSK 0.4385 - 63.36Mbps

if mimo == 0
    dataRate = [ 97896000 75376000 30576000 15840000 ];
else
    dataRate = [ 391584000 301504000 122304000 63360000 ];
end
sensitivityLevels = [ -72 -77 -83 -94 ];
```

```
%% MmWave links settings
ratName = 'mmWaves';
ratType = 'femto';
pathLossModel = 'freespace';
cFrequency = 60000000000; % carrier frequency given in Hz
txPower = 20; % given in dBm
noiseFloor = -174; % noise floor
noiseFigure = -6; % noise figure
cBandwidth=2160000000; % channel bandwidth in Hz
beamwidthAngle = 15; % default beamwidth for mmWaves - may be changed during the simulation time
height = [ 5 15 ]; % The range of heights that an mmWave basestation might be placed
mimo = 0;

% The maximum distance that the mmWave signal can still be considered as
% received -- if a very big value is chosen, it will be adapted later given
% the mmWave link budget characteristics
maxTXDistance = 200;
pathLossExponentLOS = 2.66;
pathLossExponentNLOS = 7.17;
femtoDistanceThreshold = 100; % The maximum distance separation between two femtocell basestations

%% Modulation Schemes for mmWave links
% 7 MCS with different data rates - the 8th is when not transmitting
% The SNR values based on the sensitivity of MCSs, as given from IEEE
% 802.11ad standard
% 64-QAM 13/16 - 6756.75Mbps          % SQPSK 5/8 - 866.25Mbps
% 64-QAM 5/8 - 5197.5Mbps             % pi/2-BPSK 1/2 - 385Mbps
% 16-QAM 3/4 - 4158Mbps               % pi/2-BPSK 1/2 27.5 Mbps
% QPSK 1/2 - 1386Mbps

dataRate = [ 6756750000 5197500000 4158000000 1386000000 866250000 385000000 27500000 ];
sensitivityLevels = [ -47 -51 -54 -63 -64 -68 -78 ];
```


ioannis.mavromatis@toshiba-bril.com

3.8 Potential Basestation Positions

As described in Section 3.7, DRIVE supports two "cell-like" types of communication planes. i.e., macrocell and femtocell, with the main difference being the mounting point of the basestations. For the macro-cell approach, the basestation is positioned on top of the buildings. More specifically, it is being placed at the incentre of a building block (Figure 3). The building incentres are calculated using *buildingCentres(outputMap)* function. This function finds the incentres using Delaunay Triangulation from computational geometry on the convex building block polygons generated before. Furthermore, this function finds the map areas that each building belongs to (Figure 4). All building with a least one edge within the boundaries of a given area, are added in an array and can be later used to calculate area-specific LOS/NLOS communication links or device policies from intelligent agents.

The femto-cell basestations are considered to be mounted on top of the street furniture (Figure 6). The positions of the traffic lights are imported from the map and SUMO files (i.e., using the functions *sumoFemtoPositions(outputMap,BS,map,ratName)* and *osmFemtoPositions(outputMap,BS,map,ratName)).* Importing the positions of the traffic lights from the map file, we create a set of potential positions for basestation deployment. OSM does not provide information about the lampposts on a city. Within DRIVE, having the road polygon edge information generated before, we later calculate the potential lamppost position for all the map using *addLamppostPositions(distance,matrixPos,map)*. With this function a road is discretised in equal sections and lamppost are considered to be equally spaced, based on a given distance threshold defined in ***simSettings.m*** as *femtoDistanceThreshold*. More details about that can be found in [5]. Finally, two examples for above can be found in Figure 8. On the left, we see the potential basestation positions for a macrocell communication plane with `"macroDensity = 500;"`. On the right, we see the lamppost and traffic lights on a map, identified as the potential positions for a femtocell communication plane.

Figure 8 - A potential basestation placement for macrocells (left) and femtocells (right).

3.9 Simulation output

Different verbosity levels are provided by the simulation framework. The different levels are controlled with `VERBOSELEVEL` parameter in the ***simSettings.m*** file. If the user chooses `"VERBOSELEVEL = 1;"` or `"VERBOSELEVEL = 2"`, then the simulator plots several figures with the performance of each communication plane and their datarates (e.g., as in Figure 9). The verbosity level also dictates the output on the Command Window as well. For values greater or equal that 1, several text messages are printed showing the progress of the simulation and the time required for each step. The second verbose level plots several test figures as well, such as the different map areas and tiles. It can be used particularly when the end-user modifies the framework and extends its functionality to cross-validate several aspects of the designed functions. All the above can be disabled, by choosing `"VERBOSELEVEL = 0"`.

ioannis.mavromatis@toshiba-bril.com

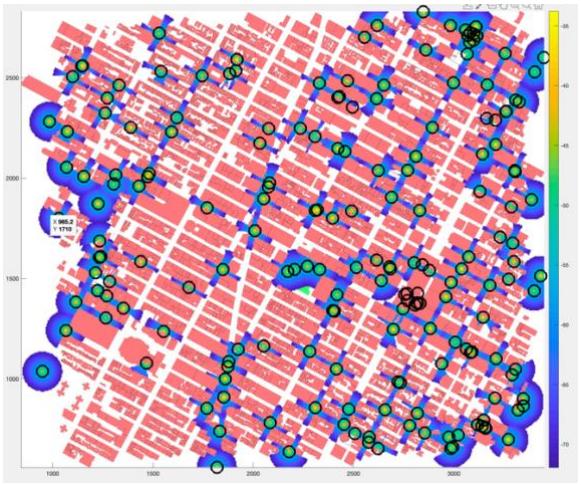

Figure 9 - A RSSI visualisation example for a given technology and a number of basestations.

3.10 Using SUMO and TraCI

The mobility traces used within DRIVE are generated using SUMO traffic generator. In order to utilise the SUMO mobility traces and maps, a link between SUMO and DRIVE should be established. For that, TraCI is used. TraCI allows the user to interact with SUMO, in a client-server fashion where MATLAB acts as the client, and SUMO as the server. In order to do so, TraCI4Matlab should be added in the MATLAB path at first and the additional Java dependencies to the MATLAB's static Java path, as described in the installation steps (Section 2.1). TraCI is responsible for running SUMO as a server, provides interfaces to modify the SUMO trace files in real time, and get help with getting the state of the environment on every simulation step. Furthermore, TraCI commands are used to progress the scenario (e.g., progress to the next timestep), and parse the various information provided (e.g., get the position of all the vehicles for a given time). More information about how to setup and use TraCI can be found under the link and a high-level diagram showing the interaction of a TraCI client (i.e., DRIVE framework) with the SUMO server can be seen in Figure 10.

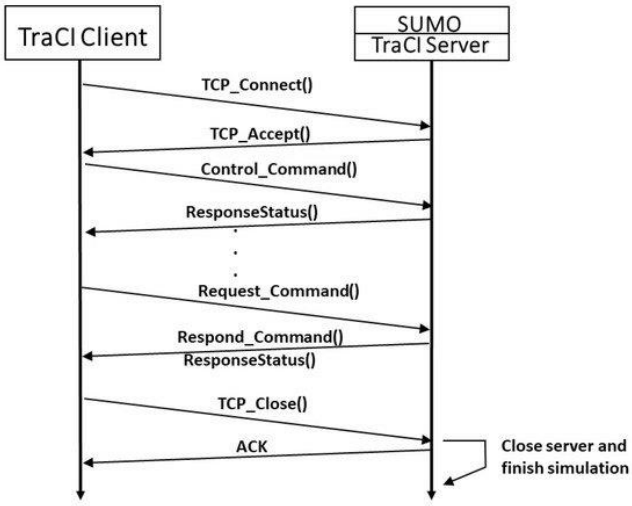

Figure 10 - Interactions between TraCI client (DRIVE) and SUMO server.

ioannis.mavromatis@toshiba-bril.com

4 Example scenarios

DRIVE can be used in various ways. Users can choose to run scenarios on just an OSM map (ideal for evaluating different communication solutions with simplistic user densities per city block), or work on more realistic C-ITS implementation (using a bidirectional interaction with SUMO traffic generator). To help the end-users design their own scenarios, DRIVE comes with three example use-cases, pre-configured and ready to be used and some example maps and SUMO mobility traces. All the use-cases (called as "modes" in DRIVE) can be found under *"./modeToolbox"* directory and they can be called modifying the ***simSettings.m*** file accordingly (`SIMULATOR.scenario`).

More specifically, the three example use-cases, demonstrating the different functionalities of DRIVE are the following:

- ***runOSM.m:*** A scenario based on just an OSM map, simulating indoor traffic and a macro-cell communication plane that calculates the average datarate for the given technology.
- ***runSUMO.m:*** A scenario with both indoor traffic and SUMO mobility traces, that changes the configuration of the given communication planes based on the user density.
- ***runV2V.m:*** A scenario that calculates the number of established links between vehicles and pedestrians, in a Vehicle-to-Vehicle and Pedestrian-to-Pedestrian fashion.

Finally, three maps and mobility traces are provided as the default DRIVE scenarios. Two of them are maps extracted from OpenStreetMap, with their corresponding mobility traces, generated using SUMO traffic generator. The third one is artificially generated map (generated using netgenerate SUMO tool). More specifically we have:

- ***City of Manhattan:*** ~9km$_2$ of buildings, foliage and roads, with vehicular (2 different types of vehicles) and pedestrian mobility traces generated.
- ***City of London:*** ~4km$_2$ of buildings, foliage and roads, with vehicular (2 types of vehicles) mobility traces generated.
- ***Smart Junction:*** A cross-junction (one lane per road), and with the corresponding vehicular (2 types of vehicles) mobility traces generated.

> ***NOTE:*** We encourage the users to test DRIVE's functionality with these scenarios and study the way we implemented them. They can later, in no time generate new scenarios to experiment with, and tailor them based on their needs. If common practises are followed when generating a scenario (as described in our Wiki page), we do not expect to see any difficulties being parsed from DRIVE. Adding a new scenario is just a matter of creating a new folder under *"./mobilityFiles/sumoFiles",* and giving the correct route file inside ***simSettings.m***.

ioannis.mavromatis@toshiba-bril.com

5 DRIVE framework dependencies

DRIVE inherits some of its functionality from external dependencies. All these functions are publicly available either from MATLAB Central File Exchange, under https://uk.mathworks.com/matlabcentral/fileexchange/ or on GitHub repositories. These functions are copyrighted by their respective developers under a BSD licence. For convenience they are included in DRIVE framework, under *externalToolbox* directory, along with their corresponding licences:

- deg2utm © 2006 by Rafael Palacios:
  https://uk.mathworks.com/matlabcentral/fileexchange/10915-deg2utm.
- Line Simplification © 2010 by Wolfgang Schwanghart:
  https://uk.mathworks.com/matlabcentral/fileexchange/21132-line-simplification.
- Fast Line Segment Intersection © 2010 by U. Murat Erdem:
  https://uk.mathworks.com/matlabcentral/fileexchange/27205-fast-line-segment-intersection.
- INPOLY: A fast points-in-polygon test © 2018 by Darren Engwirda:
  https://uk.mathworks.com/matlabcentral/fileexchange/10391-inpoly-a-fast-points-in-polygon-test.
- MatGeom: Matlab Geometry Toolbox for 2D/3D Geometric Computing © 2019 by David Legland:
  https://github.com/mattools/matGeom.
- OpenStreetMap Functions © 2016 by Ioannis Filippidis:
  https://uk.mathworks.com/matlabcentral/fileexchange/35819-openstreetmap-functions.
- Lightspeed Matlab Toolbox © 2019 by Tom Minka:
  https://github.com/tminka/lightspeed
- xml2struct © 2012 by Wouter Falkena:
  https://uk.mathworks.com/matlabcentral/fileexchange/28518-xml2struct.
- Waitbar for Parfor © by Yun Pu:
  https://uk.mathworks.com/matlabcentral/fileexchange/71083-waitbar-for-parfor
- TraCI4Matlab © 2019 by Andres Acosta:
  https://uk.mathworks.com/matlabcentral/fileexchange/44805-traci4matlab.
- questdlg timer © 2011 by Az Nephi:
  https://uk.mathworks.com/matlabcentral/fileexchange/32977-questdlg-timer
- Segment Intersect Test Functions © 2012 by Francesco Montorsi:
  https://uk.mathworks.com/matlabcentral/fileexchange/35492-segments-intersection-test-functions

ioannis.mavromatis@toshiba-bril.com

6 Acknowledgements, Licence and Citing DRIVE

6.1 Acknowledgements

This work was funded in part by Toshiba Research Europe Ltd., in part by the Next-Generation Converged Digital Infrastructures (NG-CDI) project, supported by BT Group and EPSRC (EP/R004935/1), and in part by the CAVShield project (Innovate UK, no. 133898).

6.2 Licence and citation

DRIVE is freely available under the terms found in the LICENCE file in our GitHub repository (link). If our framework is used to draft a research manuscript or in any other scientific investigation, we ask the authors to refer to that as follows:

I. Mavromatis, R. J. Piechocki, M. Sooriyabandara, A. Parekh, *"DRIVE: A Digital Network Oracle for Cooperative Intelligent Transportation Systems,"* in Proc. of IEEE Symposium on Computers and Communications (ISCC), Rennes, France, Jul. 2020.

For convenience the above citation in LaTeX format:

```
@inproceedings{driveSimulator
   author={{Mavromatis}, I. and {Piechocki}, R. and {Sooriyabandara}, M. and {Parekh},
A.},
   booktitle={Proc. of IEEE Symposium on Computers and Communications (ISCC)},
   title={{DRIVE: A Digital Network Oracle for Cooperative Intelligent Transportation
Systems}},
   year={2020},
   month={jul},
   address={Rennes, France},
}
```


ioannis.mavromatis@toshiba-bril.com

7 References

1) M. Haklay, and P. Weber, “OpenStreetMap: User-Generated Street Maps”, *in IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, Dec. 2008.
2) P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y. P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic Traffic Simulation using SUMO", *in Proc. of IEEE Intelligent Transportation Systems Conference (ITSC) 2018*, Maui, Hawaii, United States, Nov. 2018.
3) Acosta, J. E. Espinosa, and J. Espinosa, “TraCI4Matlab: Enabling the Integration of the SUMO Road Traffic Simulator and Matlab Through a Software Re-Engineering Process”, *in Modeling Mobility with Open Data, Springer International Publishing*, pp. 155–170, 2015.
4) D. Lee, S. Zhou, X. Zhong, Z. Niu, X. Zhou and H. Zhang, "Spatial modeling of the traffic density in cellular networks," *in IEEE Wireless Communications*, vol. 21, no. 1, pp. 80-88, Feb. 2014.
5) I. Mavromatis, A. Tassi, R. J. Piechocki and A. Nix, "Efficient Millimeter-Wave Infrastructure Placement for City-Scale ITS," *in Proc. Of IEEE VTC-Spring 2019*, Kuala Lumpur, Malaysia, pp. 1-5, May 2019.

ioannis.mavromatis@toshiba-bril.com