Dynamic Systems and Simulation Laboratory

School of Production Engineering & Management
Technical University of Crete

TrafficFluid-Sim: Extending SUMO for Microscopic Simulation in Lane-Free Traffic

User Manual

by

Dimitrios Troullinos, Iason Chrysomallis, Ioannis Papamichail, and Markos Papageorgiou

INTERNAL REPORT No: 2025-27

Chania, Greece
May 6, 2025

TrafficFluid-Sim: Extending SUMO for Microscopic Simulation in Lane-Free Traffic

User Manual

Dimitrios Troullinos, Iason Chrysomallis, Ioannis Papamichail, and Markos Papageorgiou

Dynamic Systems andtr Simulation Laboratory, Technical University of Crete

{dtroullinos, ichrysomallis, ipapamichail, mpapageorgiou}@tuc.gr

Abstract

TrafficFluid-Sim is a microscopic simulation tool for *lane-free traffic environments* considering Connected and Automated Vehicles. It was developed within the frame of the research project *TrafficFluid*. In this manual, technical information for users is presented. More specifically, it includes: installation and setup instructions (both for Windows 11 and Linux users), instructions for setting up lane-free traffic scenarios, design guidelines for lane-free vehicle movement strategies, usage of the provided API and other relevant features of the simulator.

Contents

1 Overview **4**
- 1.1 Motivation . . . 4
- 1.2 How to cite . . . 5
- 1.3 Notes for Developers . . . 5

2 Setup in Windows 10/11 **6**
- 2.1 Environment Variables . . . 6
- 2.2 Visual Studio Setup . . . 12

3 Setup in Ubuntu Linux **14**

4 Setup in Other Linux distributions and MacOS **16**

5 Lane-Free Plugin and Basic Principles for Designing Lane-Free Vehicle Movement Strategies **16**
- 5.1 Code Structure . . . 16
- 5.2 API . . . 17
- 5.3 Information Retrieval and Vehicle Control . . . 18
  - 5.3.1 (Double) Double Integrator Model . . . 21
  - 5.3.2 Bicycle Model . . . 22
  - 5.3.3 Global Coordinates Control . . . 23

6 Basic Principles for Setting up Lane-Free Scenarios **24**
- 6.1 Setup for a Highway Scenario . . . 24
- 6.2 Lateral Boundaries in Lane-Free Traffic . . . 28
- 6.3 Flow Demand in Lane-Free Traffic . . . 30

7 Online Traffic Network Measurements **33**
- 7.1 Density Measurements . . . 33
- 7.2 Speed Measurements . . . 34
- 7.3 Flow Measurements with Detectors . . . 35

8 Available Metrics **37**
- 8.1 Online Simulation Metrics in GUI . . . 37
- 8.2 Post Simulation Metrics in Logfiles . . . 42

9 Store and Replay Scenarios **43**

10 Ring-Road **45**

11 On-Ramps and Off-Ramps **46**
- 11.1 Scenario Setup . . . 46
- 11.2 Controller/Lane-Free Plugin Setup . . . 47

12 Bidirectional Highways and Internal Boundary Control **48**
- 12.1 Scenario Setup . . . 49
- 12.2 Controller/Lane-Free Plugin Setup . . . 50

13 Roundabouts **51**

14 Intersection **53**
14.1 Opposite Direction Movement 53
14.2 Bicycles and Pedestrians 54

15 Platoons **56**

References **57**

1 Overview

The present user manual accompanies *TrafficFluid-Sim*, a *lane-free* microscopic simulator designed for research purposes. TrafficFluid-Sim is an *extension* of SUMO [1] that explicitly targets lane-free traffic environments. It was developed within the frame of *TrafficFluid* [8][1], an ERC Advanced Grant hosted at the Technical University of Crete.

With TrafficFluid-Sim, users can design from scratch and test lane-free vehicle movement strategies in C/C++, with an API that provides information about the traffic environment, effectively emulating vehicle-to-vehicle and vehicle-to-infrastructure communication. Each API function call provided in the code is accompanied with a concise description of its usage and functionality. The underlying movement dynamics of vehicles can be either the double integrator model for both the longitudinal (x) and lateral (y) axis, or a bicycle model that better captures the orientation of the vehicles as well. A variety of common traffic environments can be designed and simulated, such as: highways, custom on-ramp and off-ramp scenarios, bidirectional scenarios that can be also tied with emergent infrastructure-based applications such as Internal Boundary Control (see Section 12), roundabouts, intersections, and ring-roads.

1.1 Motivation

Commonly, extensions of SUMO rely on the Traffic Control Interface (TraCI) API.[2] TraCI is a very convenient tool that provides great flexibility. However, for the microscopic simulation purposes of lane-free traffic environments, its usage was inadequate primarily for the following two reasons:

(a) TraCI relies on socket communication, which imposes a significant time bottleneck for large-scale environments where each individual vehicle needs to request information and provide control input. Notably, Libsumo[3] could alternatively be used to address these time-related issues. However, its usage is still quite limited.

(b) SUMO is designed from the ground-up considering lane-based environments. Even with the use of the Sublane Model,[4] the structural limitation of lanes is quite present and would require ad-hoc solutions such as directly influencing the global position of the vehicle, thus rendering SUMO simply a tool for visualization.

Consequently, we opted to fork the open-source codebase of SUMO, and develop an extension that targets lane-free environments. We underline two primary aspects that signify the design of TrafficFluid-Sim when compared to the standard SUMO application. Firstly, we have modified the internal codebase of SUMO in order to properly model the 2-dimensional movement dynamics of vehicles, completely disregarding the typical lane-keeping behaviour that is rooted in the existing simulator. Then, given the emerging nature of the lane-free paradigm and for practical reasons, we provide the user with a code structure for the development of *lane-free* vehicle movement strategies, one that contains an API for direct communication with the main application. That is in contrast to the typical use of SUMO, where users can select and calibrate one of the available models for vehicle movement.

In Figure 1, we showcase a high-level overview of the application. Essentially, the available code structure needs to be compiled following the instructions given below, in Sections 2-4, depending on the OS. The compiled file is a shared library, with the form of an .so or .dll file in

[1] Project website: https://www.trafficfluid.tuc.gr
[2] See: https://sumo.dlr.de/docs/TraCI.html
[3] See: https://sumo.dlr.de/docs/Libsumo.html
[4] See: https://sumo.dlr.de/docs/Simulation/SublaneModel.html

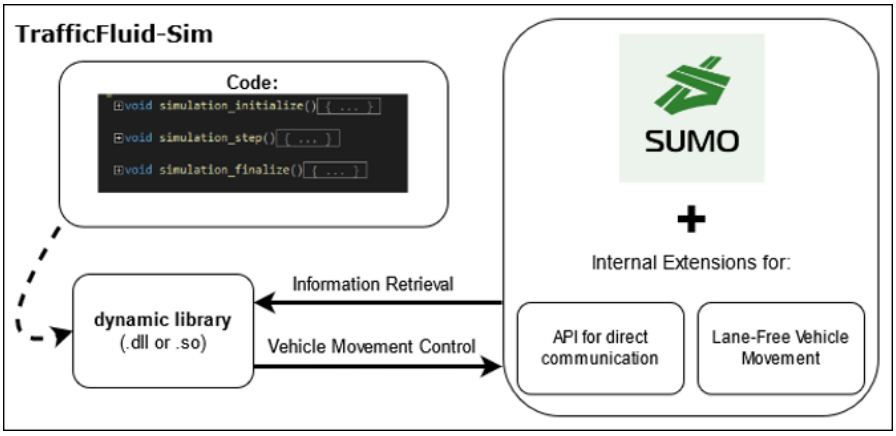


Figure 1: Core components that constitute TrafficFluid-Sim.

linux or windows, respectively. This is then combined with the main application of TrafficFluid-Sim at execution time.

1.2 How to cite

The present document should be cited if TrafficFluid-Sim is utilized for research purposes:

Troullinos, D., Chrysomallis, I., Papamichail, I., Papageorgiou, M.: TrafficFluid-Sim: Extending SUMO for Microscopic Simulation in Lane-Free Traffic – User Manual. Internal Report No. 2025-27, Dynamic Systems and Simulation Laboratory, Technical University of Crete, Chania, Greece, May 2025.

For convenience, the following BIB format can be used:

```
@techreport{trafficfluidsim_manual,
    author={Troullinos, Dimitrios and Chrysomallis, Iason
    and Papamichail, Ioannis and Papageorgiou, Markos},
    title={TrafficFluid-Sim: Extending SUMO for Microscopic Simulation in
    Lane-Free Traffic--User Manual},
    institution={Dynamic Systems and Simulation Laboratory,
    Technical University of Crete},
    number={Internal Report No. 2025-27},
    year={2025},
    month={May},
    location = {Chania, Greece}
}
```

1.3 Notes for Developers

For developers interested in exploring the underlying code of the simulator, it is important to note the complexity of navigation due to the extensive number of interconnected classes and files. To assist those interested in the codebase, we provide the following remarks:

- The tool is available for Windows (10/11) and Linux (Ubuntu 22.04 LTS).

- We provide the binaries for Ubuntu 22.04 LTS, but one can build the source code for other distributions as well following the instructions: https://sumo.dlr.de/docs/Installing/Linux_Build.html.
- Custom code changes and additions are annotated with comments labeled "LFPlugin begin" and "LFPlugin end" to clearly indicate the modified sections. An example is shown below:

```
void
MSDevice_Tripinfo::generateOutput(OutputDevice* tripinfoOut) const {
    const SUMOTime timeLoss = MSGlobals::gUseMesoSim ? myMesoTimeLoss : static_cast<MSVehicle&>(myHolder).getTimeLoss();

    // LFPlugin Begin
    const SUMOTime timeLossNoNeg = MSGlobals::gUseMesoSim ? myMesoTimeLoss : (static_cast<MSVehicle&>(myHolder)).getTimeLossNoNeg();
    const SUMOTime timeLossExcludeEdges = MSGlobals::gUseMesoSim ? myMesoTimeLoss : (static_cast<MSVehicle&>(myHolder)).getTimeLossExcludeEdges();
    const SUMOTime timeLossNoNegExludeEdges = MSGlobals::gUseMesoSim ? myMesoTimeLoss : (static_cast<MSVehicle&>(myHolder)).getTimeLossNoNegExcludeEdges();
    // LFPlugin End


    const double routeLength = myRouteLength + (myArrivalTime == NOT_ARRIVED ? myHolder.getPositionOnLane() : myArrivalPos);
    const SUMOTime duration = (myArrivalTime == NOT_ARRIVED ? SIMSTEP : myArrivalTime) - myHolder.getDeparture();
```

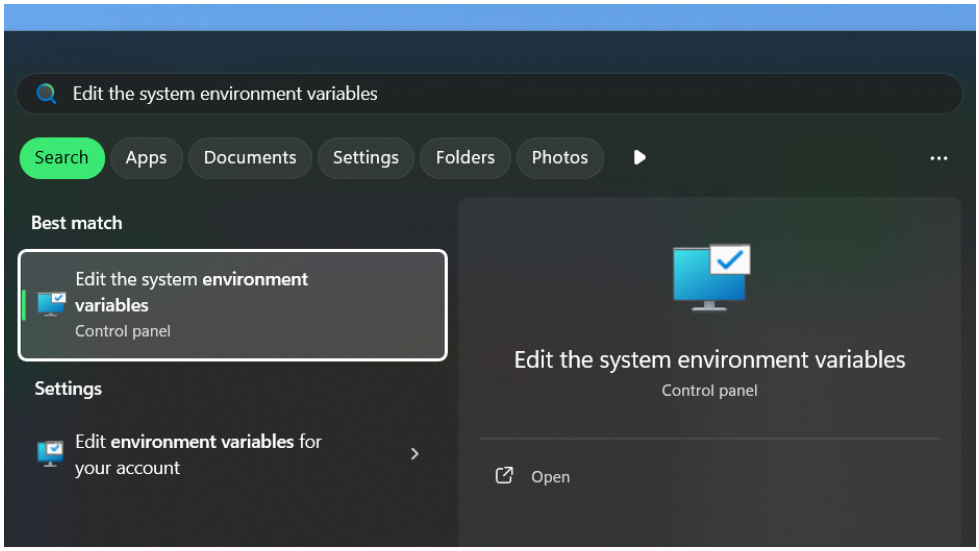
- The simulator is based on a fork of the original open-source SUMO code (https://github.com/eclipse-sumo/sumo), specifically the commit with ID 6c09f092 (dated October 20, 2020).
- We initiated our extension according to https://sumo.dlr.de/docs/Developer/How_To/Car-Following_Model.html. The central files that handle lane-free vehicle movement, communication with the dynamic library, and integration with the main application are located in the src/microsim/cfmodels folder and are titled MSCFModel_LaneFree.cpp, MSCFModel_LaneFree.h.
- For Windows, the prerequisite libraries were sourced from the SUMOLibraries repository (https://github.com/DLR-TS/SUMOLibraries) based on the commit with ID 0c8ca714 (dated May 19, 2020).

2 Setup in Windows 10/11

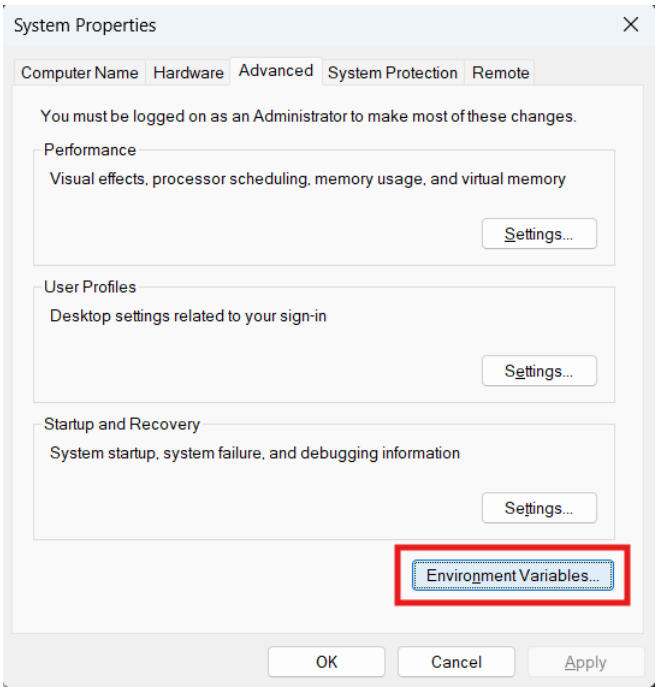
For Windows installation, in our website (https://www.trafficfluid.tuc.gr/en/trafficfluid-sim) we provide a zip file (trafficfluid-sim_vx_x_windows.zip) of the folder sumo_windows that contains the main application. The unzipped folder can be placed anywhere, but make sure that the path to the sumo_windows folder is in English and does not contain any special character, spaces, etc. The code structure of the user, provided in lanefree_plugin, is required to generate the dynamic library libLaneFreePlugin.dll to be linked with the main application, namely the sumo-gui.exe executable file inside the bin folder.

2.1 Environment Variables

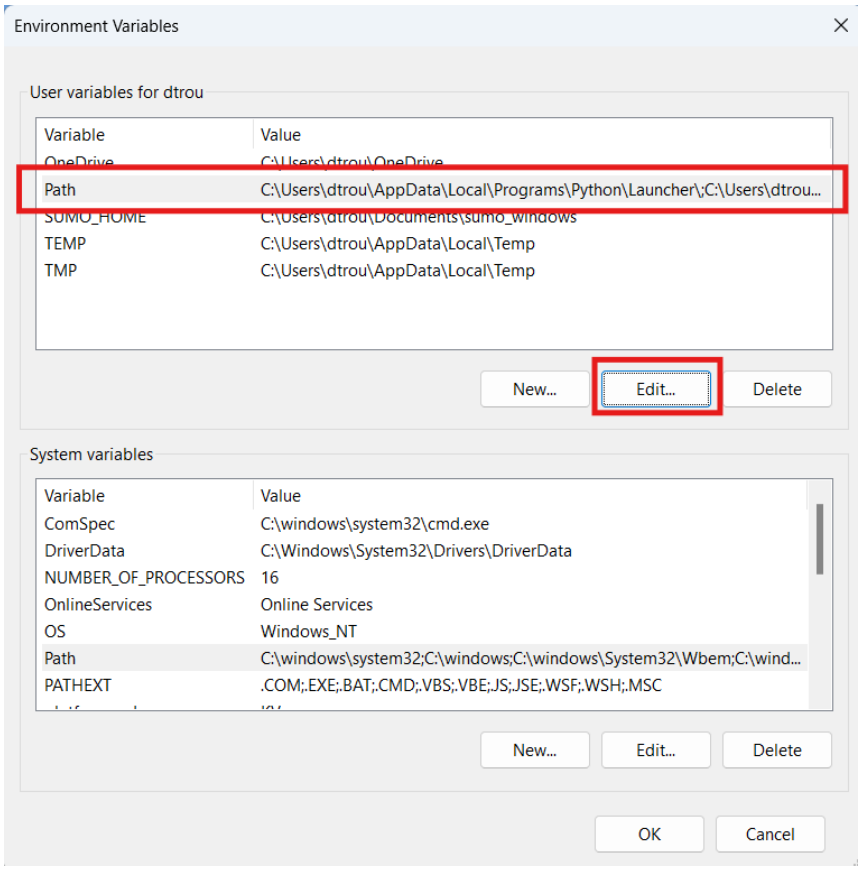
The use of Environment Variables in Windows systems is essential for linking the application with specified folders. Go to the Environment Variables:


Edit the system environment variables
Search
Apps
Documents
Settings
Folders
Photos
Best match
Edit the system environment variables
Control panel
Settings
Edit environment variables for your account
Edit the system environment variables
Control panel
Open

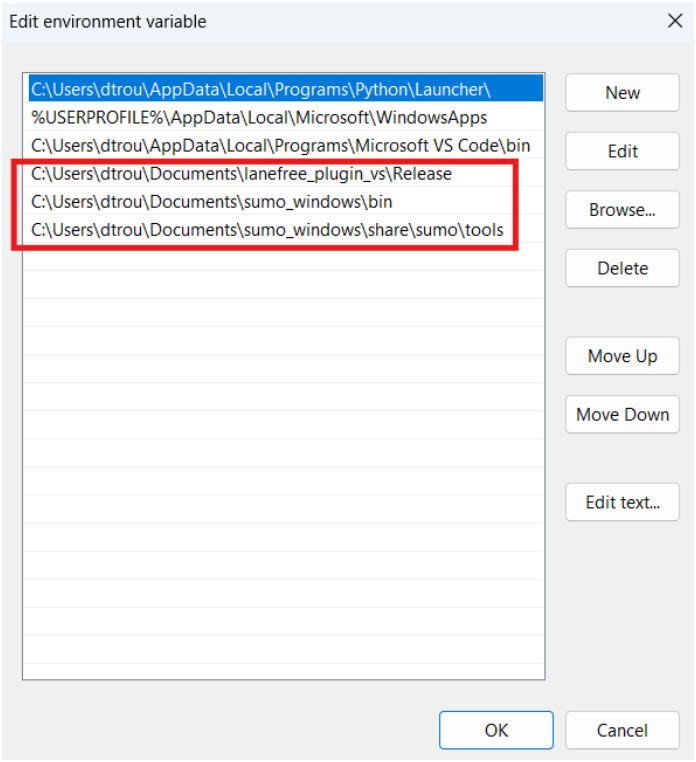
Then, click on the relevant button in the window below:



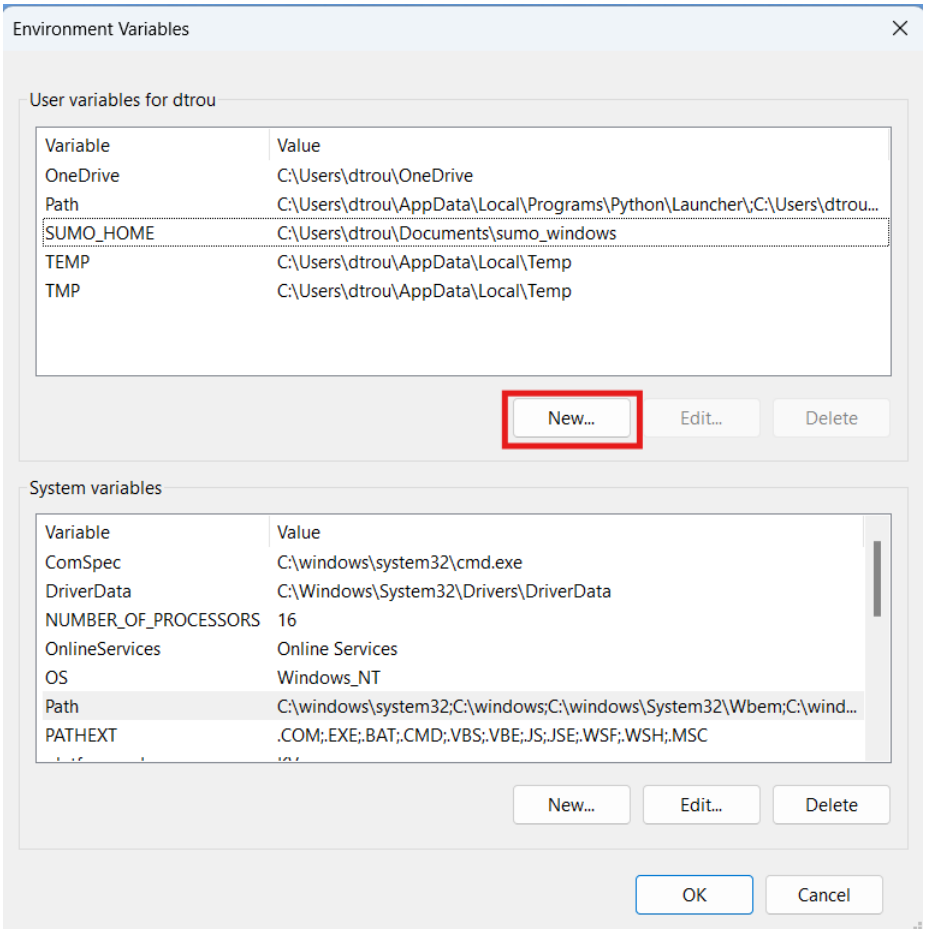
We need to add new entries to the Path variable by pressing the associated variable and pressing Edit:



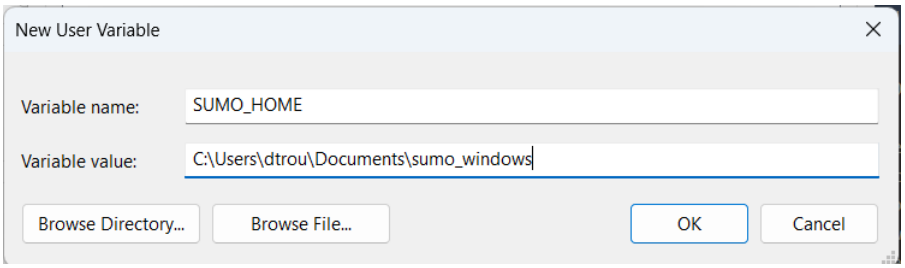
The 3 new entries should be added as shown below, by replacing the path to sumo_windows folder
C:Users\dtrou\Documents\sumo_windows with the one in the user's system,
and C:\Users\dtrou\Documents\lanefree_plugin\Release with the path to the libLaneFreePlugin.dll file. Then, press OK.



Moving forward, we need to set the SUMO_HOME variable by pressing the New button.



And finally, place as value the path to the sumo_windows folder, and press OK to all open windows.



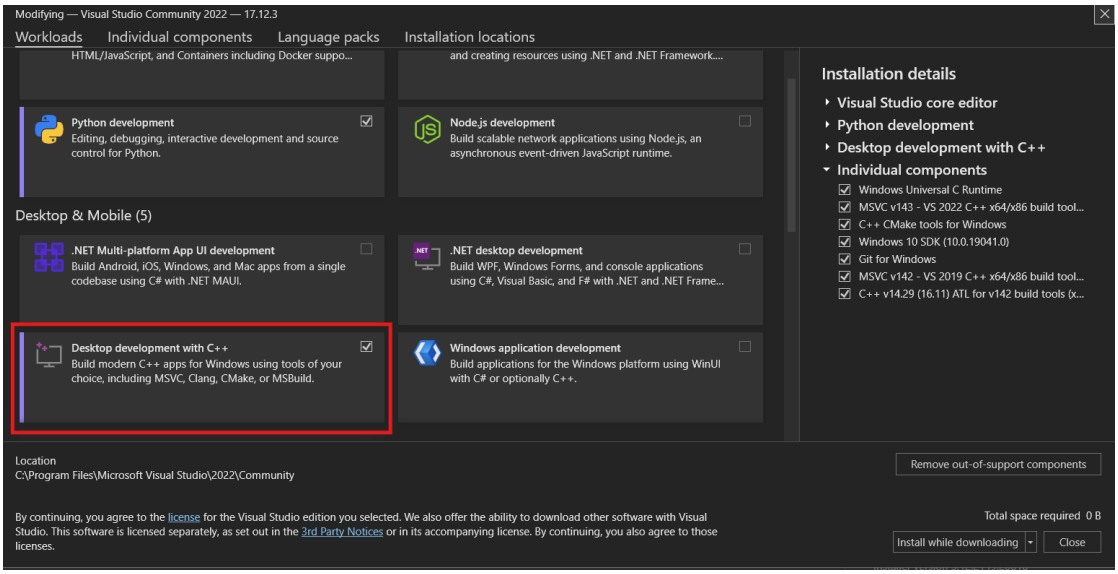
Note that we applied these changes to the User variables of the logged-in user. However, one can follow the same procedure for the System variables instead (e.g., if interested to use the application in a unified setting for multiple users).

2.2 Visual Studio Setup

For the provided code structure at https://github.com/trafficfluid-dssl/trafficfluid-sim/tree/main/lanefree_plugin_example, we rely on CMake[5] to generate the required makefiles, giving us the flexibility to cross-compile the code across windows and linux systems. For windows users, we also provide an alternative option in lanefree_plugin folder. The file liblanefreeplugin.sln can be used instead for Visual Studio (not to be confused with Visual Studio Code). Users working in Windows and not familiar with CMake are encouraged to setup Visual Studio and work with this version.

Users can find and download Visual Studio from https://visualstudio.microsoft.com/. Make sure to configure it for C/C++ development (check the related box during installation). We have checked the code in Visual Studio Community 2019 & 2022.

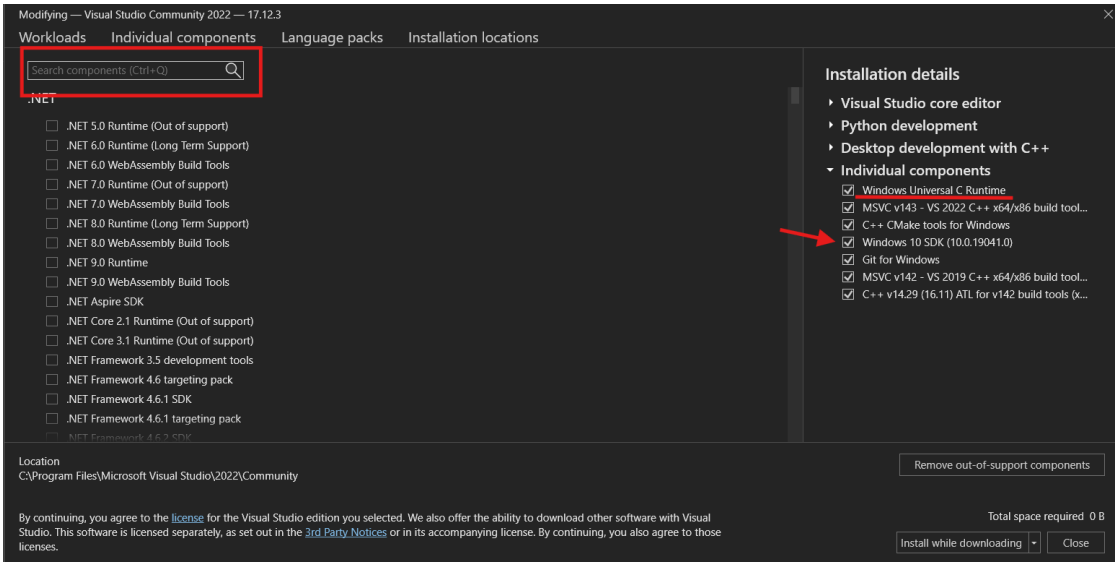
When the Installer pops-up, one should select the Workload for Desktop development with C++, as it appears below:



Before proceeding with the installation process, one can perform the following **optional steps**, suggested for alignment with the development setup of the main TrafficFluid-Sim application. For this, one needs to select the "Individual components" tab (see list of tabs at the top segment of the snapshot) and search for specific components either by scrolling through the list or by using the search bar (as indicated in the red box below) and:

1. replace the default option for the Windows SDK, with the version we currently use (as shown in the image below)
2. include Windows Universal C runtime
3. include C++ Cmake tools for Windows
4. (optional, if git repositories are utilized) include Git for Windows

[5] See: https://cmake.org/



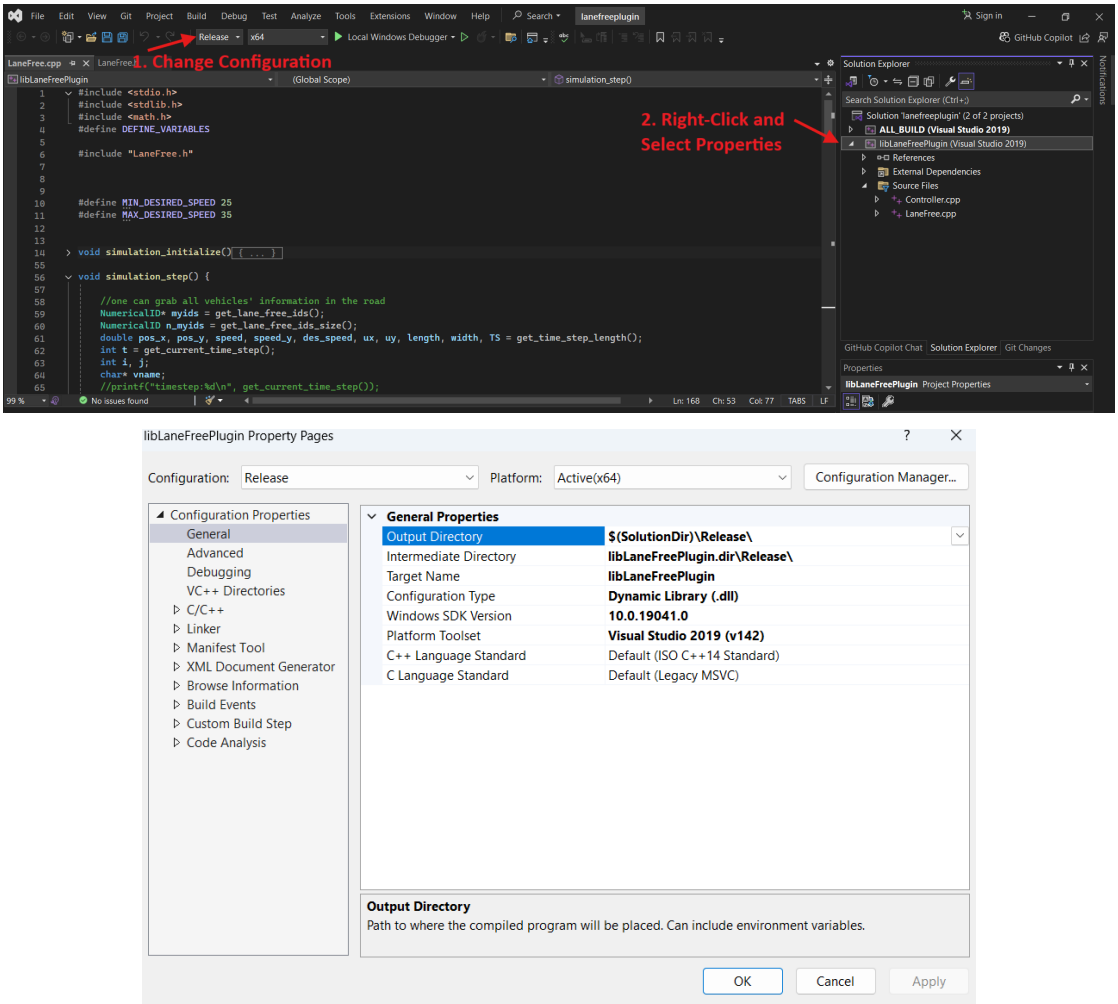
Note that even after the installation process, the Visual Studio Installer is accessible as a separate app for any post-installation modifications.

First-time Configuration of the Code Plugin After the installation, users should open the code through the provided liblanefreeplugin.sln project file inside the **lane_free_plugin_example** folder containing the code.

Only for the first time opening the project in a new PC setup, users should check and verify that their Configuration is set up properly, and make adjustments if necessary to comply with their own setup. To do so, after opening the liblanefreeplugin.sln, users should perform the following (as shown in the image below):

1. Change Configuration to Release Mode (if not already)

2. Navigate to Solution Explorer (by default located on the right side of the IDE environment as shown below), Right-Click on the laneFreePlugin entry and Press the "Properties" option. In the second image below, check that the **Windows SDK Version** and **Platform Toolset** comply with the setup. In case there is an issue, a relevant message (that this version is missing) will be displayed instead, and users will be able to adjust the version with the one available in the system through a drop-down menu when clicking on top of these.



We encourage the Release option as it removes unnecessary checks and compiles a potentially faster .dll to be bound with the main application.

If interested to perform debugging, the option is available through the Debug mode (instead of the Release Configuration in step 1. above). In that case, users should make sure to update the path to their .dll file accordingly, and use the attach to process function right after they open the main sumo-gui.exe application.

Finally, every time users wish to recompile their code and generate an updated libLaneFreePlugin.dll file, they can go to **Build** → **Build Solution** or use the keyboard shortcut **Ctrl+Shift+B**.

3 Setup in Ubuntu Linux

For linux, we provide a compiled version for Ubuntu 22.04 LTS (trafficfluid-sim_vx_x_ubuntu_22_04_lts.zip) in our website (https://www.trafficfluid.tuc.gr/en/trafficfluid-sim) that contains the sumo-gui application inside the bin folder. For an Ubuntu-based installation, the prerequisite packages (as specified in https://sumo.dlr.de/docs/Installing/Linux_Build.html) to be installed are:

```
sudo apt install git cmake python3 g++ libxerces-c-dev libfox-1.6-dev
libgdal-dev libproj-dev libgl2ps-dev python3-dev swig default-jdk maven
libeigen3-dev

sudo apt install ccache libavformat-dev libswscale-dev
libopenscenegraph-dev python3-pip python3-build
```

Once these packages are installed, one can launch the main application sumo-gui, given that the user has generated the shared object libLaneFreePlugin.so file. With the lanefree_plugin folder that includes the code, we provide a CMake configuration that compiles the source files in the src folder. To make use of the cmake format, i.e., generate first the makefiles and then compile the project, navigate to the lanefree_plugin folder and do the following:

First, create a build folder to generate all build-related files, and enter it:

```
mkdir build
cd build
```

Then, run cmake command to generate the makefiles based on the provided CMakeLists.txt file:

```
cmake ../.
```

Finally, compile the project:

```
make
```

This will generate the libLaneFreePlugin.so file. Every time users perform changes in the code, they only need to run the make command again to recompile the final shared object.

It is only needed to perform again the whole procedure when the folder is transferred to a different path or a new PC.

Alternatively, one can make use of VSCode (https://code.visualstudio.com/), install the packages related to C/C++ development and CMake Tools, and then open the folder inside VSCode. The program will automate the whole procedure above.

The sumo-gui application needs to be linked with the libLaneFreePlugin.so when executed. In Linux, this can be done by either placing sumo-gui and libLaneFreePlugin.so in the same folder, or run sumo-gui from anywhere and place libLaneFreePlugin.so in the /usr/local/lib folder that the OS automatically looks for .so files.

Additional Notes: In linux, the sumo-gui application file is portable (not the case for Windows, where the executable is bound to the bin folder), meaning that one can copy it to any folder for usage. Note that linux users should grant execution rights to the sumo-gui file. This can be done either through the file manager or with the chmod command, as shown below:

```
chmod +x sumo-gui
```

The application in linux-based systems is typically launched through the terminal:

```
./sumo-gui
```

4 Setup in Other Linux distributions and MacOS

For this, we expect a user that is proficient with programming tools in general. Full access to the TrafficFluid-Sim code is available at the following repository: https://github.com/trafficfluid-dssl/trafficfluid-sim/. It can be compiled for other linux distributions or MacOS by following the build instructions for the standard SUMO application (https://sumo.dlr.de/docs/Installing/Linux_Build.html). Note that the dynamic library file libLaneFreePlugin.so should be already generated and placed in a location that the OS looks at, e.g. in the case of Ubuntu at /usr/local/lib. For MacOS, one needs to generate the libLaneFreePlugin.dylib of the code (again with CMake) and again place it accordingly. We have already made changes in the main CMake file to support MacOS under the newer Apple silicon (ARM-based SoCs) as well. Users should follow the Homebrew instructions at: https://sumo.dlr.de/docs/Installing/MacOS_Build.html.

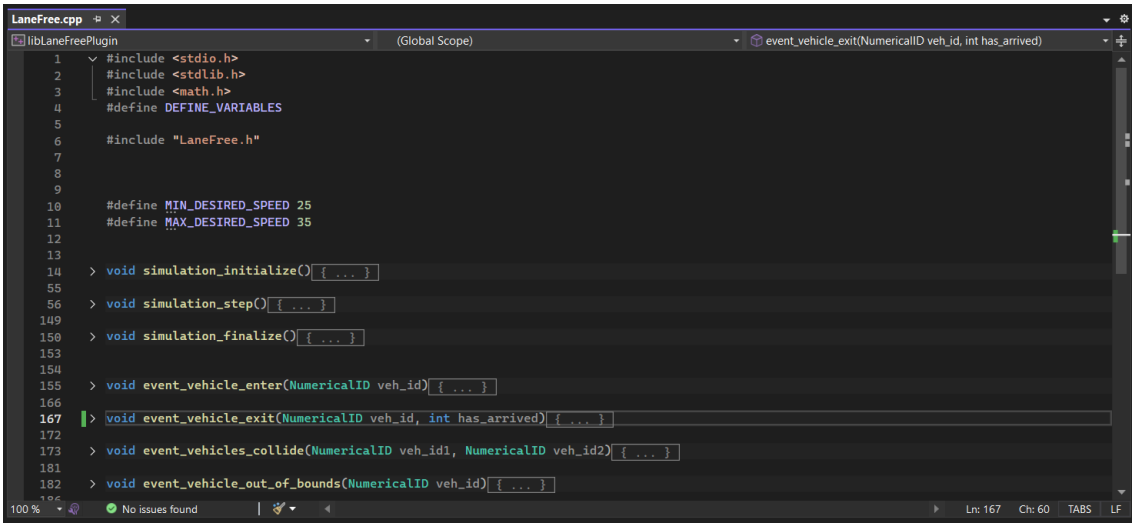
5 Lane-Free Plugin and Basic Principles for Designing Lane-Free Vehicle Movement Strategies

In what follows, we provide concise instructions for working with the available code structure, communicating with the main application at execution time, some guidelines relevant to information retrieval and control of lane-free vehicles, and the available movement dynamics.

5.1 Code Structure

The source code for the lane-free plugin example is available at: https://github.com/trafficfluid-dssl/trafficfluid-sim/tree/main/lanefree_plugin_example. Inside the src/ folder, we contain the primary 2 files LaneFree.cpp and LaneFree.h that are responsible for the connection with the main application.

The LaneFree.cpp file has the following structure:



- **simulation_initialize**: The simulation_initialize function is executed once before the simulation begins. Its use is suggested for memory allocation purposes, initialization of global

variables, etc. In addition, a popular use-case is for users interested in inserting vehicles at the start of the simulation. The provided file includes an example with the function insert_new_vehicle that places vehicles inside the simulation according to the initialization parameters of the user. Notably, a vehicle can be inserted by the user with this function at any point afterwards during the simulation.

- **simulation_step**: Then, simulation_step is the primary function that is executed once at every discrete simulation time-step and is responsible for the control of the vehicles. There, the user has the capability through the function calls of API (in LaneFree.h, see Section 5.2) primarily to a) monitor the current state of the traffic environment and b) provide control input to each vehicle. Refer to the code example in the provided file.

- **simulation_finalize**: Finally, the simulation_finalize is called once after the simulation ends either normally or abruptly (e.g., user closes the application window or the code crashes). As such, its purpose is usually for memory deallocation, logfiles, etc.

In addition, we also include several useful event-based functions, namely:

- **event_vehicle_enter**(veh_id): The event_vehicle_enter is for purposes of initialization when a vehicle enters the road (either due to the API function insert_new_vehicle or due to a flow demand in the scenario). It is called once when a vehicle enters the road with its unique ID information (prior the simulation_step at the corresponding time-step).

- **event_vehicle_exit**(veh_id,has_arrived): The event_vehicle_exit is called once when the vehicle is removed from the simulation, providing its ID information through the first argument. This can be either when the vehicle reaches the end of its designated route (then, the input argument has_arrived is true) or if the simulation ends while the vehicle is still on the road. The latter can occur either due to an ending time specified in the scenario, or premature ending (user closes application window or code crashes). Users can utilize this function for logfiles relevant to specific vehicles, deallocate memory related solely to this vehicle etc.

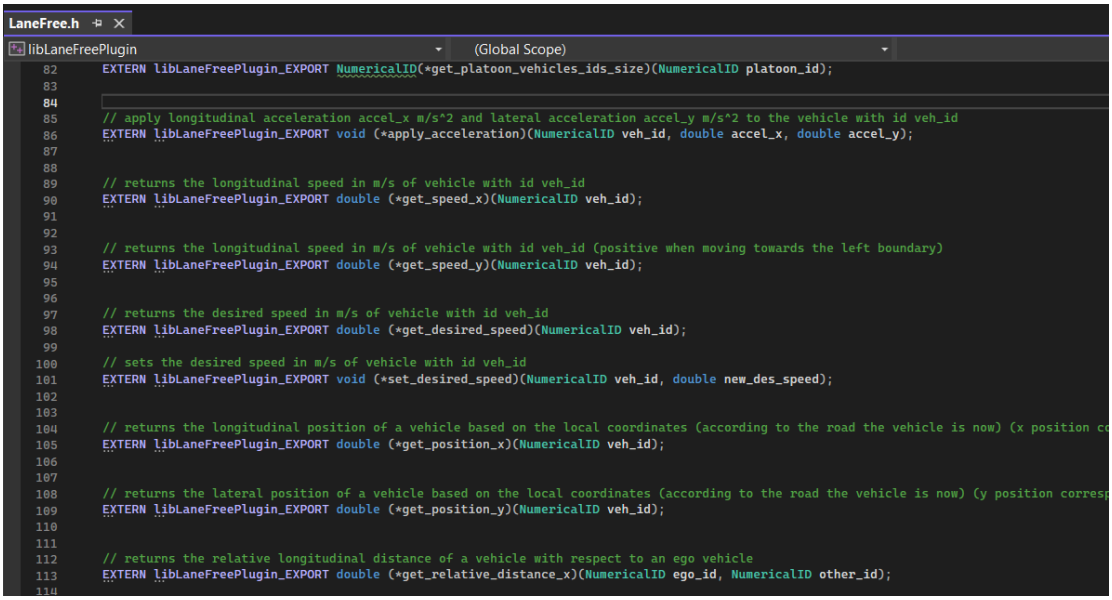
- **event_vehicles_collide**(veh_id1,veh_id2): The event_vehicles_collide that informs when there is a collision between two vehicles, providing information of their respective IDs.

- **event_vehicle_out_of_bounds**(veh_id): The event_vehicle_out_of_bounds informs in case the vehicle with ID veh_id exceeds the road boundaries.

While LaneFree.cpp is the "main" code file that should be available, users can include additional files for better code organization. We include a simple example (Controller.cpp and Controller.h files) that showcases how to include additional files with proper usage of the extern commands when necessary.

5.2 API

The developed API is responsible for online communication with the main application. Essentially, we have a set of available function calls inside the LaneFree.h file. Each function is accompanied by a short description of its usage, as shown below.



Note that for Windows users in Visual Studio, all header files are bundled inside the External Dependencies folder alongside other system header files for general purposes.

For reasons of speed and memory efficiency, we primarily rely on pointers for array-based information. As such, for function calls that yield vector-based information, we usually return a pointer to the array; and have either a separate function call to retrieve the size of the array, or rely on pass by reference to fill out the size information in the same function.

Very importantly, memory management of all the returned pointers is handled automatically. Users should:

(a) Never attempt to deallocate these pointers, and

(b) Make sure that they parse/store the relevant information before they call the same function again, as we usually maintain the same pointers and overwrite the new results on the corresponding memory addresses.

For some function calls, we rely on the C++ std::vector (https://en.cppreference.com/w/cpp/container/vector) template as return value for array-based information.

5.3 Information Retrieval and Vehicle Control

We rely on a discrete time-step that the user can specify when designing a scenario (see Section 6.1). As such, one should integrate a vehicle movement strategy within the simulation_step function. Given this, the user should provide appropriate control input at every time-step for each vehicle inside the simulation, depending on the selected vehicle movement dynamics, as we discuss in the subsequent Sections 5.3.1 and 5.3.2. Control input is either the 2-dimensional acceleration (in m/s^2) or the combination of acceleration magnitude (again in m/s^2) and steering target (in rad) depending on the selected movement dynamics. If control input is not provided for a vehicle, then a zero control vector is simply applied (retaining current speed and orientation).

Vehicle-related information is requested through the ID of each vehicle, specifically the NumericalID variable type (consistent with the standard SUMO codebase). To gain access to all vehicles currently occupying the road, one can use the related function for the array information and size:

```
NumericalID* myids = get_lane_free_ids();
NumericalID n_myids = get_lane_free_ids_size();
```

Then, users can iterate over this array, to request information about the vehicle's current state with a for loop, as in the basic example below.

```
for (i = 0; i < n_myids; i++) {

    pos_x = get_position_x(myids[i]);
    pos_y = get_position_y(myids[i]);
    speed = get_speed_x(myids[i]);
    length = get_veh_length(myids[i]);
    width = get_veh_width(myids[i]);
    des_speed = get_desired_speed(myids[i]);

    speed_y = get_speed_y(myids[i]);
    uy = erfc(speed_y)-1;
    ux = erfc(speed-des_speed)-1;
    apply_acceleration(myids[i], ux, uy);
    vname = get_vehicle_name(myids[i]);
    //printf("%s\n",vname);

    //Check if the positions follow the equations of motion for the double integrator
    // printf("vehid: %s in pos: %f, %f \n",vname, pos_x, pos_y );
    // pos_x = pos_x + speed*TS + 0.5*ux*TS*TS;
    // pos_y = pos_y + speed_y*TS + 0.5*uy*TS*TS;
    // printf("next pos: %f, %f \n", pos_x, pos_y );
}
```

The getter functions return information for the parsed ID. In the simple example above we request basic information for each vehicle, and have a simple controller where the vehicles target their desired speed objective (des_speed) on the longitudinal axis x and 0 lateral speed for the lateral axis y.

However, we suggest the following alternative way of parsing the vehicles especially when operating in large road networks. This is accomplished by first iterating over the road network. Vehicle information is **grouped by their residing road segment**, and there we also **maintain an ordered array** of vehicles according to the longitudinal positioning of the vehicles. In the example below from the provided LaneFree.cpp file, we show first how to parse vehicles to be controlled per road edge of the network. With get_all_ids_in_edge, we can get an ordered array based on positioning. This could be useful under certain situations, and is the suggested way to parse the vehicles.

Typically, in the context of microscopic simulation, we have a closed-loop system where the vehicle's controller requires information regarding its surrounding at each new time-step. As such, the user can acquire such information through the API, e.g., state information of each vehicle and its surroundings. For the surrounding traffic in a typical lane-based environment, we would follow the lane structure to provide such information, i.e., leader, follower, and two closest vehicles in each adjacent lane. In contrast, for lane-free traffic, the user instead specifies a longitudinal distance (obs_distance), based on which we return an array with ID information of all neighboring vehicles up to that distance. As shown in the code below, there are two separate function calls to obtain information about neighbors in front or in the back with respect to the vehicle requesting such information.

Importantly, for vehicle movement strategies where users rely on local coordinates,[6] we strongly suggest the usage of function calls:

[6] x,y coordinates are with respect to the residing road segment of the network

- get_relative_distance_x(ego_id, other_id)
- get_relative_distance_y(ego_id, other_id)

to obtain information regarding the distance with surrounding neighbors (longitudinal or lateral, respectively). This is especially important for environments with multiple road segments, or when emulating ring-roads (see Section 10). We have carefully developed these functions so that the distance information captures the change of the local coordinates from the perspective of the ego vehicle that requests this information (e.g., when a neighbor lies in a different road segment, just subtracting the local positions for either axis can provide wrong information). Likewise, if the user is interested not just in the distance information, but the positioning of the neighboring vehicle (other_id), this can be directly inferred by: get_position_x(ego_id) + get_relative_distance_x(ego_id, other_id), and equivalently for the _y axis. Finally, note that this is only relevant for positioning of vehicles, and not for the speed-related information in this context.

```
double obs_distance = 20;
for(i=0;i<n_myedges;i++){
    //printf("edge id: %lld\n", myedges[i]);
    n_edge_ids = get_all_ids_in_edge_size(myedges[i]);
    length = get_edge_length(myedges[i]);
    width = get_edge_width(myedges[i]);
    if(n_edge_ids>0){
        //vehicles are ordered in this case based on their longitudinal position on the road
        ids_in_edge = get_all_ids_in_edge(myedges[i]);
        //printf("Vehicles in edge with id %lld:", myedges[i]);
        for (j = 0; j < n_edge_ids; j++) {
            NumericalID myID = ids_in_edge[j];
            front_neighbors = get_all_neighbor_ids_front(myID, obs_distance, 1, &front_neighbors_size);
            back_neighbors = get_all_neighbor_ids_back(myID, obs_distance, 1, &back_neighbors_size);

            //parse neighbors in front
            for (int k_f = 0; k_f < front_neighbors_size; k_f++) {
                NumericalID neigh_id_front = front_neighbors[k_f];
                // request information regarding this neighbor
                double dx = get_relative_distance_x(myID, neigh_id_front);
                double dy = get_relative_distance_y(myID, neigh_id_front);
                double speed_neigh = get_speed_x(neigh_id_front);
                // etc...
                // depending on the movement strategy design,
                // users need to develop a proper response to surrounding traffic
            }

            //parse neighbors in back
            for (int k_f = 0; k_f < back_neighbors_size; k_f++) {
                NumericalID neigh_id_back = back_neighbors[k_f];
                // request information regarding this neighbor
                double dx = get_relative_distance_x(myID, neigh_id_back);
                double dy = get_relative_distance_y(myID, neigh_id_back);
                double speed_neigh = get_speed_x(neigh_id_back);
                // etc...
                // depending on the movement strategy design,
                // users need to develop a proper response to surrounding traffic
            }
```

Of course, the user needs to apply at the end of the inner for loop statement the calculated control variables through the apply_acceleration function, as shown within the code example.

For better code organization, the user can design the vehicle movement strategy for each parsed vehicle ID in a function located elsewhere, that can be called at this point. As long as the header file of the API LaneFree.h is included wherever that function is defined, the user can

access all API calls as done in LaneFree.cpp (getting neighbors' ID information, their current state, applying the control input, etc). See the example below.

```
for(i=0;i<n_myedges;i++){
    //printf("edge id: %lld\n", myedges[i]);
    n_edge_ids = get_all_ids_in_edge_size(myedges[i]);
    length = get_edge_length(myedges[i]);
    width = get_edge_width(myedges[i]);
    if(n_edge_ids>0){
        //vehicles are ordered in this case based on their longitudinal position on the road
        ids_in_edge = get_all_ids_in_edge(myedges[i]);
        //printf("Vehicles in edge with id %lld:", myedges[i]);
        for (j = 0; j < n_edge_ids; j++) {
            vname = get_vehicle_name(ids_in_edge[j]);
            NumericalID ego_veh_id = ids_in_edge[j];
            vehicle_movement_strategy(ego_veh_id);
        }

    }
}
```

In the following two Subsections, we outline the two available movement dynamics in our lane-free microscopic simulation environment and the corresponding control input that is required for vehicle control.

5.3.1 (Double) Double Integrator Model

The Double Integrator Model is the default option for lane-free movement dynamics, and is already available as an option in the standard SUMO application with the term 'Ballistic'.[7] Its usage in SUMO is limited only for the longitudinal movement update (x axis), i.e., longitudinal position x and speed v_x. We are interested to extend this for 2-dimensional movement. Therefore, we embed internally in TrafficFluid-Sim the (Double)[8] Double Integrator Model, which likewise updates the lateral position y and speed v_y of the vehicles.

Essentially, the Double Integrator Model assumes a constant acceleration (the control variable) throughout the discrete time-step length T. As such, at time-step k, given an acceleration input for both dimensions a_x, a_y (in m/s^2), the equations of motion for each vehicle are as follows:

$$v_x(k+1) = v_x(k) + a_x \cdot T \tag{1}$$

$$v_y(k+1) = v_y(k) + a_y \cdot T \tag{2}$$

$$x(k+1) = x(k) + v_x(k) \cdot T + \frac{1}{2} \cdot a_x \cdot T^2 \tag{3}$$

$$y(k+1) = y(k) + v_y(k) \cdot T + \frac{1}{2} \cdot a_y \cdot T^2 \tag{4}$$

The function call in the API for vehicle with ID veh_id is:

- apply_acceleration(veh_id, a_x, a_y)

[7] See https://sumo.dlr.de/docs/Simulation/Basic_Definition.html#defining_the_integration_method
[8] Two dimensions x,y

with a_x, a_y provided in m/s^2.

The use of double integrator is especially targeted towards highways, where the following considerations are appropriate:

- Orientation of vehicles is always in parallel with the road segment
- Lateral movement is completely independent from longitudinal movement

These assumptions render the design of the vehicle movement strategy much easier, since the orientation of the vehicle is controlled automatically, and the vehicle follows its routing scheme as an *unfolded straight road*, in which it may only need to perform lateral maneuvers in order to follow it (e.g., on-ramp merging, or exit from an off-ramp).

However, these assumptions may be unfitting for lane-free environments such as roundabouts and intersections, where the cartesian coordinates might be limiting and the user might be interested to have control over the orientation of the vehicle as well, or in general a more realistic depiction of the movement dynamics. For that, we have incorporated the *bicycle model* as an alternative.

5.3.2 Bicycle Model

For the bicycle model dynamics, we no longer have independent control over the longitudinal and lateral movement. Instead, the control variables now are the acceleration magnitude F and the steering angle δ. We also introduce information regarding the orientation θ of each vehicle. Then, for the movement dynamics we update the longitudinal ($x_b(k)$) and lateral ($y_b(k)$) rear axle midpoint of each vehicle; and the speed variable is now 1-dimensional ($v(k)$) as we now model the orientation as well. Since we rely on discrete-time microscopic simulation, we again assume constant values for the two control signals over the time-step length T, and employ the equations for the movement dynamics from [6], but included an extra case for the position update (Equations 7, 8) when $\delta = 0$, to avoid division by zero ($tan(0) = 0$). As such, at time-step k and with control variables F, δ the relevant variables x_b, y_b, v, θ are updated for $k+1$ as:

$$\theta(k+1) = \theta(k) + v(k)\frac{\tan(\delta)}{l}T + F\frac{\tan(\delta)}{2l}T^2 \tag{5}$$

$$v(k+1) = v(k) + F \cdot T \tag{6}$$

$$x_b(k+1) = \begin{cases} x_b(k) + \frac{l}{\tan(\delta)} \cdot \Big[\sin(\theta(k+1) - \sin(\theta(k))\Big] & \text{if } \delta \neq 0 \\ x_b(k) + v(k) \cdot \cos(\theta(k)) \cdot T + F \cdot \cos(\theta(k)) \cdot \frac{T^2}{2} & \text{otherwise} \end{cases} \tag{7}$$

$$y_b(k+1) = \begin{cases} y_b(k) + \frac{l}{\tan(\delta)} \cdot \Big[\cos(\theta(k) - \cos(\theta(k+1))\Big] & \text{if } \delta \neq 0 \\ y_b(k) + v(k) \cdot \sin(\theta(k)) \cdot T + F \cdot \sin(\theta(k)) \cdot \frac{T^2}{2} & \text{otherwise} \end{cases} \tag{8}$$

with l corresponding to the length of the vehicle, and for the case $\delta = 0$, i.e., the orientation of the vehicle will remain constant ($\theta(k+1) = \theta(k)$), and the position x_b, y_b update essentially reflects the Double Integrator Model.

Important note: Users should be careful when using control values where $|\delta| < \epsilon$ below some ϵ threshold that the C++ compiler would render the numerical result of $l/\tan(\delta) = \pm inf$.

We can obtain the relevant information that is unique to the bicycle model with provided API calls. Specifically, for the scalar speed information $v(k)$ with the function:

- get_speed_bicycle_model(veh_id)

and the orientation of the vehicle θ with:

- get_veh_orientation(veh_id)

For retrieval of position information, user can utilize the same functions as before, but note that we return the **center** point of the vehicle. Back position x_b, y_b can be calculated by accounting for length and width of the vehicle through the available function calls that return such information. The function call to apply the control input (F,δ) in the API for vehicle with ID veh_id is:

- apply_control_bicycle_model(veh_id, F, delta)

with F provided in m/s^2 and delta in rad.

5.3.3 Global Coordinates Control

Following the paradigm of the existing simulation infrastructure of SUMO, we utilize local coordinates for vehicles in order to simplify the design of vehicle movement strategies. As such, the positioning of the vehicles is with respect to the residing road segment/edge of the network. Especially for the double integrator case, this is an indispensable choice for road networks where the orientation of the vehicles should be affected. With local coordinates, the vehicles follow their designated route and their orientation is automatically adjusted according to the orientation of the road (see related discussion at the beginning of Section 5.3). Therefore, the user has no control over turning operations.

This principle is followed for the bicycle model as well, at least in its default mode. While the orientation of the vehicles can be monitored and controlled, it is always with respect to the geometry of the residing road. Hence, in a more intricate scenario such as a roundabout, controlling the vehicles' turning and overall movement in a circular road over the local coordinates would be extremely cumbersome; and not actually reflective of the real-world scenario.

To facilitate these use cases, and in general grant full control over the vehicles movement, there is an *alternative option* exclusive to the bicycle model. Upon entrance, vehicles that are specified to be modeled according to the bicycle model, can "switch" to control the vehicle over the global coordinates with the following function of the API:

- set_global_coordinate_control(veh_id, use_global_coordinates)

for vehicle with id veh_id and an additional input argument use_global_coordinates. The latter serves as the switch that enables (use_global_coordinates=1) or disables (use_global_coordinates=0) the update of movement based on global coordinates. The suggested placement for this function is inside the event-based function event_vehicle_enter(veh_id).

Note that global coordinate control is specified per vehicle, and therefore the simulation infrastructure has the flexibility to simulate different movement strategies that may not use the same movement dynamics.

Information retrieval for Global Coordinates Control For global coordinates, the function call for orientation:

- get_veh_orientation(veh_id)

is automatically adjusted to now return the angle of the vehicle with respect to the global coordinates of the system, and not according to the x-axis of the residing road segment.

Important note: The positioning information (get_position_x,get_position_y) of the vehicle maintains the local coordinates' information. Users should now rely the get_global_position_x, get_global_position_y to obtain proper information in this mode. Likewise, the distance functions mentioned earlier (get_relative_distance_x, get_relative_distance_y) are not valid anymore, as they are targeted towards the double integrator model and address the issue of vehicles having correct distance information while they reside in different road segments, and therefore have different local coordinates.

Even though vehicles now operate with global coordinates, we took extra care in order to maintain the functionality of the functions that return information on neighboring vehicles. Therefore, each vehicle can—exactly as shown for the local coordinates case—request information about surrounding traffic either upstream or downstream (with respect to its routing).

As we expand below in Section 6, when setting up a scenario, we need to define each type of vehicles (as in the standard SUMO[9]) to be introduced in our lane-free scenario. There, we need to specify (per vehicle type) which of the two vehicle movement dynamics shall model the vehicles' movement and consequently the control input.

6 Basic Principles for Setting up Lane-Free Scenarios

To construct scenarios, we inherit the existing infrastructure of SUMO in order to design road networks,[10] define vehicle types and demand,[9] and combine these to a single configuration file.[11]

Naturally, there are several distinctions within the process in order to adapt to TrafficFluidSim. Below, we contain a highway environment example with accompanying instructions/details that showcase the basic elements.

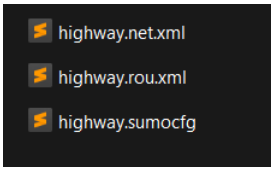
In subsequent Sections, we contain examples for more complex environments, and supply the necessary setup instructions. All examples are available at:
https://github.com/trafficfluid-dssl/trafficfluid-sim/tree/main/scenarios_examples.

6.1 Setup for a Highway Scenario

Users should have access to the accompanying scenario folder 01_highway_scenario. For every scenario (as in standard SUMO), we require three configuration files in xml format, namely a:

- Network file (.net.xml)
- Route file (.rou.xml)
- SUMO configuration file (.sumocfg)

with the associated files inside the scenario folder



[9] https://sumo.dlr.de/docs/Definition_of_Vehicles%2C_Vehicle_Types%2C_and_Routes.html
[10] Basic tutorial: https://sumo.dlr.de/docs/Tutorials/Hello_World.html
[11] https://sumo.dlr.de/docs/Tutorials/quick_start.html

Network file (.net.xml) The network file can be generated via the Netedit application. For Windows users, we have a Netedit executable that is tied to our version (inside the netedit folder), but one can directly use the standard application provided by SUMO. Refer to https://sumo.dlr.de/docs/Netedit/index.html to get familiar with the tool.

As a standard practice, we construct roads with one big lane that stretches along the designated road width, and add additional lanes only when necessary to connect road segments, i.e., acceleration/deceleration lanes for on-ramp/off-ramp scenarios (see Section 11). This is relevant to the demand handling functionality we have developed (refer to Section 6.3), since it operates over a lane's width at origin points of the network. Users are encouraged to examine the provided scenario examples to get more familiar.

Route file (.rou.xml) The route file includes definitions regarding:

- Types of vehicles to appear in the simulation
- All the different routes for the vehicles to follow
- Flow demand

Refer to the example below (highway.rou.xml):

```xml
<routes>

    <vType id="lane_free_car_example" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="3.20" width="1.60"
    minGap="0" tau="0.4" minGapLat="0.3" color = "yellow"/>

    <vType id="lane_free_car_bicycle_model_example" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree"
    carMovementDynamics="Bicycle" length="3.20" width="1.60" minGap="0" tau="0.4" minGapLat="0.3" color = "yellow"/>

    <vTypeDistribution id="typedist1">
        <vType id="lane_free_car11" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="3.20" width="1.60"
        minGap="0" tau="0.4" minGapLat="0.3" color = "blue"  probability="0.1666"/>
        <vType id="lane_free_car22" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="3.90" width="1.70"
        minGap="0" tau="0.4" minGapLat="0.3" color = "green"  probability="0.1666"/>
        <vType id="lane_free_car33" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="4.25" width="1.80"
        minGap="0" tau="0.4" minGapLat="0.3" color = "magenta"  probability="0.1666"/>
        <vType id="lane_free_car44" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="4.55" width="1.82"
        minGap="0" tau="0.4" minGapLat="0.3" color = "orange"  probability="0.1666"/>
        <vType id="lane_free_car55" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="4.60" width="1.77"
        minGap="0" tau="0.4" minGapLat="0.3" color = "yellow"  probability="0.1666"/>
        <vType id="lane_free_car66" vClass="passenger" maxSpeed="40" carFollowModel="LaneFree" length="5.15" width="1.84"
        minGap="0" tau="0.4" minGapLat="0.3" color = "white"  probability="0.1666"/>
    </vTypeDistribution>

    <route id="route0"  edges="long_edge" leftBoundaryLevelPoints="10.2 10.2" leftBoundaryOffsets="100" leftBoundarySlopes="
    0.045" rightBoundaryLevelPoints="0 0" rightBoundaryOffsets="100" rightBoundarySlopes="0.05" rightBoundaryConstant="0"
    leftBoundaryVisualizerColor="orange" rightBoundaryVisualizerColor="red"/>

    <flow id="normal_flow" type="typedist1" begin="0" end="3600" number="10000" route="route0" departSpeed="30" departPos="0"
     />
    <flow id="normal_flow2" type="typedist1" begin="3600" end="5400" probability="0.55" route="route0" departSpeed="25"
    departPos="0" />

</routes>
```

Vehicle Types In each vType xml tag, users can look at[9] for basic usage. The essentials include: define a unique string id, set vclass="passenger" for vehicles, and the basic characteristics, length, width, maxSpeed (default desired speed), colour etc. The parameters tau and minGapLat are relevant to the rules that the vehicle will follow for flow demand (see more details below in Section 6.3).

Important: Users need to set carFollowModel="LaneFree". This setting essentially binds the vehicle's control (not just for car following) to the dynamic library and grants access to custom control of vehicles through the API (see Section 5).

Then, the carMovementDynamics parameter has by default the value “DoubleIntegrator”. Therefore, its definition can be omitted for this case. To switch to the bicycle model, this should be set with the value “Bicycle” as shown in the example above.

One can also utilize the vTypeDistribution of SUMO as shown above, in order to have a single stochastic type of vehicles that samples one of the defined vTypes inside according to the probability values.

Routes The xml tag route specifies each route, with a unique string id parameter. One should again follow the instructions for SUMO[9] and also refer to our examples. In short, the route can be designed either with an array of consecutive edges (as defined in the network file); or specify only the first and last edge and SUMO has a mechanism to automatically generate the shortest path.

Important: Within the route tag, we also specify the shape of the lateral boundaries, with the related tags. If these are not provided, then the vehicle can only assume that the lateral limits of the road follow its width (through road width information from the API). See below in Section 6.2 for more detailed instructions.

Flow Demand The flow tag specifies the demand. Again, we rely on the existing format for SUMO. However, we have devised a mechanism that explicitly targets lane-free environments on how to introduce new vehicles at any origin point of the network and meet the specified flow demand. See more details in Section 6.3.

Configuration File (.sumocfg) The configuration file is the central component that combines all elements. This file is used to launch a scenario, as in the standard SUMO application.

Below, we contain the example for the highway scenario, then outline the primary elements, their purpose, and direct to corresponding sections that include further information and instructions for each element.

```
<configuration>
   <input>
       <net-file value="highway.net.xml"/>
       <route-files value="highway.rou.xml"/>
       <gui-settings-file value="gui-settings.cfg"/>
       <additional-files value="detectors.add.xml"/>
   </input>
   <processing>
         <step-method.ballistic value="true"/>
         <collision.action value="none"/>
         <step-length value="0.2"/>
         <seed value="0"/>
         <video-record-or-replay value="nothing"/>
   </processing>
   <output>
      <tripinfo-output value="trip_info.xml"/>
      <statistic-output value="avg_stats.xml"/>
      <video-logfile value="recording_file.log"/>
   </output>
</configuration>
```

Regarding the input files:

- net-file: Simply add the network file
- route-files: Add the route file (can be multiple)
- gui-settings-file: This is an optional file. We have a simple setting that renders automatically to the desired visualization instead of the default one on SUMO
- additional-files: This is used for additional terms, specifically for specifying detectors. We refer the user to Section 7.3.

Important argument settings for processing elements:

- step-method.ballistic: This should always be enabled. By default, SUMO employs simpler movement dynamics with constant speed (instead of constant acceleration) per time-step.
- collision.action: This should always be set to "none". SUMO can trigger certain actions when a collision occurs (e.g., teleportation, remove both vehicles[12]). In all our endeavours, we do not want to affect the behaviour of the vehicles. Each collision occurrence is reported through an event function in the provided code structure (see Section 5.1).
- step-length: This is the *time-step length* T in *seconds*.
- seed: This is the random seed value, which is also accessible through the API for synchronization with the vehicle movement strategy initialization. Therefore, with proper usage in the users' code, this ensures proper reproducibility of simulations.
- video-record-or-replay: This is an optional argument, where one can either "record" the scenario to the video-logfile value specified in line 18, or "replay" the scenario by loading the trajectories logfile that was previously recorded. More details on this feature can be found in Section 9. If not defined, or defined with any other value (e.g., "nothing"), then this feature is disabled.

Output files:

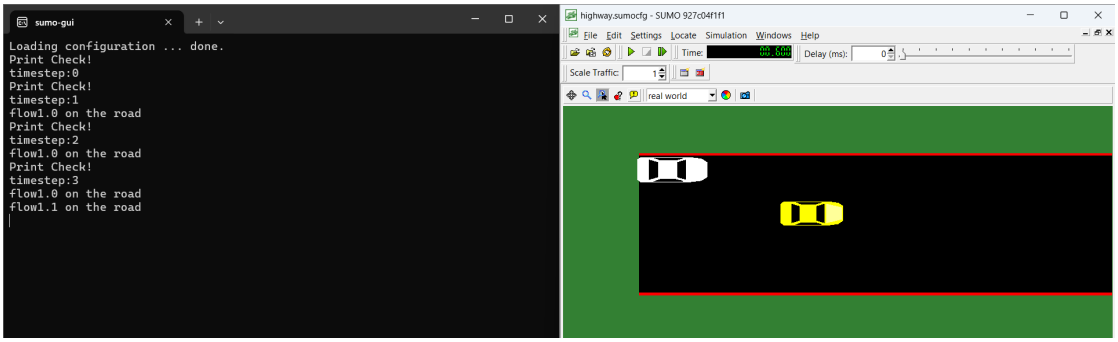
- tripinfo-output: Contains metrics for each individual vehicle that was spawned in the network. This feature directly follows from `https://sumo.dlr.de/docs/Simulation/Output/TripInfo.html`. See the related Section 8.
- statistic-output: Contains average metric for the whole scenario, e.g., average speed and delay. Refer again to Section 8 for more details and for clarification of adjustments and additional metrics we have introduced with respect to standard SUMO.
- video-logfile: Serves a dual purpose. When the scenario is configured to "record", all vehicle actions are logged and saved to the specified file. When set to "replay", the scenario reproduces all vehicle actions previously recorded and stored in the specified file.

All output files above are optional in principle, and can be omitted if the user is not interested in the relevant features.

Finally, to open the scenario, users need to launch the sumo-gui executable (inside the bin folder), and then: **File→Open Simulation** and navigate to the .sumocfg file inside their scenario folder. Alternatively(/Equivalently), simply press the Open Simulation button as shown below:

[12] `https://sumo.dlr.de/docs/Simulation/Safety.html#collisions`

A terminal window is launched alongside the main application. There, users can print out online information through their codebase with regular printing commands (e.g., printf, std::out), as shown below:

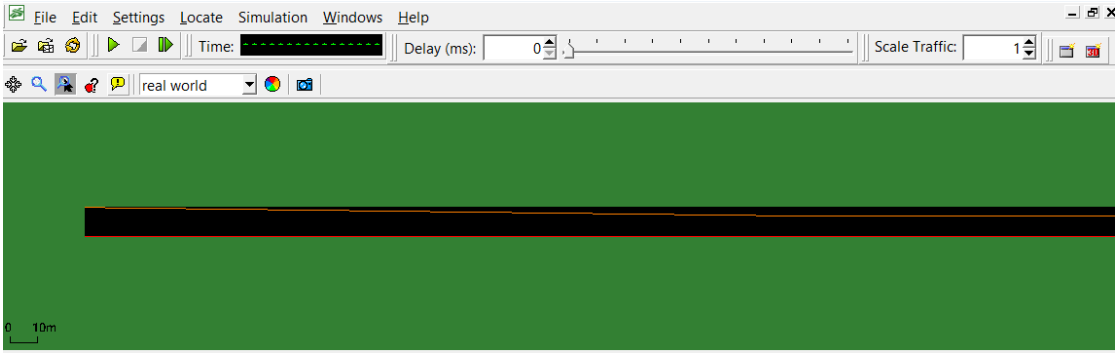


6.2 Lateral Boundaries in Lane-Free Traffic

We illustrate the design of lateral boundaries with 02_highway_boundary_shrink_scenario, a small variation of the 01_highway_scenario. There, inside the route file, we have:

```
<route id="route0"  edges="long_edge" leftBoundaryLevelPoints="10.2 7.2" leftBoundaryOffsets="100" leftBoundarySlopes="
0.045" rightBoundaryLevelPoints="0 0" rightBoundaryOffsets="100" rightBoundarySlopes="0.05" rightBoundaryConstant="0"
leftBoundaryVisualizerColor="orange" rightBoundaryVisualizerColor="red"/>
```

This will make the boundary shrink accordingly. If one loads the scenario, it should look like this:



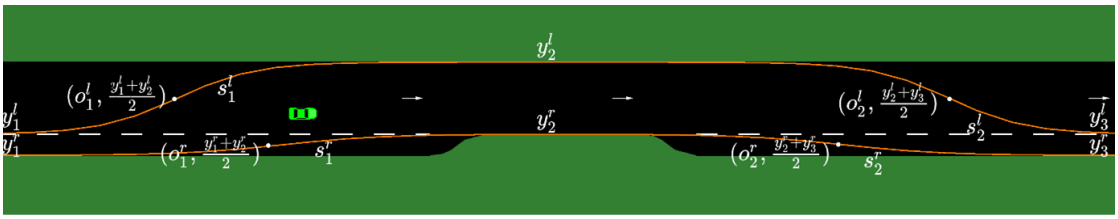
Moving boundaries are defined within each route's definition. Regarding the left boundary, the user needs to provide the following variables:

- leftBoundaryLevelPoints: contains the different lateral levels (in global coordinates)

- leftBoundaryOffsets: is the offset point in the longitudinal (x) axis (again, in global coordinates)
- leftBoundarySlopes: contains the slope for the corresponding offset point

Given a number N of leftBoundaryLevelPoints, then the associated offsets and slopes should be N-1 in size. Likewise, the user can provide the same information regarding the right boundary, at the following variables: rightBoundaryLevelPoints, rightBoundaryOffsets, rightBoundarySlopes.

In the illustrative example (taken from [3]) below:



the y^l, o^l, s^l (y^r, o^r, s^r) variables correspond to the leftBoundaryLevelPoints, leftBoundaryOffsets, leftBoundarySlopes (rightBoundaryLevelPoints, rightBoundaryOffsets, rightBoundarySlopes) respectively. If interested in the underlying equations, users can refer to [3] for a description with the full equations that control the boundaries.

Online boundary information retrieval through the API From the API part, there is a function for online information on the moving boundaries:

- get_distance_to_road_boundaries_at(veh_id, longitudinal_distance_x, * left_boundary_distance, * right_boundary_distance, * left_boundary_speed, * right_boundary_speed)

which calculates the lateral distance from left and right road boundaries for veh_id, at longitudinal_distance_x from the center of vehicle veh_id (vehicle can also observe upstream with negative values). Regarding the boundaries, it calculates the boundaries' distances (assigns values to the provided pointers left_boundary_distance, right_boundary_distance) and speed of the moving boundary (with left_boundary_speed, right_boundary_speed variables). If speed information is not useful, one can simply place NULL pointers to the respective arguments (left_boundary_speed, right_boundary_speed), and the function will automatically neglect this computation.

We can also visualize both left and right boundaries with these additional arguments:

- leftBoundaryVisualizerColor: select the color for the left boundary visualization (if this argument is not provided, the respective boundary will not be visualized). Common colors can be specified in SUMO with a String name, e.g., "white", "yellow", "green". Otherwise, use rgb coloring either for more customized options, or if users want to have opacity through rgba coloring. See here (at the ⟨COLOR⟩ bullet): https://sumo.dlr.de/docs/Basics/Notation.html#referenced_data_types
- leftBoundaryVisualizerStep: (optional argument) select the visualization accuracy, e.g., leftBoundaryVisualizerStep="10" means that the visualizer will draw a point every 10 meters (and then visualize the boundary by connecting all points). Minimum accepted (and default) value for this variable is: (length of path in meters) / (8*(#number of offsets))

- leftBoundaryVisualizerLineWidth: (optional argument) select the width of the line, e.g., leftBoundaryVisualizerLineWidth="0.2" means that the line will have a thickness of 0.2 meters. Default value is 0.1.

Same principle is followed for the right boundary: rightBoundaryVisualizerColor, rightBoundaryVisualizerStep, rightBoundaryVisualizerLineWidth. The only exception is the optional variable rightBoundaryConstant which can be prescribed only for the right boundary, and is relevant for the Internal Boundary Control application (see Section 12 for more details).

Limitation: The lateral boundaries only work for road networks where all road segments are in parallel to the x axis. However, it fully supports bidirectional scenarios, as shown in the example for Internal Boundary Control (see Section 12).

6.3 Flow Demand in Lane-Free Traffic

A major aspect of microscopic simulation is setting up demand at origin points of the network. Following the existing SUMO infrastructure, users specify demands in the route file (as discussed in Section 6.1). Each flow demand is specified for a time window through the **begin** and **end** tags, with values in seconds, and users need to assign a desired flow either with the parameter **number** or **probability**. For **number**, the exact amount of vehicles will be inserted, spread uniformly across the given time window. For **probability**, a vehicle is inserted at every time-step based on the outcome of a random sample according to this probability.

Important note: When specifying a demand with **number** in SUMO, vehicles spawn in a *deterministic* manner (uniformly spread throughout the designated time window). This can be alternatively performed in a *stochastic* manner by instead providing the **probability** value. However, the way SUMO handled this for lane-based environments would be quite limiting in lane-free settings. In the standard application, the probability value determines the *chance* that a vehicle will spawn per *second*. As such, the maximum achievable flow is 3600 veh/h. For a lane-based system, this limit is acceptable as it exceeds the maximum achievable value (per lane) under realistic conditions in a highway. However, in lane-free settings, such flows are quite limiting.

Thereby, we have **modified** this behaviour, and now probability value p determines the chance that a vehicle will spawn per *discrete time-step* T, e.g., for a time-step length $T = 0.2s$, the maximum flow ($p = 1$) would be: 18000 veh/h ($= (1\,veh)/(T\,s) \cdot (3600\,s/h)$). In general, given a desired demand value d (in veh/h), the user can calculate the corresponding probability value p by:

$$p = \frac{d \cdot T}{3600} \tag{9}$$

Additional note: When using probabilities, this relationship between probability and demand follows the probabilistic nature of random sampling, i.e., for small amount of vehicles (small demand values and/or time windows), the sampled flow will not necessarily coincide with the true mean (the specified demand d). Based on the law of large numbers, a proper inference of the true mean can be obtained with more samples (e.g., through increasing time window or averaging the flow information on the same scenario with multiple seed values).

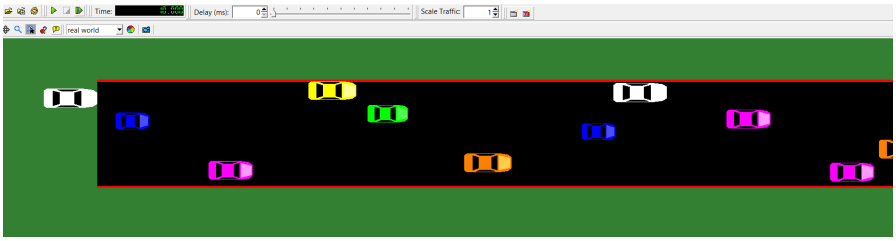
According to a demand, either through the number or probability, users can also specify *how* the vehicles should be inserted in the road, through the **insertionPolicy** option within the flow xml tag. This will select one of the available policies to insert vehicles in the road. The default policy we have developed is latExploit, which automatically inserts vehicles based on the user-specified tau (time-gap/time headway) in seconds and minGapLap (minimum lateral distance

between vehicles) in meters. These values are taken from the corresponding vehicle type to be inserted, as specified inside the .rou.xml file (see Vehicle Types in Section 6.1). The provided scenario 03_highway_flows_scenario demonstrates all different insertion policies. Users can refer to this example and have a look inside the highway.rou.xml file for proper usage:

```
<flow id="flow1" type="typedist1" begin="0" end="20" number="60" route="route0" departSpeed="25" departPos="0" />
<flow id="flow2" type="typedist1" begin="20" end="30" number="30" latLow="0.2" latHigh="0.7" route="route0"
departSpeed="25" departPos="0" />
<flow id="flow3" type="typedist1" begin="30" end="40" number="30" departSpeedLimitFrontDist="50" route="route0"
departSpeed="25" departPos="0" />
<flow id="flow4" type="typedist1" begin="40" end="50" number="20" insertionPolicy="desSpeedAlign" desSpeedLow="25"
desSpeedHigh="35" route="route0" departSpeed="25" departPos="0" />
<flow id="flow5" type="typedist1" begin="50" end="60" number="10" insertionPolicy="center" route="route0"
departSpeed="25" departPos="0" />
<flow id="flow6" type="typedist1" begin="60" end="70" number="10" insertionPolicy="platoon" platoonSize="4"
platoonTimeStepDistance="3" route="route0" departSpeed="25" departPos="0" />
```

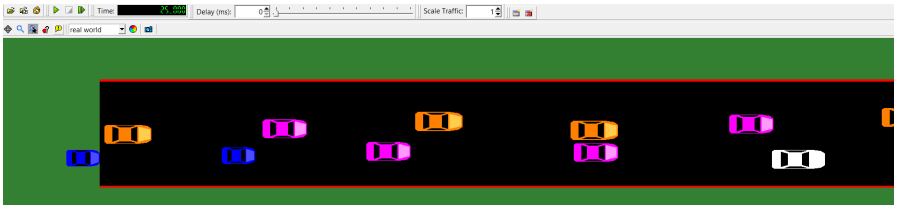
The available insertion policies are the following:

- **latExploit**: (default) If no policy is specified, this will be automatically employed. Each vehicle to be inserted will examine the road near the entry for available lateral regions that comply with the tau and minGapLat parameters. If none can be found due to high density situations, then the vehicle remains in the virtual queue. Otherwise, it randomly chooses a feasible lateral placement to enter. Entry (longitudinal) speed is the specified departSpeed parameter (in m/s). However, this can be mitigated due to slower vehicles in front in order to avoid strong decelerations. Refer to [8] for a more detailed presentation (termed as method 1 in the related Section on demand handling). See snapshot from the simulator below near the road entry for flow "flow1" from the relevant example:



Accompanying parameters to specify:

  - **latLow**, **latHigh**: By default (if not specified), these values are set as: latLow="0.0" and latHigh="1.0", corresponding to the full lateral capacity of the road. Users can restrict (per flow) entry to specific lateral parts of the road by setting these two parameters accordingly. For example, if latLow="0.0" and latHigh="0.5", then vehicles will only be inserted at the right half of the road (bottom half from the top-down view). This functionality can be useful for several use-cases, e.g., vehicles that will soon need to exit to an off-ramp. See snapshot for flow "flow2" with latLow="0.2" and latHigh="0.7":



- **departSpeedLimitFrontDist**: If this parameter is not specified (default option), then the departure speed (departSpeed parameter) of the vehicle to be inserted will be affected according to the average speed (avgSpeedFront) of the first 5 vehicles from the entry point. Then departSpeed=min(avgSpeedFront,departSpeed). If specified, then the average speed measurement will take into account all vehicles within the first departSpeedLimitFrontDist meters of the road.
- **departSpeedLimitDownstream**: (for more experienced users, **we do not suggest disabling this option**) This is a boolean (takes true/false or 1/0 values) parameter, and it controls whether the departSpeed will be affected by the neighboring vehicles. By default (if not specified), this functionality is enabled, as it allows for an adaptive departure speed that also better correspond to a real-life scenario and avoids many unnecessary situations due to speed deviations of vehicles. However, we provide this "switch" for users that do not want the departSpeed to be affected by our mechanism. If set to false, users should proceed with caution as it does not fare well in situations with high demands.

- **desSpeedAlign**: If this policy is specified, then vehicles are inserted laterally according to their desired speed objective. A linear interpolation is established between the two lateral limits of the road and the lower and higher desired speed objective (desSpeedLow and desSpeedHigh parameters). As such, the slowest (fastest) vehicles are spawned at the rightmost (leftmost) part of the road with desSpeedLow (desSpeedHigh) desired speed objective. Again, vehicles will follow the specified tau (time-gap) and minGapLat (lateral distance) gaps in order to be spawned. In addition, this insertion policy spreads the vehicles laterally according to a uniform distribution. Refer to [8] for more details (termed as method 2 in the related Section on demand handling).

- **center**: Vehicles can be spawned only at the center of the road, based on a time-gap/time-headway policy (according again to the tau variables). This is useful for demands on an on-ramp, where the lateral capacity of the road is limited, and makes more sense to only insert new vehicles at the road's center. See snapshot below from the simulator for the associated flow "flow5".

- **platoon**: If this policy is selected, we essentially again rely on the latExploit insertion policy. However, we now only insert *platoons of vehicles*. Therefore, the number or probability values now reflect the number of platoons to be entered (and not individual vehicles). As such, **make sure to properly calculate these values corresponding to the associated demand** based on the selected platoon's size. Additional parameters to be specified:
  - **platoonSize**: The number of vehicles in each platoon
  - **platoonTimeStepDistance**: The number of time-steps between spawning a new tailing vehicle in the platoon

  See snapshot below from the simulator for the associated flow "flow6".

Users can refer to Section 15 for a detailed usage example.

Important note: One can specify multiple flows that are potentially overlapping in time (e.g., a platoon demand alongside a simple latExploit for individual vehicles, or multiple simple latExploit but on different lateral parts of the road). We do not have any limitation on this, but make note that the virtual queue of vehicles to be spawned will append each vehicle that could not find a proper place to enter, one that respect the designated gaps. The virtual queue can be monitored through the main application via the "insertion-backlogged-vehicles" parameter in the Network window. This measurement is also available per road edge. Refer to Section 8.1 for more details.

7 Online Traffic Network Measurements

A part of the available functions in the API can be used to get online measurements from the road network. Below, we detail the usage for each type of measurement:

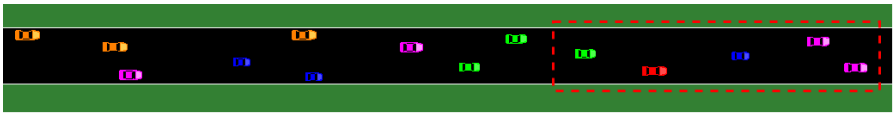
7.1 Density Measurements

We calculate density measurements based on the number of vehicles per kilometer (veh/km). There is a list of available functions inside the API with associated comments regarding usage:

```
// returns the density of vehicles (veh/km) for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT double (*get_density_on_segment_region_on_edge)(NumericalID edge_id, double segment_start, double segment_end);


// returns the number of vehicles # for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT int (*get_number_of_vehicles_on_segment_region_on_edge)(NumericalID edge_id, double segment_start, double segment_end);


// returns the density of vehicles (veh/km) only on the main highway for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT double (*get_density_on_segment_region_on_edge_only_highway)(NumericalID edge_id, double segment_start, double segment_end);


// returns the number of vehicles only on the main highway for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT int (*get_number_of_vehicles_on_segment_region_on_edge_only_highway)(NumericalID edge_id, double segment_start, double segme


// returns the density of vehicles (veh/km) only on the ramp (either on-ramp or off-ramp) for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT double (*get_density_on_segment_region_on_edge_only_ramp)(NumericalID edge_id, double segment_start, double segment_end);


// returns the number of vehicles only on the ramp (either on-ramp or off-ramp) for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT int (*get_number_of_vehicles_on_segment_region_on_edge_only_ramp)(NumericalID edge_id, double segment_start, double segment_


// returns the density of vehicles (veh/km) per segment for a given edge id, and a segment length
// E.g., given an edge with length 1000m and segment length of 100m, this will return a double array of 10 elements, associated with each 100m segment
// if (edge_length modulo segment_length) != 0, then the resulting array will contain an additional segment at the end associated with the remaining distanc
EXTERN libLaneFreePlugin_EXPORT double* (*get_density_per_segment_per_edge)(NumericalID edge_id, double segment_length);
// returns the size of the array provided above
EXTERN libLaneFreePlugin_EXPORT int (*get_density_per_segment_per_edge_size)(NumericalID edge_id, double segment_length);


// returns the density of vehicles (veh/km) for a given segment region for a given edge id and for a given type of vehicles
EXTERN libLaneFreePlugin_EXPORT double (*get_density_on_segment_region_on_edge_for_type)(NumericalID edge_id, double segment_start, double segment_end, char
```

Note that all density-related measurements are provided **per road edge**, where the user needs to specify the NumericalID of the desired edge. When designing a road network for a scenario (the .net.xml file), users have specified each edge name in a string format, and SUMO automatically assigns a unique NumericalID value to each edge when the network is loaded. In practice, users can utilize the get_edge_name API function that returns the string information (in char*) when iterating over each edge id to obtain this information.

Measurements in Specific Regions Another common element across most available functions is the inclusion of arguments **segment_start** and **segment_end**. These are specified in meters, and dictate the region within the road edge that this information should be measured from, e.g., in the example below



if we assume that what is displayed is the total road segment, its length is 150 meters, and we are interested only in the last 50 meters, we define the region as segment_start=100, segment_end=150. If we are interested in the whole road edge, we simply specify segment_start=0, segment_end=get_edge_width(edge_id).

Important Information:

- The same calculations are performed internally for get_density_on_segment_region_on_edge and get_number_of_vehicles_on_segment_region_on_edge. The only differentiation is whether the return value is divided according to the length of the specified segment (to have proper density value in veh/km) or users are simply interested in the number of vehicles within that region, respectively.
- In road segments containing more than one lane (e.g., on-ramps, off-ramps in the context of lane-free traffic), the standard functions above take measurements from the whole road width. If interested in distinguishing these measurements depending on whether vehicles are on the main highway or on the ramp, users can utilize the functions (as shown above) with the suffix **_only_highway** or **_only_ramp**, respectively.
- The **get_density_per_segment_per_edge** is convenient if interested to partition the road in consecutive regions of equal length. As such, it returns an array of density measurements according to the partitioning of the road given the segment_length input argument.
- Users can also request density information **per vehicle type**. This is accomplished with the function get_density_on_segment_region_on_edge_for_type and the additional input argument of the vehicle type name (corresponding to the type name defined in the .rou.xml file) in char*.

7.2 Speed Measurements

Speed information for a specified road region can be obtained by averaging the current longitudinal speed of the vehicles currently within that prescribed region. The following functions are available:

```
// returns the average speed of vehicles for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT double (*get_average_speed_on_segment_region_on_edge)(NumericalID edge_id, double segment_start, double segment_end);


// returns the average speed of vehicles (m/s) only on the ramp (either on-ramp or off-ramp) for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT double (*get_average_speed_on_segment_region_on_edge_only_highway)(NumericalID edge_id, double segment_start, double segment_end);


// returns the average speed of vehicles (m/s) only on the ramp (either on-ramp or off-ramp) for a given segment region for a given edge id
EXTERN libLaneFreePlugin_EXPORT double (*get_average_speed_on_segment_region_on_edge_only_ramp)(NumericalID edge_id, double segment_start, double segment_end);


// returns the average speed of vehicles for a given segment region for a given edge id and for a given type of vehicles
EXTERN libLaneFreePlugin_EXPORT double (*get_average_speed_on_segment_region_on_edge_for_type)(NumericalID edge_id, double segment_start, double segment_end, char* v
```

Important Note: Refer to the instructions in Section 7.1 above for density measurements. The same principles are applicable for the available speed measurement function calls above as well.

7.3 Flow Measurements with Detectors

For online flow measurements through the API, this is accomplished by placing **loop detectors** in the network. We follow the infrastructure of SUMO for this,[13] meaning that detectors are placed through an *additional-file* (as called in SUMO), the existence of which is mentioned when discussing the Configuration file format (see Section 6.1). As an example, we include the 04_highway_detectors_scenario, where the detectors.add.xml file now contains 2 detectors, as shown below:

```
<additional xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/additional_file.xsd">
    <e1Detector id="detec_0" lane="long_edge_0" pos="20.00" freq="900.00"
    file="e1Detector_detec_0.xml"/>
    <e1Detector id="detec_1" lane="long_edge_0" pos="120.00" freq="900.00"
    file="e1Detector_detec_1.xml"/>
</additional>
```

There, the tag e1Detector (as in SUMO) is used for each detector to be specified. Then, we need to define:

1. **id**: A unique string id.
2. **lane**: The lane name (as defined in the network file .net.xml). Note that the naming of lanes is generated automatically by adding a suffix with the lane index (starting from the rightmost lane). For example, if edge "long_edge" contained two lanes, then "long_edge_0" would be the right lane, and "long_edge_1" would be the left lane (indexing starts from the bottom with respect to a top down view). Therefore, detectors measure the flow per lane. This is important for applications with on-/off-ramps where we want to have separate measurements for ramps and the main highway. Of course, one could simply accumulate these values accordingly to obtain the total flow.
3. **pos**: This is the longitudinal position in meters (with respect to the beginning of the lane) for the detector to be placed. Note that the system checks for improper values that exceed the lane's length. If so, the scenario will not load properly.
4. **freq**, **file**: The frequency of measurements (in seconds) for the output file with the filename according to the second argument. This output file will contain flow measurements based on the prescribed frequency. This feature directly stems from the existing SUMO application, without any influence from our part. **Note that** this does not affect in any way the online measurements of the API. As such, any arbitrary (but feasible) frequency value would suffice for users not actually interested in this feature of SUMO.

If we load the scenario, detectors will be visualized as follows:

[13] Refer to https://sumo.dlr.de/docs/Simulation/Output/Induction_Loops_Detectors_%28E1%29.html for more details

Notably, we have updated the visualization of detectors with a more minimal and unobtrusive look with respect to the one in SUMO.[14]

Online Measurements from Detectors through the API The available functions for detectors through the API in the code are the following:

```
// returns the ids of the detectors
EXTERN libLaneFreePlugin_EXPORT NumericalID* (*get_detectors_ids)();
// returns the size of the ids of the detectors
EXTERN libLaneFreePlugin_EXPORT NumericalID(*get_detectors_size)();


// returns the detector's name, based on the detector id
EXTERN libLaneFreePlugin_EXPORT char* (*get_detector_name)(NumericalID detector_id);


// returns the values of all detectors (number of vehicles for each detector) (NOTE: every time we obtain a measurement from detectors, they are reset)
EXTERN libLaneFreePlugin_EXPORT int* (*get_detectors_values)();


// returns the value of a single detector, based on the detector's id (NOTE: every time we obtain a measurement from detectors, they are reset)
EXTERN libLaneFreePlugin_EXPORT int (*get_detector_value)(NumericalID detector_id);


// returns the value that corresponds to the specified type of a single detector, based on the detector's id (NOTE: every time we obtain a measurement
EXTERN libLaneFreePlugin_EXPORT int (*get_detector_value_for_type)(NumericalID detector_id, char* veh_type);
```

Furthermore, the main file LaneFree.cpp in the lanefree_plugin, contains the following code example:

```
int detector_frequency_measurements = 1000; //in time-steps

if (get_current_time_step() % detector_frequency_measurements == 0) {
    NumericalID* detector_ids = get_detectors_ids();
    int* detector_values = get_detectors_values();

    NumericalID detectors_size = get_detectors_size();

    char* detector_name;
    for (j = 0; j < detectors_size; j++) {
        detector_name = get_detector_name(detector_ids[j]);
        printf("detector:%s count:%d\n", detector_name, detector_values[j]);
    }
}
```

where we first obtain the array of detector ids (get_detectors_ids), its size, and take a measurement from each detector (get_detectors_values) again in an array form. Then we iterate over the array, get the string name of each detector (get_detector_name) as defined in the additional file; and finally, print each name and value pair.

Important Notes:

- The detectors return the **count** of vehicles. As such, the flow (veh/h) should be calculated according to the frequency that the user specifies.
- There are 3 different function calls to get detector measurements, namely:

[14] See https://sumo.dlr.de/docs/Simulation/Output/Induction_Loops_Detectors_%28E1%29.html#visualisation

  - get_detectors_values: returns an array with one measurement per detector (for all detectors in the scenario)
  - get_detector_value(detector_id): returns a scalar with one measurement only for the detector with id detector_id
  - get_detector_value_for_type(detector_id, veh_type): returns a scalar with one measurement only for detector with id and **only for the vehicle type** with name (in char*) veh_type, corresponding to the vehicle type name that is defined in the route file (.rou.xml).

- The detector's counter **resets** after each get function (for all 3 ways above). Note that the counter for each vehicle type is independent from the main counter of the detector, i.e., the _for_type function has multiple counters (one per vehicle type) and is reset only from the corresponding function call. In contrast, the general counter of a detector is reset either from the get_detectors_values or get_detector_value(detector_id).

8 Available Metrics

In this section, we specify all accessible metrics (either already available from SUMO that are applicable in lane-free settings, along with new ones we have introduced) and provide accompanying details regarding usage as well. We can distinguish two types of metrics that are available to users, namely:

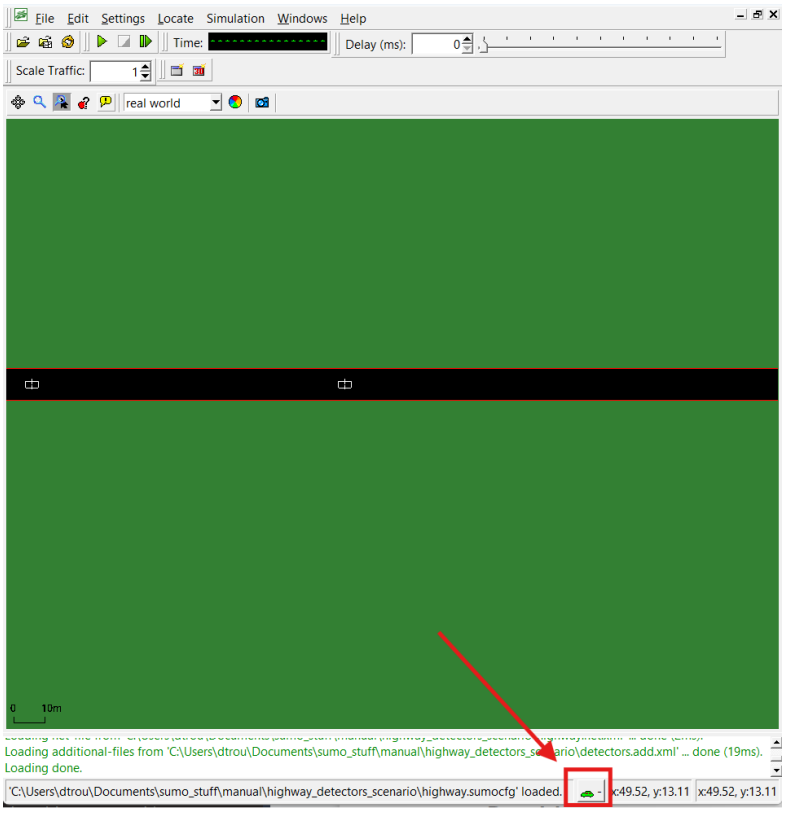
- **Online Simulation Metrics in GUI**: These are available through the GUI application window. Refer to Section 8.1.
- **Post Simulation Metrics in Logfiles**: These are available post-simulation through logfiles. Detailed instructions can be found in Section 8.2.

8.1 Online Simulation Metrics in GUI

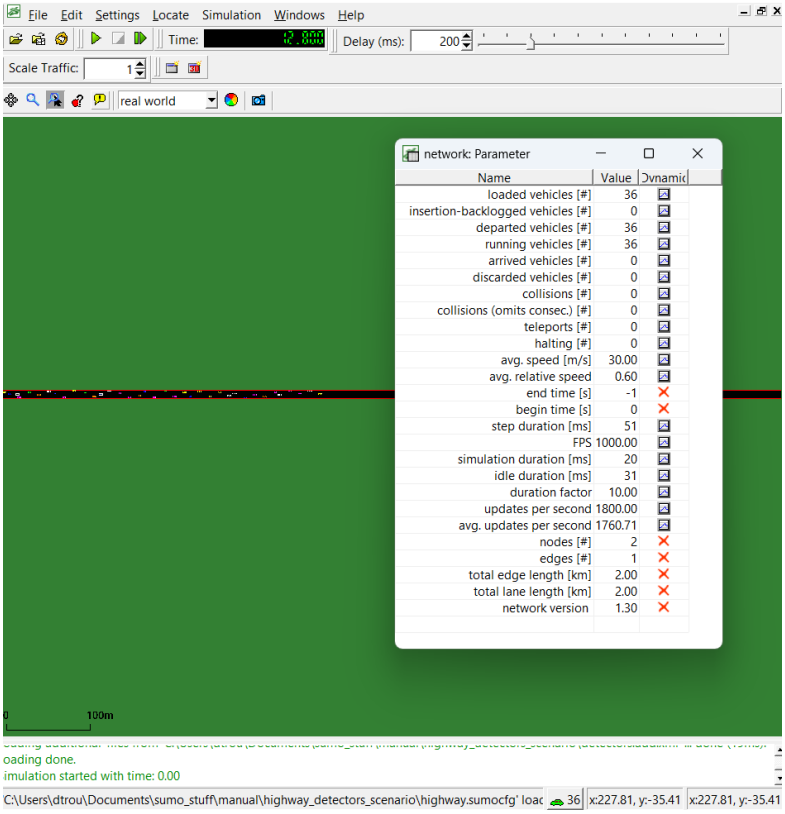
Within the GUI application window, we leverage the existing SUMO infrastructure for online metrics through available parameters,[15] at 3 different levels, namely:

1. **Overall road network**: Once we have loaded a scenario, we can press the button as shown below:

[15] Refer also to https://sumo.dlr.de/docs/sumo-gui.html for additional information


File Edit Settings Locate Simulation Windows Help
Time:
Delay (ms): 0
Scale Traffic: 1
real world
0 10m
Loading additional-files from 'C:\Users\dtrou\Documents\sumo_stuff\manual\highway_detectors_scenario\detectors.add.xml' ... done (19ms).
Loading done.
'C:\Users\dtrou\Documents\sumo_stuff\manual\highway_detectors_scenario\highway.sumocfg' loaded.
x:49.52, y:13.11
x:49.52, y:13.11

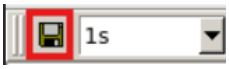
Then, we see the parameter window on top of the simulation:



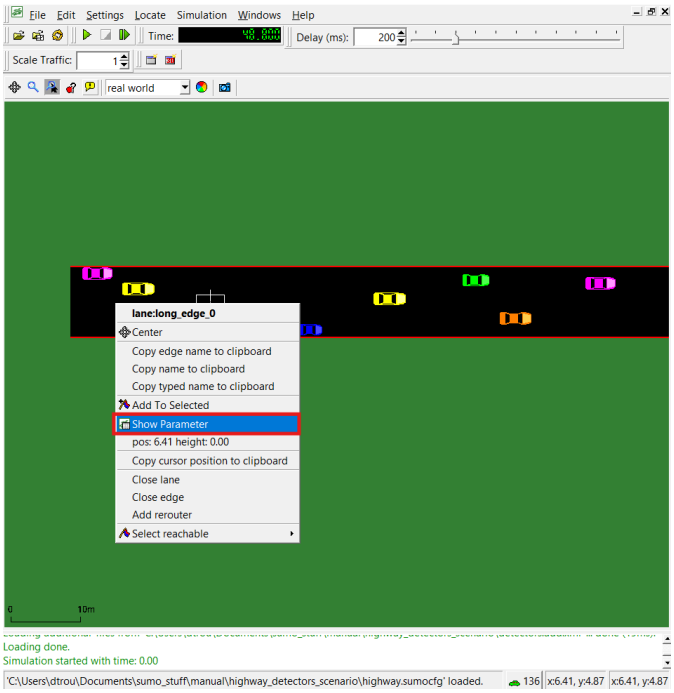
The naming suggests the meaning of each value. Notably, the "insertion-backlogged vehicles" is the one indicating the number of vehicles currently in the virtual queue of SUMO due to unhandled demand at entry points (i.e., vehicles that should enter the road but could not due to limited availability of the road). All parameters stem from the existing application, with the exception of "collisions (omits consec.)" that we included. This parameter measures collision occurrences but only counts the first time-step that two vehicles collided. In addition the running vehicles information (vehicles currently on the road) is displayed at the bottom part, within the button that launches this window.

Furthermore, parameters that are *Dynamic* (they can change over time) have a button (instead of the × mark) on the right side. This will launch a new window that automatically creates a plot according to these values over time. See related information in:https://sumo.dlr.de/docs/sumo-gui.html#plotting_object_properties.

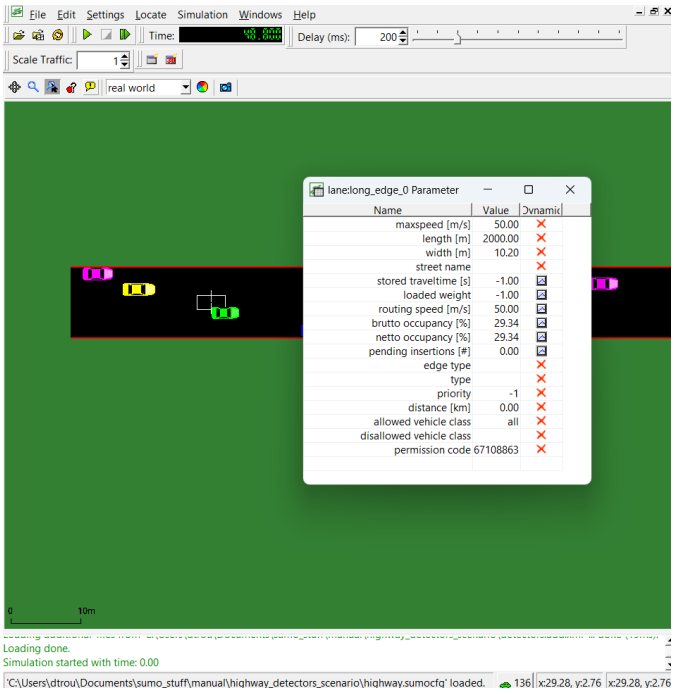
Important Note: All these online plots can be directly logged in a simple .csv file format for post processing through the save button that appears on top.



2. **Road lane**: Likewise, we can navigate to a lane, and with a right-click anywhere on the road (but not on top of a vehicle):

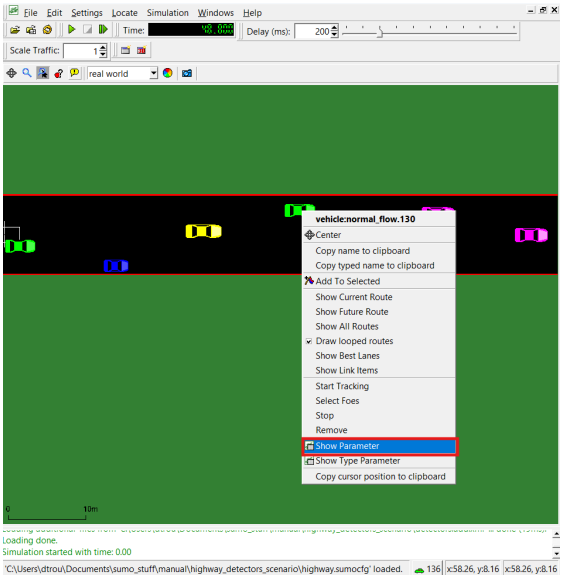


and launch the parameter window per lane:

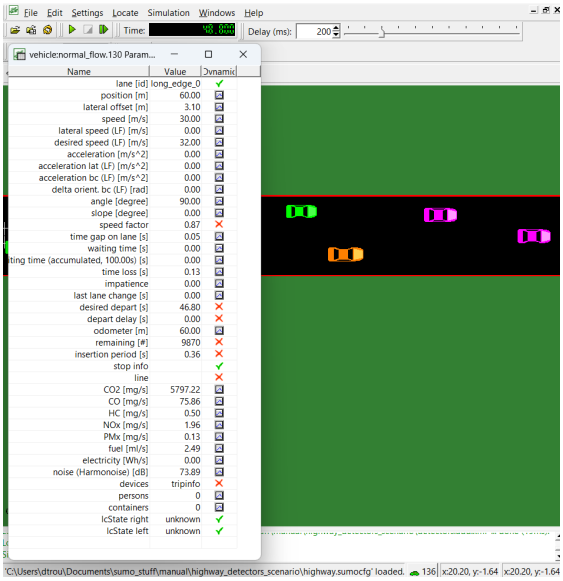


where apart from the lanes characteristics (length, width), an important parameter relevant to flow demand handling is the "pending insertions [#]", which measures the virtual queue of vehicles not being able to be inserted in the road for the specific lane.

3. **Vehicle**: Finally, we can focus on a specific vehicle, by simply right-clicking on top of it:



Important Note: The "Start Tracking" functionality is commonly used to visually track the movement of the pointed vehicle. Essentially, the GUI locks to the center of the vehicle as it moves. This is heavily utilized e.g., for video purposes to exemplify a vehicle movement strategy.



The set of parameters show the online state of the vehicle. The parameters with the (LF) indication at the end are the ones that we included and are relevant to lane-free vehicle movement. Additionally, the parameters including the bc term in their name are relevant only for vehicles operating with the bicycle model. Note that several parameters of SUMO

assuming a lane-based system are now not relevant and contain values that a) do not have any importance, e.g., last lane change, or b) the different nature of lane-free movement can result in measurements for default variables that are outright mishandled.

8.2 Post Simulation Metrics in Logfiles

Additionally, we provide the option to generate log files capturing metrics for the entire simulation run, available upon the completion of a scenario. These logs include detailed information for individual vehicles as well as aggregated metrics and overall summaries of the simulation.

Two distinct types of output log files are available:

- **TripInfo**: This log file follows the structure of the original *TripInfo* output format. It records essential details about the trip of each vehicle, such as departure and arrival times, associated lanes, and other relevant data. Comprehensive documentation for all fields included in this log file is available at https://sumo.dlr.de/docs/Simulation/Output/TripInfo.html. An example of a *TripInfo* log from an executed scenario is provided below:

```
<tripinfos xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd
/tripinfo_file.xsd">
    <tripinfo id="normal_flow_0_straight.0" depart="0.00" departLane="EN0_2" departPos="0.00" departSpeed="10.00"
    departDelay="0.00" arrival="18.00" arrivalLane="EX2_2" arrivalPos="84.50" arrivalSpeed="12.00" duration="18.00"
    routeLength="200.00" waitingTime="0.00" waitingCount="0" stopTime="0.00" rerouteNo="0" devices="
    tripinfo_normal_flow_0_straight.0" vType="lane_free_car3" speedFactor="0.82" delay="11.91" vaporized=""/>
    <tripinfo id="normal_flow_1_straight.0" depart="5.00" departLane="EN1_2" departPos="0.00" departSpeed="10.00"
    departDelay="0.00" arrival="23.00" arrivalLane="EX3_2" arrivalPos="100.00" arrivalSpeed="12.00" duration="18.00"
    routeLength="231.00" waitingTime="0.00" waitingCount="0" stopTime="0.00" rerouteNo="0" devices="
    tripinfo_normal_flow_1_straight.0" vType="lane_free_car11" speedFactor="1.08" delay="12.10" vaporized=""/>
    <tripinfo id="normal_flow_0_straight.1" depart="10.00" departLane="EN0_2" departPos="0.00" departSpeed="10.00"
    departDelay="0.00" arrival="28.00" arrivalLane="EX2_2" arrivalPos="84.50" arrivalSpeed="12.00" duration="18.00"
    routeLength="200.00" waitingTime="0.00" waitingCount="0" stopTime="0.00" rerouteNo="0" devices="
    tripinfo_normal_flow_0_straight.1" vType="lane_free_car6" speedFactor="1.14" delay="12.11" vaporized=""/>
    <tripinfo id="normal_flow_2_straight.0" depart="12.00" departLane="EN2_2" departPos="0.00" departSpeed="10.00"
    departDelay="0.00" arrival="30.00" arrivalLane="EX0_2" arrivalPos="100.00" arrivalSpeed="12.00" duration="18.00"
    routeLength="231.00" waitingTime="0.00" waitingCount="0" stopTime="0.00" rerouteNo="0" devices="
    tripinfo_normal_flow_2_straight.0" vType="lane_free_car6" speedFactor="1.11" delay="10.29" vaporized=""/>
    <tripinfo id="normal_flow_1_straight.1" depart="15.40" departLane="EN1_2" departPos="0.00" departSpeed="10.00"
    departDelay="0.07" arrival="33.40" arrivalLane="EX3_2" arrivalPos="100.00" arrivalSpeed="12.00" duration="18.00"
    routeLength="231.00" waitingTime="0.00" waitingCount="0" stopTime="0.00" rerouteNo="0" devices="
    tripinfo_normal_flow_1_straight.1" vType="lane_free_car7" speedFactor="0.91" delay="11.74" vaporized=""/>
    <tripinfo id="normal_flow_0_straight.2" depart="20.00" departLane="EN0_2" departPos="0.00" departSpeed="10.00"
    departDelay="0.00" arrival="38.00" arrivalLane="EX2_2" arrivalPos="84.50" arrivalSpeed="12.00" duration="18.00"
    routeLength="200.00" waitingTime="0.00" waitingCount="0" stopTime="0.00" rerouteNo="0" devices="
    tripinfo_normal_flow_0_straight.2" vType="lane_free_car7" speedFactor="0.93" delay="11.30" vaporized=""/>
</tripinfos>
```

- **Statistic**: This log file also follows the original *Statistic* output format but includes several additional fields. Its primary purpose is to provide a holistic summary of the simulation, including metrics on vehicle and pedestrian dynamics, safety indicators, trip statistics, and overall performance measures. This output is designed to evaluate the behavior and efficiency of the entire simulation rather than focusing on individual vehicles. While most definitions are available in the official documentation at https://sumo.dlr.de/docs/Simulation/Output/StatisticOutput.html, we additionally provide the following:
  - delayAvg (s): average delay of vehicles based on the total expected time. The latter is computed by dividing the total route length by each vehicle's desired speed. Only exited vehicles are included in these metrics.
  - delayAvgNoNeg (s): positive-only average delay of vehicles based on the total expected time (this includes only the vehicles slower than their desired speed).
  - delayAvgOnlyNeg (s): negative-only average delay of vehicles based on the total expected time (this includes only the vehicles faster than their desired speed).

– totalTimeSpent_hours ($veh \cdot h$): measures the accumulated travel time summed over all vehicles introduced to the simulation.

An example of a *Statistic* log from an executed scenario is provided below:

```
<statistics xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/
xsd/statistic_file.xsd">
    <vehicles loaded="124" inserted="124" running="0" waiting="0"/>
    <teleports total="0" jam="0" yield="0" wrongLane="0"/>
    <safety collisions="33" emergencyStops="0"/>
    <persons loaded="12" running="0" jammed="0"/>
    <vehicleTripStatistics routeLengthAvg="214.59" speedAvg="9.18" durationAvg="27.33" waitingTimeAvg="0.26"
    departDelayAvg="0.05" departDelayWaiting="-1" delayAvg="20.45" delayAvgNoNeg="20.45" delayAvgOnlyNeg="0.00"
    totalTimeSpent_hours="0.94"/>
    <pedestrianStatistics number="12" routeLengthAvg="169.38" durationAvg="144.42" timeLossAvg="14.60"/>
    <rideStatistics number="0"/>
    <transportStatistics number="0"/>
</statistics>
```

The filenames for these log files can be customized by specifying the desired names in the corresponding parameters of the .sumocfg file for the scenario (*tripinfo-output* for the *TripInfo* log and *statistic-output* for the *Statistic* log):

```
1  <configuration>
2     <input>
3         <net-file value="highway.net.xml"/>
4         <route-files value="highway.rou.xml"/>
5         <gui-settings-file value="gui-settings.cfg"/>
6         <additional-files value="detectors.add.xml"/>
7     </input>
8     <processing>
9           <step-method.ballistic value="true"/>
10          <collision.action value="none"/>
11          <step-length value="0.2"/>
12          <seed value="0"/>
13          <video-record-or-replay value="nothing"/>
14    </processing>
15    <output>
16       <tripinfo-output value="trip_info.xml"/>
17       <statistic-output value="avg_stats.xml"/>
18       <video-logfile value="recording_file.log"/>
19    </output>
20 </configuration>
```

9 Store and Replay Scenarios

In some scenarios, the runtime performance can be significantly affected by the computational demands of complex controllers or the limited processing power of low-end hardware. This often results in slow or unstable simulations with noticeable stuttering. While these issues do not affect the accuracy of the simulation or the behavior of the agents, they can be problematic when trying to present the simulation in real time, such as for creating video recordings or monitoring controller performance visually.

To address this, we have introduced a feature that allows the behavior of a scenario execution to be recorded for subsequent replay by storing data. This replay mechanism bypasses

the need for controller inputs or computations during playback, thereby creating a lightweight reproduction of the exact scenario execution.

As a reminder we provide an example of a .sumocfg scenario file:

```
1  <configuration>
2     <input>
3         <net-file value="highway.net.xml"/>
4         <route-files value="highway.rou.xml"/>
5         <gui-settings-file value="gui-settings.cfg"/>
6         <additional-files value="detectors.add.xml"/>
7     </input>
8     <processing>
9           <step-method.ballistic value="true"/>
10          <collision.action value="none"/>
11          <step-length value="0.2"/>
12          <seed value="0"/>
13          <video-record-or-replay value="nothing"/>
14    </processing>
15    <output>
16       <tripinfo-output value="trip_info.xml"/>
17       <statistic-output value="avg_stats.xml"/>
18       <video-logfile value="recording_file.log"/>
19    </output>
20 </configuration>
```

To enable recording or replaying a scenario execution, the following configuration steps must be performed:

- **Recording a Scenario**: Set the .sumocfg parameter *video-record-or-replay* (line 13) to "record". This will ensure that all vehicle actions are logged in the file specified by the output parameter *video-logfile* (line 18).
- **Replaying a Scenario**: Change the value of *video-record-or-replay* (line 13) to "replay". This will replicate the actions stored in the *video-logfile* (line 18) with minimal computational overhead or delay.

Any other value of the parameter *video-record-or-replay* (such as the default value of "nothing") will result in a standard execution of the scenario without enabling recording or replay functionality.

Important Notes:

- If the underlying controllers driving the vehicles incorporate any stochastic elements, these will be precisely replicated during replay, regardless of how many times the same stored data is replayed. To examine different outcomes under varying conditions, multiple recordings should be generated.
- Bypassing controller inputs during replay disables a significant portion of the underlying simulation code. While this feature ensures the visual validity of the simulation, it should not be relied upon for generating log outputs.
- For Internal Boundary Control application (see Section 12), the vehicles' movement in replay mode will properly follow the trajectories based on the moving boundaries. However, online visualization of the boundaries moving is not supported in this mode.

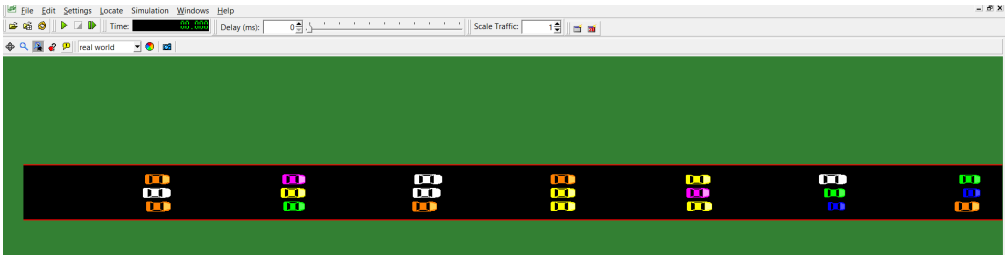
10 Ring-Road

In TrafficFluid-Sim, we emulate ring-road scenarios in highway environments by placing vehicles about to exit accordingly at the beginning of the road. In this case, the vehicles are directly "transmitted" to the starting point of the segment instead of exiting. Longitudinal distance $d_x(i, j)$ information is properly adjusted for this feature, e.g., a vehicle towards the end of the road segment will properly observe vehicles at the starting point through the API function get_relative_distance_x(ego_id, other_id). In order to introduce vehicles in the ring-road, the recommended solution is to utilise the API instead of flow demand at origin points (due to the nature of the ring-road environment), and insert the desired density of vehicles when initializing the simulation within the simulation_initialize function. As such, we contain the 05_ring_road_emulation_scenario, which is essentially the highway scenario, but without the inclusion of flow demands. Changes for ring-road settings involve updating the code accordingly, as we show below.

We contain an example in the lanefree_plugin code, within the simulation_initialize, on how to place vehicles on the road with initial conditions:

```
void simulation_initialize(){

    //initialize srand with the same seed as sumo
    srand(get_seed());

    //insert n_init vehicles
    int n_init = 0;

    // define the spacing in the x axis through x_incr, and y axis through y_incr
    // and whether the initial speed vx will be different as we insert more rows through vx_incr
    double x_incr=25, y_incr=2.5, vx_incr=5;

    // initialize the x,y, vx values for the first vehicle to be inserted
    double x_val=x_incr, y_val=y_incr, vx_val=vx_incr;
    // number of virtual lanes
    int virtual_lanes = 3;
    // considered road width
    double width=10;

    char veh_name[40];
    //route_id and type_id should be defined in the scenario we are running
    char route_id[20]="route0";
    char type_id[20]="typedist1";
    NumericalID v_id;
    for(int i=0;i<n_init;i++){
        sprintf(veh_name, "%s_plugin_%d", type_id,(i+1));
        v_id = insert_new_vehicle(veh_name, route_id, type_id, x_val, y_val, vx_val,0,0,0);
        printf("%s inserted\n", veh_name);
        y_val = y_val + y_incr;
        if(i%virtual_lanes==(virtual_lanes-1)){
            x_val += x_incr;
            if (vx_val < 35) {
                vx_val += vx_incr;
            }
            y_val = y_incr;
        }
    }
```

Through this code, we insert vehicles at the first time-step of the simulation and place them in rows according to the specified spacing. See below a snapshot of the simulation at the first step (when setting n_init=50):



A **necessary** change in the code is to instruct vehicles upon entry to follow this ring-road behaviour instead of exiting. This is accomplished within the event_vehicle_enter function, and by setting the set_circular_movement function argument accordingly:

```
void event_vehicle_enter(NumericalID veh_id){

    set_desired_speed(veh_id, MIN_DESIRED_SPEED + (((double)rand()/RAND_MAX) * (MAX_DESIRED_SPEED - MIN_DESIRED_SPEED)));
    //char* vname1 = get_vehicle_name(veh_id);
    // printf("Vehicle %s entered with speed %f.\n",vname1,get_speed_x(veh_id));

    //make the vehicles emulate a ring road scenario
    set_circular_movement(veh_id, false);

}
```

We should replace with: set_circular_movement(veh_id,**true**).

Important Note: Delay-related metrics are now unsuitable for this scenario.

11 On-Ramps and Off-Ramps

A scenario example containing both an off-ramp and an on-ramp, with vehicles following all possible routes can be found in 06_highway_on_off_ramps_scenario folder.

11.1 Scenario Setup

There, users need to have a proper look at the newtwork (.net.xml) file to observe how off-ramps and on-ramps can be added. This can be done either directly through the xml file, or by making use of the netedit tool, as discussed in Section 6.1.

The specific scenario is a highway that contains one off-ramp and one on-ramp, therefore all vehicles to be inserted can follow one of the three possible routes. A small-scale setup of the network is shown below:

To integrate ramps within a highway setting, we need to properly make use of lateral boundaries, as presented in Section 6.2. In this example, we have a more complex setup containing multiple overlapping routes (e.g. vehicles exiting from an off-ramp while others exit from the main highway), and therefore overlapping lateral boundaries. In the .rou.xml file of this example, we first define the 3 routes corresponding to the three possible paths in our network, and then the corresponding flow demands (refer to Section 6.3) for the initial time window $[0, 300]s$.

```
<route id="main_highway"  edges="warm_up segment1 segment2 segment3 segment4 segment5 segment6"
leftBoundaryLevelPoints="0 0 0 0 0 0 0" leftBoundaryOffsets="500 1000 1500 2000 2500 3000" leftBoundarySlopes="0.045
0.045 0.045 0.045 0.045 0.045" rightBoundaryLevelPoints="-10.2 -10.2" rightBoundaryOffsets="1020" rightBoundarySlopes
="0.01" rightBoundaryConstant="-10.2"/>
<route id="on_ramp_to_highway"  edges="segment5  segment6"  leftBoundaryLevelPoints="-10.2 0 0" leftBoundaryOffsets="
2800 3000" leftBoundarySlopes="0.01 0.01" rightBoundaryLevelPoints="-13.2 -10.2" rightBoundaryOffsets="2850"
rightBoundarySlopes="0.007" rightBoundaryConstant="-10.2"/>
<route id="exit_from_off_ramp"  edges="warm_up segment1 segment2" leftBoundaryLevelPoints="0 0 0 -10.2"
leftBoundaryOffsets="500 1000 1140" leftBoundarySlopes="0.01 0.01 0.01" rightBoundaryLevelPoints="-10.2 -13.2"
rightBoundaryOffsets="1100" rightBoundarySlopes="0.05" rightBoundaryConstant="-10.2"/>


<flow id="highway" type="typedist1" begin="0" end="300" probability="0.02" departPos="0" departSpeed="30" latLow="0"
latHigh="1" route="main_highway">
</flow>

<flow id="on_ramp_highway" type="typedist1" begin="0" end="300" probability="0.0001" departPos="0" departLane="0"
departSpeed="25" latLow="0" latHigh="1" route="on_ramp_to_highway">
</flow>

<flow id="highway_off_ramp" type="typedist1" begin="0" end="300" probability="0.0001" departPos="0" departSpeed="30"
latLow="0" latHigh="0.2" route="exit_from_off_ramp">
</flow>
```

Important Note: We remind the reader that the lateral boundaries can be visualized through the relevant parameter (see Section 6.2).

Other Details: We can visualize the acceleration area (and deceleration area if needed) on an on-ramp through the *additional* file format, based on existing functionalities of SUMO. Specifically, the additional (add.xml) file of the example now contains a white line below the definitions of loop detectors through the "poly"[16] xml tag, as follows:

```
    <e1Detector id="1-s5" lane="segment6_0" pos="3.01" freq="900.00" file="e1Detector_segment6_0_8.xml"/>
    <e1Detector id="1-s6" lane="segment6_0" pos="491.09" freq="900.00" file="e1Detector_segment6_0_10.xml"/>
    <poly id="on_ramp_1_line" color="white" fill="0" lineWidth="0.10" layer="128.00" shape="2505.000000,-10.200000
    2650.000000,-10.200000"/>
</additional>
```

where the shape parameter determines the two points (x_1, y_1) and (x_2, y_2) (on the global x-y coordinates of SUMO) that form the following line in the network:

11.2 Controller/Lane-Free Plugin Setup

Typically, we examine these scenarios with the (double) double integrator model (see Section 5.3.1) for vehicle movement strategies. As such, users need additionally to *ensure* that vehicles **follow the lateral boundaries** of their routing scheme.

As discussed in Section 6.2, distance information is provided both from the left and right lateral boundaries. Hence, a boundary controller can be developed for the vehicles to follow. Within our code example (LaneFree.cpp file in the lanefree_plugin folder) below, we contain a simple lateral boundary controller based on the implementation in [9].

[16] See: https://sumo.dlr.de/docs/Simulation/Shapes.html

```
// As a simple measure, we check the distance to the boundary at 50 meters ahead. This could be more elaborate, e.g., based on time headway.
double longitudinal_distance_to_look = 50;
get_distance_to_road_boundaries_at(myids[i], longitudinal_distance_to_look, 0, &left_boundary_dist, &right_boundary_dist, NULL, NULL, NULL);
double target_y = pos_y;
double y_const = 0.2;


// We assume that the lateral space is always large enough, i.e., the vehicle cannot overlap both boundaries at the same time.
if (left_boundary_dist < width / 2.) {
    // we target the lateral position that will make this condition false, and thus make the vehicle follow the boundary. A small constant te
    target_y = pos_y - ((width / 2.) - left_boundary_dist) - y_const;
}
else if (right_boundary_dist < width / 2.) {
    target_y = pos_y + ((width / 2.) - right_boundary_dist) + y_const;
}
double K_p=1, K_d=2;

// simple PD boundary controller
uy = K_d * (-speed_y) + K_p * (target_y - pos_y);

// simple controller where longitudinal acceleration targets the desired speed objective
ux = erfc(speed-des_speed)-1;

//apply_acceleration(myids[i], 0, 0); // default: zero acceleration on both axes
apply_acceleration(myids[i], 0, uy); // in the context of our example, we only want to maintain the vehicle within the road boundaries, and r
//apply_acceleration(myids[i], ux, uy); // typical use-case
```

Proper usage of get_distance_to_road_boundaries_at is crucial for this operation. In this example, each vehicle requests the distances through this API function according to a longitudinal distance, as evident in the code. Naturally, this allows the vehicle to properly plan according to an upcoming boundary change. Of course, this assumes that within the designated longitudinal distance we examine: a) the lateral boundary (either left or right) at the current placement already has an appropriate safety lateral distance, and b) the boundary will shift to the observed distance as a monotonic function (i.e., no sudden changes in the boundary that would cause "bulges").

In the code example above, we have a simple heuristic rule that obtains a lateral placement to target by taking the following into account: a) the lateral distance to the boundary, b) the vehicle's width, and c) an additional safety distance. Therefore, the actual lateral acceleration to be applied stems directly from a PD controller that has as input this targeted lateral placement.

Important Note: This boundary controller example provides one way of dealing with boundaries. This serves as a working example to showcase the usage alongside the API function that returns relevant distance information. Users can utilize this directly, extend it, or develop something entirely different that may be better tied to their own controller. For instance, the boundary controllers in [7, 2] has instead the form of a lateral acceleration bound based on distance information (and the vehicle's current speed).

12 Bidirectional Highways and Internal Boundary Control

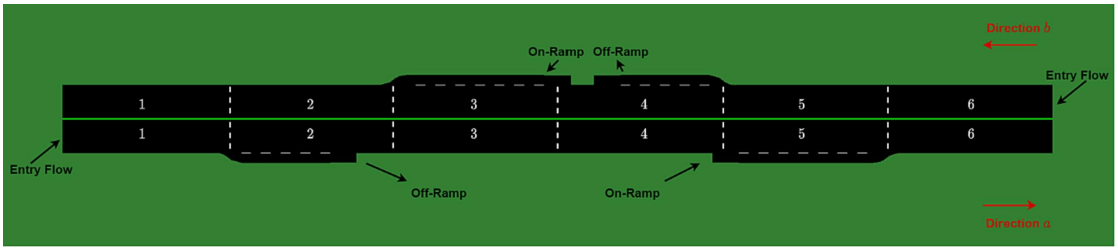
We can directly extend the scenario above in a bidirectional highway environment, one that contains on-ramps and off-ramps as well in each direction. Scenario example 07_highway_bidirectional_ibc_scenario follows exactly the same design principles as above, but it also contains the opposite direction in the road network. Notably, no change in the vehicles' controller is required, given that the lateral boundaries are properly designed.

In the context of lane-free traffic, this bidirectional highway environment with on-ramps and off-ramps is examined for the emerging lane-free application of *Internal Boundary Control* (IBC). The provided example closely follows the scenario examined in [3]. Interested users can refer to this paper for more details. In IBC application, the lateral boundary separating the two opposite directions *can be controlled* and move according to the conditions of the road (based on online measurements of density and flow), in order to grant more lateral capacity towards one direction;

and therefore increase efficiency by exploiting the available lateral capacity of the road. As such, for IBC we assume that there is *no physical separation* of the two directions, and that lane-free vehicles conform to a *virtual separating boundary*.

12.1 Scenario Setup

A small-scale version of the network setup (in the .net.xml) file is the following (network and snapshot used in [3]):



This is achieved by simply mimicking the same network structure for the opposite direction with proper placing of the new road edges and vertices. Likewise, in the route file (.rou.xml), we need to additionally prescribe the routes of the opposite direction as well, along with the lateral boundaries. Therefore, the route file (.rou.xml) now contains:

```
<route id="main_highway"  edges="warm_up segment1 segment2 segment3 segment4 segment5 segment6"
leftBoundaryLevelPoints="0 0 0 0 0 0 0" leftBoundaryOffsets="500 1000 1500 2000 2500 3000" leftBoundarySlopes="0.045
0.045 0.045 0.045 0.045 0.045" rightBoundaryLevelPoints="-10.2 -10.2" rightBoundaryOffsets="1020" rightBoundarySlopes
="0.01" rightBoundaryConstant="-10.2" leftBoundaryVisualizerColor="green"/>
<route id="on_ramp_to_highway"  edges="segment5  segment6"  leftBoundaryLevelPoints="-10.2 0 0" leftBoundaryOffsets="
2800 3000" leftBoundarySlopes="0.01 0.01" rightBoundaryLevelPoints="-13.2 -10.2" rightBoundaryOffsets="2850"
rightBoundarySlopes="0.007" rightBoundaryConstant="-10.2" influencedBy="main_highway"/>
<route id="exit_from_off_ramp"  edges="warm_up segment1 segment2" leftBoundaryLevelPoints="0 0 0 -10.2"
leftBoundaryOffsets="500 1000 1140" leftBoundarySlopes="0.01 0.01 0.01" rightBoundaryLevelPoints="-10.2 -13.2"
rightBoundaryOffsets="1100" rightBoundarySlopes="0.05" rightBoundaryConstant="-10.2"/>

<!-- For the opposite side, we do the same operation. Note that the "left" and "right" is w.r.t. the local
orientation of the vehicle, and we place the boundaryLevelPoints, offsets, slopes, with the opposite order for the
opposite side, as in this example below -->
<route id="main_highway_opp"  edges="warm_up_opp segment6_opp segment5_opp segment4.2_opp segment4.1_opp
segment3_opp segment2_opp segment1_opp" leftBoundaryLevelPoints="0 0 0 0 0 0 0" leftBoundaryOffsets="3500 3000 2500
2000 1500 1000" leftBoundarySlopes="0.045 0.045 0.045 0.045 0.045 0.045" rightBoundaryLevelPoints="10.2 10.2"
rightBoundaryOffsets="2480" rightBoundarySlopes="0.01" rightBoundaryConstant="10.2" leftBoundaryVisualizerColor="red"
/>
<route id="on_ramp_to_highway_opp"  edges="segment3_opp  segment2_opp segment1_opp" leftBoundaryLevelPoints="10.2 0
0 0" leftBoundaryOffsets="1700 1500 1000" leftBoundarySlopes="0.01 0.01 0.01" rightBoundaryLevelPoints="13.2 10.2"
rightBoundaryOffsets="1650" rightBoundarySlopes="0.01" rightBoundaryConstant="10.2" influencedBy="main_highway_opp"/>
<route id="exit_from_off_ramp_opp"  edges="warm_up_opp segment6_opp segment5_opp segment4.2_opp
leftBoundaryLevelPoints="0 0 0 0 10.2" leftBoundaryOffsets="3500 3000 2500 2350" leftBoundarySlopes="0.01 0.01 0.01
0.01" rightBoundaryLevelPoints="10.2 13.2" rightBoundaryOffsets="2400" rightBoundarySlopes="0.05"
rightBoundaryConstant="10.2" influencedBy="main_highway_opp"/>
```

where new arguments (highlighted in red boxed) are needed in order to properly update the boundaries through the API as we show below. Essentially, the values of leftBoundaryLevelPoints for each route can be adjusted in an online manner. The control variable from the user's perspective is an array of normalized *epsilon* values (with the same array size), with values of 1 corresponding to the initial configuration in the rou.xml file. See also Equation 10 below.

Specifically:

- **rightBoundaryConstant:** is the value that will be used for the **online adjustment** of the leftBoundaryLevelPoints through the API. For instance, given the boundary control variable e for a specific $leftBoundaryPoint$, its adjusted value will be calculated as:

$$
\begin{aligned}
leftBoundaryPointAdjusted &= rightBoundaryConstant \\
&+ e \cdot (leftBoundaryPoint - rightBoundaryConstant)
\end{aligned} \tag{10}
$$

Note that rightBoundaryConstant is an optional argument. By default, its value is set to the maximum (or minimum, depending on the direction right/left) of rightBoundaryLevelPoints.

- **influencedBy**: this additional argument can be used so that when epsilon values are updated for one route's left boundaries, then all other routes that have specified this influence will have their epsilons updated automatically.

  For instance, consider the route "main_highway" and the route "on_ramp_to_highway". When defining the boundaries for route "on_ramp_to_highway", we can specify the influencedBy argument as "main_highway". This means that when we update the left boundary on the main_highway, the left boundaries on the route "on_ramp_to_highway" will be updated as well. For this to work, the influencer (in this example, "on_ramp_to_highway") must contain a subset of leftBoundaryOffsets of the influencer ("main_highway"). E.g., if "main_highway" has leftBoundaryOffsets="1000 2000 3000 4000 5000", then "on_ramp_to_highway" must have at least two consecutive offsets for this to work, e.g., leftBoundaryOffsets="1500 2000 3000 4000 5000", means that the segments within the "2000 3000 4000 5000" will be affected automatically by the epsilons used for the main highway.

12.2 Controller/Lane-Free Plugin Setup

The relevant function in the API for the online update of epsilon values is:

- **set_epsilon_left_boundary**(char* route_name, double* epsilon_array, size_t epsilon_array_size)

with accompanying technical usage instructions in the header file:

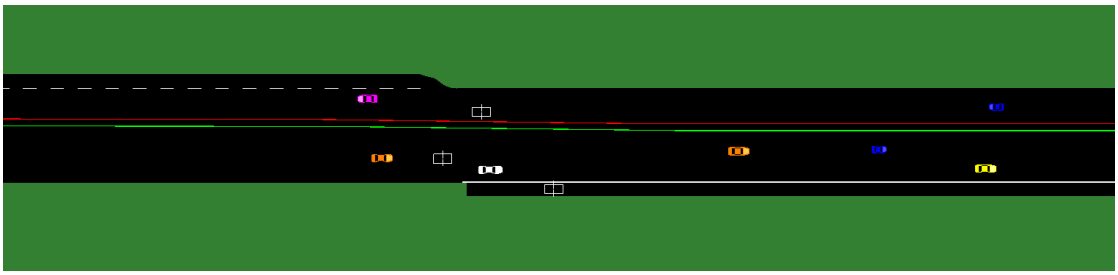
```
// Provide the associated route_name with a char array, and an array of epsilon values to change the left boundary of route_name online
// (and the size of the double array), according to the epsilon_array values
// epsilon_array should have the same size with the defined leftBoundaryLevelPoints for this route.
// Online adjustment of the left boundary for a specific leftBoundaryLevelPoint, and epsilon_value is as follows:
// leftBoundaryAdjusted = rightBoundaryConstant + epsilon_value * (leftBoundaryLevelPoint - rightBoundaryConstant)
// E.g., for epsilon_value=1, the left boundary will not change.
// All epsilon values should be within the range 0 < epsilon_value < 2
EXTERN libLaneFreePlugin_EXPORT void (*set_epsilon_left_boundary)(char* route_name, double* epsilon_array, size_t epsilon_array_size);
```

and a simple code example within the LaneFree.cpp file:

```
// For IBC Application
int ibc_frequency_update = 100; // in time-steps

if (get_current_time_step()>0 && (get_current_time_step() % ibc_frequency_update) == 0) {

    char route_name [30] = "main_highway"; //need the string information of the route to update the epsilons
    int epsilon_size = 7; // number of leftBoundaryLevelPoints in the .rou.xml file

    //array of epsilons (1 is the default value, exactly at the point of road's width)
    double epsilon_array[7] = { 0.95, 1.1, 1.2, 1.3, 1.2, 1.1, 0.95};


    set_epsilon_left_boundary(route_name, epsilon_array, epsilon_size);

    char route_name_opp[30] = "main_highway_opp";
    double epsilon_array_opp[7];
    double epsilon_threshold = 0.05;
    // simply assign the complementary lateral availability of the road,
    // accounting for an additional 5% gap between the two opposing boundaries
    for (int e_i = 0; e_i < epsilon_size; e_i++) {
        epsilon_array_opp[e_i] = 1 + (1 - epsilon_array[e_i]) - epsilon_threshold;
    }

    set_epsilon_left_boundary(route_name_opp, epsilon_array_opp, epsilon_size);

}
```

For the purposes of this example, we simply set a constant value for the epsilons (and consequently the lateral level points) that gives more lateral capacity to the right direction (from the top-down view). Users interested in working with IBC should properly extend it, and combine it with proper measurements (e.g. density, flow, as specified in Section 7) to establish a closed-loop system. Refer to [3] for additional details, and in Section 3.2 of that paper for further technical/practical guidelines that could potentially be useful in related endeavours.

A snapshot of the simulation environment based on this example follows:



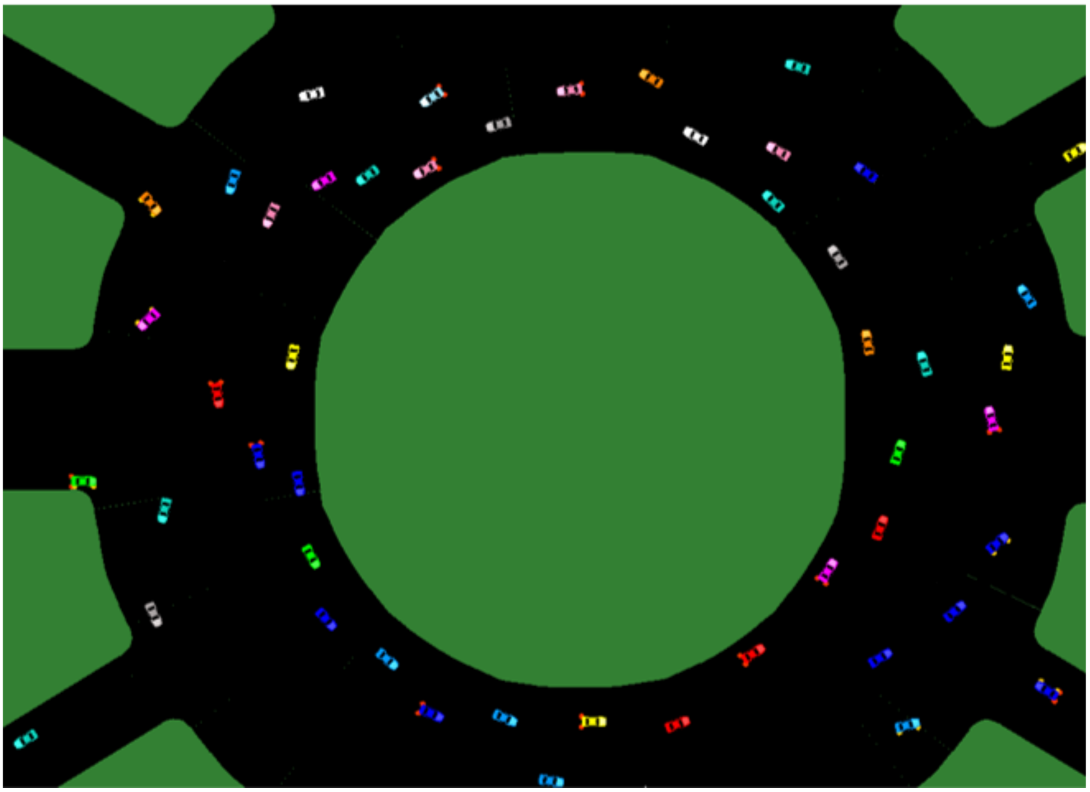
13 Roundabouts

Roundabouts are quite distinct from the aforementioned scenarios, in the sense that the double integrator model for the vehicle movement dynamics are not fitting in this case. We need to properly handle the lateral movement of vehicles, so that they properly enter, exit and generally navigate within the roundabout. Certainly, the double integrator can be technically employed (users can directly examine the provided roundabout scenario with the example), but the infrastructure of SUMO does not exhibit smooth and realistic motion of vehicles as they navigate between segments of the road.

Therefore, we suggest users to work with the bicycle model, and specifically with the option for control over the global coordinates for movement strategies targeting roundabout scenarios.

Related details are provided in Sections 5.3.2, 5.3.3. Please refer also to [5, 6], which propose a vehicle movement strategy for lane-free roundabout environments, and make use of the relevant features in our simulation environment. Other related work in lane-free roundabouts, but without the use of our microscopic simulation environment is [4].

The provided scenario 08_roundabout_scenario involves a complex roundabout that resembles the famous roundabout at the Arc de Triomphe in Paris, and was utilized for the experiments in [6]. A snapshot of the simulation environment in this environment when populated with vehicles and an appropriate controller follows:



Important Notes:

- Ad-hoc adjustments for a fully-fledged vehicle movement strategy are necessary in this environment when operating with the global coordinates. Vehicles need to navigate according to their designated route, and need to request such route-related information from the API since the maneuvering is not handled automatically from the underlying mapping from local to global coordinates (as done otherwise). This is accomplished with the API function: get_edge_of_vehicle(veh_id) that returns the residing edge of the requesting vehicle, and other functions that return edge-related information given the vehicle's routing: get_destination_edge_id(veh_id), get_origin_edge_id(veh_id), get_next_edge_id(veh_id), get_previous_edge_id(veh_id). For instance, vehicles need to know the designated exit turn and then plan their movement accordingly.

- Design of lateral boundaries is not applicable in this case, since it assumes a road structure

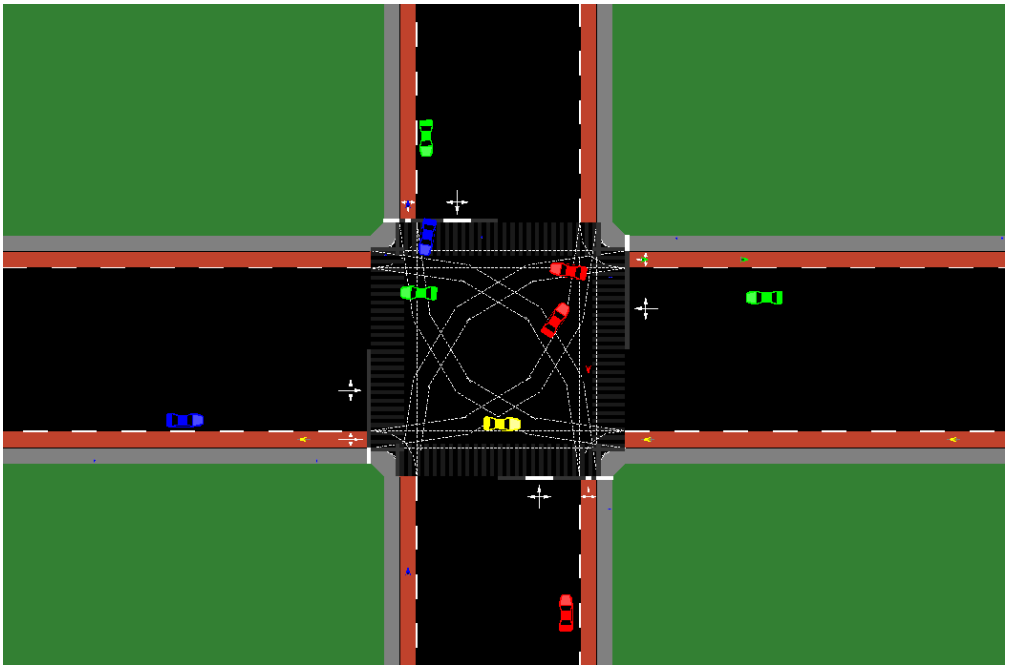
that is not curved, and users need to properly integrate the dimensions and characteristics of the roundabout accordingly.

- Neighbor information is properly adjusted for this environment, along with collision notification.

14 Intersection

We also include intersection scenarios, where four directions typically meet at a shared common space. This setup lacks strict predefined rules for the shared area, making it a challenging but important topic for traffic-related studies.

A preconfigured scenario 09_intersection_scenario is included among the example files, providing insight into the design of intersections and guidance on constructing similar setups.



Additionally, two specific features are developed and examined within this context:

- opposite-direction movement with collision detection, and
- bicycles and pedestrians integration

This section will provide detailed instructions on configuring these features and discuss their associated limitations.

14.1 Opposite Direction Movement

In intersections, the shared space lacks distinct areas allocated to specific directional flows. Instead, an edge is shared among all directional flows simultaneously, meaning that collisions occurring within the intersection's shared space are not assigned by the simulator to any specific directional flow. While the simulator handles such scenarios correctly, we observed that collisions

frequently occurred on conflicting edges closely outside the common area. Specifically, vehicles would enter opposite or neighboring lanes while attempting to follow their controller trajectories, resulting in head-on collisions that were not detected by the simulator. This issue highlighted the need for detecting opposite-direction movements and associated collisions.

Opposite or neighboring lanes for this purpose cannot be officially defined in Netedit using traditional tags. Instead, these labels must be manually specified **after** the network creation in Netedit. More precisely, following the generation of the .net.xml file, it is necessary to manually add a custom tag, "neigh_lane", to each edge with a designated neighboring lane. This tag should include the ID of the opposite lane. An example of this configuration is shown in the following snapshot, and a complete intersection example is included in the tutorial files:

```
<edge id="EN0" from="gneJ2" to="gneJ1" priority="-1">
    <lane id="EN0_0" index="0" speed="13.89" length="106.00" width="10.00" shape="
    120.00,5.00 14.00,5.00"/>
    <neigh_lane  lane="EX0_0"/>
</edge>
<edge id="EN1" from="gneJ4" to="gneJ1" priority="-1">
    <lane id="EN1_0" index="0" speed="13.89" length="106.00" width="10.00" shape="
    -5.00,120.00 -5.00,14.00"/>
    <neigh_lane  lane="EX1_0"/>
</edge>
<edge id="EN2" from="gneJ0" to="gneJ1" priority="-1">
    <lane id="EN2_0" index="0" speed="13.89" length="106.00" width="10.00" shape="
    -120.00,-5.00 -14.00,-5.00"/>
    <neigh_lane  lane="EX2_0"/>
</edge>
<edge id="EN3" from="gneJ3" to="gneJ1" priority="-1">
    <lane id="EN3_0" index="0" speed="13.89" length="106.00" width="10.00" shape="
    5.00,-120.00 5.00,-14.00"/>
    <neigh_lane  lane="EX3_0"/>
</edge>
<edge id="EX0" from="gneJ1" to="gneJ2" priority="-1">
    <lane id="EX0_0" index="0" speed="13.89" length="106.00" width="10.00" shape="
    14.00,-5.00 120.00,-5.00"/>
    <neigh_lane  lane="EN0_0"/>
</edge>
```

Important Notes:

- Opposite lanes must always be of equal length. Discrepancies in length can result in invalid internal calculations, leading to undetected collisions.
- Modifications to the network using Netedit will remove the "neigh_lane" tags. As a result, these tags must be re-added after any changes. It is strongly recommended to maintain a backup of the .net.xml files when utilizing this feature.

14.2 Bicycles and Pedestrians

Another important feature for studying intersections is the inclusion of bicycles and pedestrians. Intersections are often located in urban environments where bicycles and pedestrians are common, introducing additional complexity. The need to account for those types of entities with differing capabilities makes this a particularly challenging and valuable area of study.

To support this, we provide a set of new functions for the vehicle controller. These functions allow access to a comprehensive list of bicycles and pedestrians in the network, including their global positions and the edges of their origins and destinations.

To simulate flows of bicycles and pedestrians, specific additions must be made to the .rou.xml file:

- **Bicycle entities:** a new type must be declared, after which flows can be defined similarly to vehicle flows. An example is provided below:

```
<vTypeDistribution id="typedist2">
    <vType id="lane_free_bicycle" vClass="bicycle" lcStrategic="-1." lcCooperative="-1."
    lcKeepRight="0." lcSpeedGain="0."/>
</vTypeDistribution>

<flow id="bike_flow_1" type="typedist2" begin="25" end="122" number="12" route="
origin1_straight" departSpeed="3" departPos="0"/>
<flow id="normal_flow_0_left" type="typedist1" begin="25" end="150" number="2" route="
origin0_left" departSpeed="10" departPos="0" departSpeedLimitDownstream="false" latLow="0.0
" latHigh="1"/>
<flow id="bike_flow_2" type="typedist2" begin="35" end="132" number="12" route="
origin2_straight" departSpeed="3" departPos="0"/>
<flow id="normal_flow_1_left" type="typedist1" begin="35" end="160" number="2" route="
origin1_left" departSpeed="10" departPos="0" departSpeedLimitDownstream="false" latLow="0.0
" latHigh="1"/>
```

- **Pedestrian entities:** for pedestrian flows, the "personFlow" method must be used, which offers similar control variables to standard flow definitions but with a slightly different syntax. An example is shown below:

```
<vTypeDistribution id="typedist3">
    <vType id="lane_free_pedestrian" vClass="pedestrian" jmIgnoreFoeSpeed="100"
    jmIgnoreFoeProb="1." impatience="10." lcAssertive="1" lcImpatience="1"
    lcTimeToImpatience="1"/>
</vTypeDistribution>

<personFlow id="p" type="lane_free_pedestrian" begin="21" end="100" period="30">
    <walk from="EN0" to="EX2"/>
</personFlow>
<personFlow id="p1" type="lane_free_pedestrian" begin="21" end="110" period="30">
    <walk from="EN3" to="EX1"/>
</personFlow>
```

The behavior or controller for those entities is designed separately:

- **Bicycle entities:** by default, bicycles utilize the IDM controller. A complete list of predefined controllers is available at https://sumo.dlr.de/docs/Definition_of_Vehicles%2C_Vehicle_Types%2C_and_Routes.html#car-following_models. When the "carFollowModel" parameter is set to "LaneFree", bicycles can be fully controlled by the custom controller using the same functions available for vehicles.
- **Pedestrian entities:** Pedestrians currently use the predefined model "striping". Further details about this model can be found at https://sumo.dlr.de/docs/Simulation/Pedestrians.html#model_striping.

Important Notes:

- All bicycles and pedestrians neglect any safety constraints, leaving this task up to the controlled vehicles.
- Collisions are currently detected only between vehicles and bicycles. Moreover, only the front part of the bicycle is considered its hitbox. Collisions involving the rear of the bicycle are not registered.

15 Platoons

We provide functionality for spawning platoons of vehicles on the road, enabling the simulation of groups of aligned vehicles that move together as a unit. Each platoon consists of a leader (the vehicle at the front) and a specified number of followers. When a platoon is spawned, the leader enters the simulation first, and lateral space behind it is reserved to ensure all followers are successfully added to the simulation. This approach guarantees the correct formation and behavior of the platoon as an integrated group. Multiple flows with platoons can coexist within the same edge or lane.

Using the custom controller, each vehicle in a platoon can be individually controlled, allowing for full customization of behavior. A set of dedicated functions is available for managing platoons:

- get_all_platoon_ids(): returns a pointer to an array containing all unique platoon IDs. Each platoon is assigned the ID of its leader.
- get_all_platoon_ids_size(): provides the size of the above array.
- get_platoon_vehicles_ids(NumericalID platoon_id): given a platoon ID, returns a pointer to an array containing the IDs of all vehicles in the specified platoon.
- get_platoon_vehicles_ids_size(NumericalID platoon_id): given a platoon ID, provides the size of the above array.

Each type of platoon must be declared as a separate flow in the .rou.xml file. To enable platoon functionality, the following field modifications and additions are required:

- insertionPolicy: this field must be set to “platoon” to activate platoon-specific behavior.
- number: represents the number of platoon leaders, not the total number of vehicles. For instance, to simulate 5 platoons with 6 vehicles each, this field should be set to 5 (not 30).
- platoonSize: specifies the number of vehicles in each platoon. For the above example, this value would be 6.
- platoonTimeStepDistance: indicates the time-step-based spacing between vehicles within a platoon. By adjusting this value relative to the initial speed, the distance between vehicles during insertion can be controlled.

An example of a .rou.xml file can be found in the following snapshot, taken from the scenario configuration 10_platoons_scenario in the tutorial files:

```
<flow id="normal_flow_0_left" type="typedist1" begin="0" end="55" number="5" route="
origin0_straight_platoon" departSpeed="10" departPos="0" departSpeedLimitDownstream="false"
 latLow="0.0" latHigh="1" insertionPolicy="platoon" platoonSize="5" platoonTimeStepDistance
="4"/>
<flow id="normal_flow_0_left1" type="typedist1" begin="4" end="55" number="7" route="
origin0_straight_platoon" departSpeed="10" departPos="0" departSpeedLimitDownstream="false"
 latLow="0.0" latHigh="1" insertionPolicy="platoon" platoonSize="3" platoonTimeStepDistance
="4"/>/>
<flow id="normal_flow_0_right" type="typedist1" begin="3" end="55" number="5" route="
origin0_straight" departSpeed="8" departPos="0" departSpeedLimitDownstream="false" latLow="
0" latHigh="0.5"/>
<flow id="normal_flow_1_right" type="typedist1" begin="4" end="55" number="5" route="
origin1_straight" departSpeed="8" departPos="0" departSpeedLimitDownstream="false" latLow="
0.0" latHigh="0.5" />
```

References

[1] 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 *21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2575–2582, 2018.

[2] M. Malekzadeh, D. Manolis, I. Papamichail, and M. Papageorgiou. Empirical investigation of properties of lane-free automated vehicle traffic. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 2393–2400, 2022.

[3] M. Malekzadeh, D. Troullinos, I. Papamichail, M. Papageorgiou, and K. Bogenberger. Internal boundary control in lane-free automated vehicle traffic: Comparison of approaches via microscopic simulation. *Transportation Research Part C: Emerging Technologies*, 158:104456, 2024.

[4] M. Naderi, M.-K. Mavroeidi, I. Papamichail, and M. Papageorgiou. Optimal orientation for automated vehicles on large lane-free roundabouts. In *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 8207–8214, 2023.

[5] M. Naderi, M. Papageorgiou, I. Karafyllis, and I. Papamichail. Automated vehicle driving on large lane-free roundabouts. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1528–1535, 2022.

[6] M. Naderi, M. Papageorgiou, D. Troullinos, I. Karafyllis, and I. Papamichail. Controlling automated vehicles on large lane-free roundabouts. *IEEE Transactions on Intelligent Vehicles*, 9(1):3061–3074, 2024.

[7] M. Papageorgiou, K.-S. Mountakis, I. Karafyllis, I. Papamichail, and Y. Wang. Lane-free artificial-fluid concept for vehicular traffic. *Proceedings of the IEEE*, 109(2):114–121, 2021.

[8] D. Troullinos, G. Chalkiadakis, D. Manolis, I. Papamichail, and M. Papageorgiou. Lane-free microscopic simulation for connected and automated vehicles. In *IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 3292–3299, 2021.

[9] D. Troullinos, G. Chalkiadakis, I. Papamichail, and M. Papageorgiou. Conditional max-sum for asynchronous multiagent decision making. In *Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems, (Accepted)*, 2025.