A Stochastic Continuous Cellular Automata Traffic Model with Fuzzy Decision Rules

POLITECNICO DI MILANO

DEPARTMENT OF MATHEMATICS

DOCTORAL PROGRAMME IN MATHEMATICAL ENGINEERING

Doctoral Dissertation of: **Öznur Yeldan**

Matricola: 724247

Cycle: XXIII

Supervisor: **Prof. Alberto Colorni**

Tutor: **Prof. Roberto Lucchetti**

Tha Chair of the Doctoral Program: **Prof. Paolo Biscari**

2012

I would like to dedicate my thesis to my beloved ones, especially...

to my dad and mom who have believed in me to achieve this task and encouraged me all the way since the beginning of my studies;

to the joy of my life who has been a great source of motivation and inspiration with his patience, love and understanding;

to his family who have taken care of me all the time I was away from my parents and supported me for my achievements.

Abstract

Traffic models based on cellular automata are computationally efficient because of their simplicity in describing complex vehicular behaviors and their ability of being easily implemented for parallel computing. On the other hand, the other microscopic models such as car-following models are computationally more expensive, but they have more realistic driver behaviors and detailed vehicle characteristics. In this dissertation, we propose a hybrid between these two categories defining a traffic model based on continuous cellular automata. In this way, we are able to combine the efficiency which is typical of CA models, with the accuracy of the other microscopic models. More precisely, we introduce a stochastic continuous cellular automata traffic flow model where the space is not coarse-grain like in the Nagel-Schreckenberg kind of models, but it is continuous. The continuity allows us also to embed naturally a multi-agent system based on fuzzy logic which is proposed to handle uncertainties in decision making on road traffic. Therefore, we can simulate different driver behaviors and study the effect of heterogeneity (different composition of vehicles) within the traffic stream from the macroscopic point of view. We define our model first for a single-lane road and then we extend the model to the multi-lane case. The extension is done by a union of interacting single-lane models where the interaction is given by a transfer operation. We then show that this model can actually be simulated by a continuous cellular automata. In this way, we frame the multi-lane model inside the class of continuous cellular automata. The results obtained by a series of experiments have shown us that our model is able to reproduce the typical traffic flow phenomena with a variety of effects due to the heterogeneity of traffic.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In recent decades, growing traffic congestion and increased number of accidents have become one of the most prior problem of the society. In populated areas the existing road networks are not able to satisfy the demand. The construction of new roads is usually not a solution and often is not socially desired. These reasons together with the great economical costs lead to new traffic management and information systems.

Traffic models are thus fundamental resources in the management of road network. There is a wide range of alternative modeling approaches now available which can be roughly divided into three categories: macroscopic, mesoscopic and microscopic models depending on the level of detail. Microscopic models are promising models for their ability to simulate detailed phenomena (each individual vehicle) in traffic which yields to an accurate representation of traffic flow, and macroscopic ideas can be studied with microscopic models. On the other hand, these models have the disadvantage of the computational requirements and their associated costs (e.g., parallel computing) requiring modern computer power. This is likely the reason microscopic models were not used till recent decades. However, as computers increased in power, microscopic modeling became significantly convenient.

Among microscopic traffic flow models, cellular automata (CA) models have the ability of being easily implemented for parallel computing because of their intrinsic synchronous behavior. However, CA models are lack of the accuracy of other microscopic traffic models such as the time-continuous car-following models. This lack is compensated by their simplicity which make them numerically very efficient and can be used to simulate large road networks in real-time or even faster.

In this dissertation, we aim to give a completely new CA traffic model which gets closer to time-continuous car-following models introducing some continuity without losing the computational advantages which are typical of CA models. All the previously introduced CA traffic models have the property of representing the space of road discretely as cells. Therefore, to define a CA model where the space is a continuous variable we have to abandon this idea and embrace a new philosophy where we assume that cells represent vehicles. This gives the immediate advantage of having less cells to compute compared to the previous CA traffic models. Moreover, introducing the continuity in space gives us the possibility to refine the microscopic rules that govern the traffic dynamic using fuzzy reasoning (fuzzy logic) to mimic different real-world driver behaviors.

Human decisions imply uncertainties since most of our behaviors have fuzzy nature rather than crisp, and the application of fuzzy set theory is a useful tool to handle uncertainties. All parameters of the decision process of the drivers are modeled individually by means of fuzzy subsets, thus various types of drivers (kinds of vehicles in our case) can be taken into consideration. This gives us the possibility to study how the heterogeneity of drivers can influence the traffic macroscopically.

1.1 Research Motivation and Objectives

Our principal aim is to present a new approach to cellular automata traffic flow models for single-lane and multi-lane roads, in order to simulate the effect of heterogeneity of driver behaviors in traffic in an efficient way, where the heterogeneity is obtained via fuzzy decision rules. The main concerns that we would like to face in this dissertation are:

- Is it possible to define a stochastic single-lane CA traffic model where,
    - The physical variables defining a vehicle such as position and velocity, are continuous.
    - The number of cells is related only on the number of vehicles to have a more efficient model in terms of computational time. Indeed, in the Nagel-Schreckenberg (NaSch) model and in the variants of this model (see Section

    2.2.2), there are in general cells representing empty pieces of road. These cells are in any case computed even if they are not occupied by a vehicle.
  - It is implemented the driver behavior into the model via fuzzy decision rules.
  - The neighborhood is a classic one such as von Neumann neighborhood, etc. (see Section 3.2). In the NaSch-type models the neighborhood is not of this form and it depends on the speed limit (see Section 3.2.3). Note that the smaller the neighborhood is, the less cells to take into account in the update of each cell state and so less computational time.

- Is it possible to extend this stochastic single-lane CA traffic model to a multi-lane one. The question is non-trivial since it is lost the cell-space correspondence which is typical of the NaSch-type models.

The final objectives that we consider in Chapter 6 are;

- Analyzing the real-time simulation results to see which traffic phenomena are observed, and the interactions of a big amount of vehicles with different types, i.e., examining the dynamic structure of the traffic stream.

- Examining some macroscopic variables such as the flow and the density of the multi-lane traffic road with a variety of different initial conditions (scenarios) given, to give a first test of how the model reacts.

1.2 Outline of the Dissertation

The dissertation is organized as follows:

Chapter 2 We present the state of the art. An overview of traffic flow theories including microscopic, macroscopic and mesoscopic models are briefly introduced. Particular attention is devoted to the deterministic and stochastic cellular automata traffic flow models in literature.

Chapter 3 We give some basic definitions and some preliminaries on fuzzy logic, fuzzy system modeling and cellular automata. We also provide a formal definition of two well-known CA traffic models: Wolfram and Nagel-Schreckenberg.

Chapter 4 We give the reasons of introducing a new traffic flow model, and then we describe our stochastic continuous CA single-lane traffic model via fuzzy decision rules in detail.

Chapter 5 We extend our model to design a stochastic continuous CA traffic model for multi-lane roads where we introduce lane-changing rules. We also prove that this model can be simulated by a continuous cellular automata, framing this model into the class of continuous CA models.

Chapter 6 We first give a general description of the code implemented in Python 2.7 (see Appendix A), then we describe the scenarios of the experiments we have performed. Finally, we comment the results obtained by the experiments from the usual traffic phenomena point of view.

Chapter 7 The last chapter is devoted to draw the conclusions and it is given the recommendations for further studies.

Chapter 2

The State of the Art

In this chapter, we focus on the different traffic flow models that exist in literature. There exits several methods to discriminate between the families of models based on whether they operate in continuous or discrete time, whether they are deterministic or stochastic, or depending on the level of detail. More information can be found in [28]. In Section 2.1, we present an overview that is based on the discriminating according to the level of detail, where microscopic models have the lowest level of aggregation and the highest level of detail, macroscopic models have the highest level of aggregation and the lowest level of detail, and mesoscopic models have a high level of aggregation and a low level of detail.

2.1 Overview of Traffic Flow Models

As we mentioned before, traffic flow models can be broadly categorized into microscopic, macroscopic and mesoscopic in terms of level of detail and process representation.

- *Microscopic traffic flow models* simulate the motion of individual vehicles, i.e., the way drivers behave in traffic stream through a system. Microscopic models are in general created using ordinary differential equations, with each vehicle having its own equation. They are typically functions of position, velocity, and acceleration. In other words, they consider the features, characteristics and interactions between individual vehicles within a traffic stream, such as:
    - Car-following [7, 10, 27], lane-changing [18, 54, 55] and gap-acceptance models [22, 46],

  - Optimal velocity models [3],
  - Psycho-physiological spacing models,
  - Traffic cellular automata models [37, 59],
  - Models based on queueing theory.

  The biggest advantage of microscopic models is the ability to study individual vehicle motion. This feature gains importance because of the fact that each driver drives in a different manner. Macroscopic ideas like flow and density can also be studied with microscopic models. Furthermore, microscopic traffic flow models can yield more detailed and accurate representations of traffic flow. The ability to simulate traffic behavior with high accuracy is a benefit but also a weakness. In order to gain such a high level of accuracy, microscopic simulation models require big amounts of roadway geometry, traffic control, traffic pattern, and driver behavior data. Providing this amount of data can limit users to model smaller networks than those that can be modeled in macroscopic and mesoscopic analysis. The required input data also causes computational intensiveness and in general makes them not suitable for real-time implementation. An important disadvantage of microscopic models is that one ordinary differential equation is required for each vehicle. They are not appropriate to use in case of extreme conditions. Therefore, microscopic models become computationally expensive with large systems of equations, requiring modern computer power to make them convenient. However, as computers increased in power and decreased in cost, microscopic models have recently gained more importance and used in simulating traffic on the level of cities and freeway networks.

- *Macroscopic traffic flow model* is a mathematical model that uses aggregate data to describe the behavior of large numbers of vehicles in terms of flow, density and speed of a traffic stream, such as:
  - The continuum approach,
  - The Lighthill, Whitham, Richards model (the LWR model), is based on a scalar, time-varying, non-linear, hyperbolic partial differential equation. One of its basic assumptions is that velocity depends on traffic density, so it uses the resemblance of vehicles in traffic flow to particles in a fluid [35, 49],

  - The Aw and Rascle Model (the AR model) is a more recent model that attempts to move away from a fluid-flow based model. The authors argue that the older macroscopic models have held too closely to the fluid dynamic approach [2],
  - The H. Michael Zhang model (the Zhang Model) moves completely away from fluid behavior. The Zhang Model implements a second equation derived from a microscopic model, which establishes a macro-micro link [65].

  Macroscopic models are based on continuum mechanics and typically require fluid-dynamic models. In macroscopic approach, the individual vehicle manoeuvres, such as lane-changing, are usually not explicitly represented. The primary advantage of macroscopic models is that they have relatively simple calculations when compared to microscopic models. While the equations model density, flow, and average velocity, only a small number of different parameters are required. A disadvantage of a macroscopic model is the loss of small details or dynamics that can be modeled with microscopic models, since in macroscopic models one does not distinguish and study individual vehicles. Instead a “coarse-grained fluid-dynamical description in terms of density and flow is used. Traffic is then viewed as a compressible fluid formed by the vehicles. Density and flow are related through a continuity equation. Some of the existing macroscopic models have been found to exhibit instabilities in their behavior and often do not track real traffic data correctly.

- *Mesoscopic traffic flow model* is a combination of micro- and macroscopic modeling, i.e., at an intermediate level of detail, vehicles are modeled individually as in microscopic modeling, but governed by rules similar to those seen in macro-simulations. The most well-known mesoscopic flow models are gas-kinetic traffic flow models in which driver behavior is explicitly considered.

  Mesoscopic modeling is appropriate for larger networks when computation resources must be managed effectively and some level of detail is still needed. In terms of vehicle and driver behavior, mesoscopic simulation takes a higher-level view than that seen with microscopic modeling. Vehicles are modeled variously as joining packets, cells or individually in making their way around the road network.

For more detailed information on traffic flow models, an extensive overview is available in [36].

2.2 Cellular Automata Traffic Flow Models in Literature

In traffic flow modeling, microscopic traffic simulation has always been regarded as a time consuming, complex process involving detailed models that describe the behavior of individual vehicles. A real progress in the study of traffic has obtained only with introducing models based on cellular automata. The main advantages of CA are;

- being powerful tools to implement on computers,
- providing a simple physical representation of the system,
- being easily modified to deal different aspects of traffic.

A cellular automaton is a collection of cells (sites) on a grid of specified shape (lattice) that evolves through a number of discrete time steps according to a set of local rules based on the states of neighboring cells. Cellular automata models are capable of capturing micro-level dynamics and relating these to macro-level traffic flow behavior. These models are conceptually simple, thus it can be used a set of simple rules to simulate a complex behavior. The mathematical concepts of CA models were first introduced by John von Neumann in 1948 while trying to develop an abstract model of self-reproduction in biology [56]. He was working on the conception of a self-reproductive machine, called "kinematon", relying on A. Turing's works. In the early 1950's, the physical structure of a cellular automaton was developed with the suggestions of a mathematician Stanislaw Ulam. Ulam was interested in the evolution of graphic constructions generated by simple rules. The base of his construction was a two-dimensional space divided into "cells", a sort of grid. Each of these cells could have the states either 'on' or 'off'. Starting from a given pattern, the following generation was determined according to neighborhood rules. For example, if a cell was in contact with two 'on' cells, it would switch on too, otherwise it would switch off. He noticed that this mechanism permitted to generate complex and graceful figures and these figures could, in some cases, self-reproduce. Ulam introduced to von Neumann the concept of "cellular spaces" to build his self-reproductive machine and therefore to design his universal constructer. In 1970s, CA models entered in a major direction called "simulation games" with one of the most famous application, called "Game of Life" by John Conway, [5, 20]. With the use of powerful computers, these models can outline the complexity of the real world traffic behavior and produces clear physical patterns that are similar to those we see in everyday life.

There are several researches on CA multi-lane traffic flow models. The two-lane or multi-lane traffic simulations using CA and lane-changing rules can be found in [32, 41, 44, 50, 57].

In the following sections, we will give a brief information about the deterministic and stochastic CA traffic models in literature. For a more general review see [37]. Note that throughout this dissertation, abbreviation CA refers to both cellular automata (plural) and cellular automaton (singular).

2.2.1 Deterministic Models

A basic one-dimensional CA model for highway traffic flow was first introduced by Wolfram, where he gave an extensive classification of CA models as mathematical models for self-organizing dynamic systems [16, 61]. It is also called an elementary cellular automaton (ECA). In this deterministic model, a road is defined as a one-dimensional array which has a local neighborhood of three cells wide and therefore there are $2^{2^3} = 256$ different local rules possible which are classified by Wolfram around 1983. One of these rules is called *Rule* 184, derived from Wolfram's naming scheme which is based on the representation of how a cells state evolves in time, depending on its local neighborhood [60]. The rules are governing dynamics of particles (vehicles)[1] and each cell has a binary state where 0 corresponds an empty cell and 1 corresponds to a cell occupied by a vehicle. More specifically, the state of each cell is entirely determined by the occupancy of the cell and its two nearest neighbors. The maximum speed is 1 cell/timestep, therefore, during the motion each vehicle can be at rest or move to the next neighbor side, clearly only if this cell is empty to avoid collisions. The positions are updated synchronously in successive iterations (discrete time steps) in the following way:

---

[1] If we interpret each cell in Rule 184 as containing a particle, these particles behave in many ways similarly to vehicles in a single lane of traffic. Traffic models such as Rule 184 and its generalizations that discretize both space and time are commonly called particle-hopping models [42].

Acceleration and Braking:

$$v_i(t) \to \min(d_i(t-1), 1).$$

Vehicle motion:

$$x_i(t) \to x_i(t-1) + v_i(t).$$

where v_i is the speed and x_i is the position of the i-th vehicle, and d_i represents the distance between the i-th and its front vehicle. The second rule is not actually a "real" rule, it is given just to advance the vehicles in the system.

Wolfram showed that even these simplest rules are capable of emulating complex behavior and he related cellular automata to all disciplines of science [61].

The Rule 184 (R184) ECA is widely used as a prototype of deterministic models of traffic flow. In 1996, Fukui and Ishibashi introduced a generalization of the this model [19], which has a deterministic and a stochastic version (see Section 2.2.2). In this CA model, the maximum speed is increased from 1 to v_{max} $cells/sec$ (one time step is assumed to be 1 second), and vehicles can accelerate instantaneously to the highest possible speed if there are v_{max} or more empty sites in front of them.

2.2.2 Stochastic Models

In 1992, Nagel and Schreckenberg [43] proposed a traffic simulation model for the description of single-lane highway traffic using CA, which is a variant of R184. This model is the first nontrivial traffic simulation model based on CA. In the literature, there are many papers analyzing this model in details such as [42, 47, 51, 52, 53]. Moreover, there are many traffic flow models formulated based on the Nagel-Schreckenberg approach and modified the model for better simulations such as [6, 9, 17, 25, 38, 48, 64].

Nagel-Schreckenberg (NaSch) model is a *time-discrete* and a *space-discrete* model. The traffic road is divided into cells of 7.5 m and it is defined on a one-dimensional array of L sites with closed (periodic) boundary conditions. Each cell may either be occupied by a vehicle or be empty. All vehicles are of the same size and each of them is characterized by its position (cell number) and its velocity (a discrete value between zero and a fixed maximum velocity, v_{max}). The velocity is expressed as the number of cells that a vehicle advances in one time step which is assumed to be 1 second. In the original model v_{max} is assumed to be 5 $cells/sec$, which corresponds with the velocity of

$5 \times 7.5 = 37.5\ m/s$ (135 km/h). Every vehicle has the same target velocity v_{max}. Each update of the movements of the vehicles in this model is determined by four consecutive rules that are performed in parallel to each vehicle at each second as following:

Let us denote the velocity of the n-th vehicle as v_n, the distance between n-th vehicle and its preceding vehicle as $(\Delta x)_n$, which is considered as the distance from front bumper to front bumper.

Acceleration: If v_n has not reached to v_{max} and if the distance $(\Delta x)_n$ is larger than $v_n + 1$, then the velocity is increased by 1. In symbols,

$$v_n \to \min(v_n + 1, v_{max}).$$

Deceleration: If the distance $(\Delta x)_n$ is less than or equal to v_n, the velocity is decreased to $(\Delta x)_n - 1$ (the gap between two vehicles). In symbols,

$$v_n \to \min(v_n, (\Delta x)_n - 1).$$

Randomization: With a probability p, the non-zero velocity of each vehicle is decreased by 1. In symbols,

$$v_n \to \max(0, v_n - 1) \text{ with probability } p.$$

Vehicle motion: Each vehicle is advanced the number of cells equals to the velocity that they have been assigned by the above steps of the model, in other words the position of each vehicle is updated according to the velocity calculated by the preceding rules. In symbols,

$$x_n \to x_n + v_n.$$

The acceleration step is given by the attempt to drive as fast as possible, and the possible acceleration is 1 $cell/s^2$ (corresponding to 7.5 m/s^2) for all vehicles. The deceleration step is introduced to avoid collisions which means that a vehicle cannot move over or pass the position of the front vehicle with the distance $(\Delta x)_n$. The randomization step is introduced to have an additional deceleration of 1 with the probability of p, see Figure 2.1 for an illustration showing how the cells' states evolve in one time step where it is also seen the effect of randomization step (recall that this model has periodic boundary conditions, so a vehicle reaches the end of the road enters from the beginning part). The randomization is due to some driver behaviors such as; choosing not to reach to maximum speed, additional speed losing occurred by an over-reaction at braking, keeping a too large distance to the front vehicle, a delay in the acceleration process caused by stopping in congestion or a sudden deceleration by distraction, which are all realistic human reactions in traffic. The NaSch model would be completely deterministic without this randomization step.

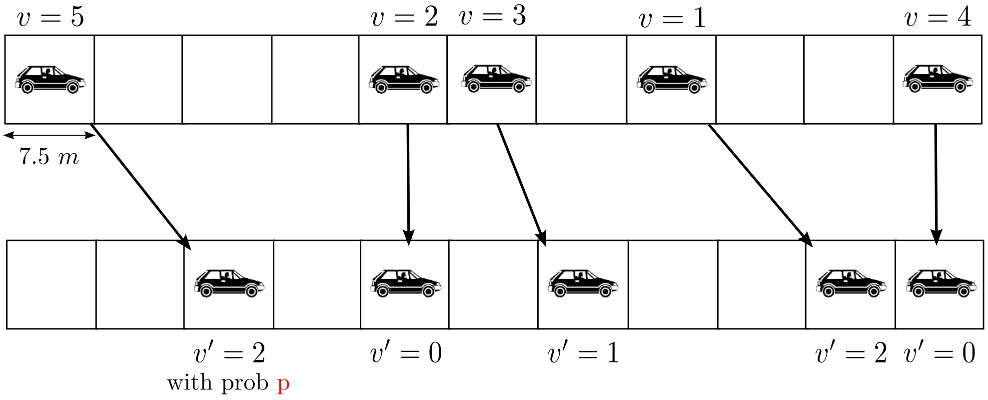


Figure 2.1: An example of one update of the system of the NaSch model. -

Fukui and Ishibashi [19] also introduced a stochastic one-dimensional CA traffic model (the FI model). The FI model differs from the NaSch model in that only the vehicles driving at the highest possible speed of v_{max} $cells/sec$ has a probability of slowing down. More precisely, the stochastic delay is not applied to slower cars since they are already slow. The two models are identical for $v_{max} = 1$ $cell/sec$.

Wang et al. [58] proposed a one-dimensional CA traffic model of high speed vehicles with the FI-type acceleration for all vehicles and the NaSch-type stochastic delay only for the vehicles following the trail of the vehicle ahead, which means that only the vehicles with spacing ahead smaller than the speed limit may be delayed.

Benjamin et al. [4] developed another model (the BJH model) which is an extension of the NaSch model. In the BJH model, drivers have a possibility of starting slowly (starting with some delay) when they start to accelerate from the situation of being stopped. This can arise from a driver's loss of attention as a result of having been stuck in the queue occurred by traffic congestion. This model introduces a probability p_{slow} to simulate it stochastically. The p_{slow} is the "probability of starting slowly" from a static situation. When the velocity of a vehicle is 0 and the distance with the front vehicle is long enough, this vehicle stays at velocity 0 on this time step with probability p_{slow} and accelerates to 1 on the next time step. On the other hand, this vehicle may accelerate normally with probability $1 - p_{slow}$. This rule is called a "*slow-to-start*" rule.

Clarridge and Salomaa [13] proposed a "*slow-to-stop*" rule, which is decelerating before the traffic congestion to avoid collisions, and added the new rule into the BJH model. They observed that the vehicles in the previous models have an unrealistic behavior when approaching a traffic congestion. If a driver has a vehicle in his front with the velocity 0, then this driver may drive up to the front at velocity v_{max} only to brake down to velocity zero in one time step in the cell right behind the front vehicle. Therefore, to make it more realistic, they suggested the addition of a "*slow-to-stop*" rule which causes drivers to go slower when approaching congestions since drivers would slow down much more before where a small congestion is visible from a distance. When Clarridge and Salomaa used this rule in the BJH model, they demonstrated that there were fewer long congestions with many vehicles at a complete stop, and instead there appear to be many slowdowns to avoid these situations, which is more realistic than before.

Note that in all these models that simulate traffic road with CA, the cells represent the space as it is in the NaSch model, so from now on we call them as "NaSch-type" models.

Chapter 3

Preliminaries

3.1 Fuzzy Logic and Fuzzy Systems

In daily language, there is a great deal of imprecision, or we can say “fuzziness” such as the statements: “He is tall” or “He is young”. The classifications, e.g., *healthy, large, old, far, cold*, are fuzzy terms in the sense that they cannot be sharply defined. In other words, these are the statements that are uncertain and imprecise. When we speak of the subset of healthy people in a given set of people, it may be impossible to decide whether a person is in this subset or not. We can give a yes-or-no answer, but there may be loss of information since the *degree* of healthiness is not taken into consideration. At this point, the theory of fuzzy concept becomes an important tool in practical applications.

The mathematical modeling of fuzzy concepts was firstly introduced by Zadeh in 1965, [62], by using the notion of partial degrees of membership, in connection with the representation and manipulation of human knowledge automatically. Since then, successful applications of fuzzy set theory have been developed [31].

3.1.1 Fuzzy Logic and Fuzzy Sets

Fuzzy logic is a logic that aims to provide the structure for approximate reasoning using imprecise propositions based on fuzzy set theory, in a way similar to the classical reasoning using precise propositions (that are either true or false) based on the *classical logic* (also called a *two-valued logic*). In the classical reasoning, the deductive inferences are precise such as the following example:

i) Everyone who is 45 years old or younger are young.

ii) Jane is 45 years old and Jack is 46 years old.

iii) Jane is young but Jack is not.

This is a very precise inference that is correct in the sense of the two-valued logic, but there are some other inferences that cannot be handled by the classical reasoning using two-valued logic such as:

i) Everyone who is 25 to 45 years old is young but if a person is 24 years old or younger then that person is very young; everyone who is 46 to 80 years old is old but if a person is 81 years old or older then that person is very old.

ii) Jane is 45 years old and Jack is 46 years old.

iii) Jane is young but not very young; Jack is old but not very old.

In order to deal with such imprecise inference, we should consider an approximate reasoning such as *fuzzy logic* which allows the imprecise linguistic terms (properties) such as: "old", "high", "fast", "many", "few", where it is required to express the degree of truth by means of belonging concept.

A first attempt to give different degree of truth was developed by Jan Lukasiewicz and A. Tarski formulating a logic on n truth values where $n \geq 2$ in 1930s. This logic called n-valued logic differs from the classical one in the sense that it employs more than two truth values. To develop an n-valued logic, where $2 \leq n \leq \infty$, Zadeh modified the Lukasiewicz logic and established an infinite-valued logic by introducing the concept of membership function.

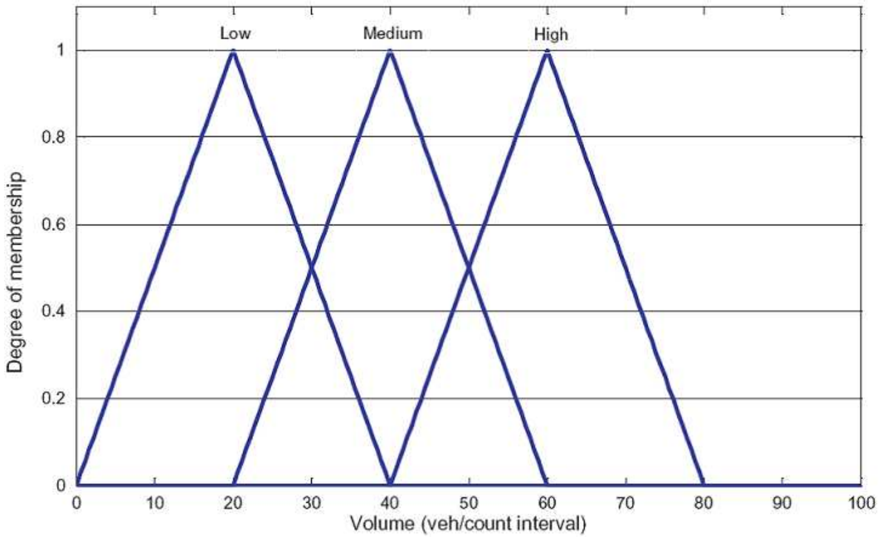
Let X be a classical set of objects, called the universe, whose generic elements are denoted by x. An ordinary subset A of X is determined by its *characteristic function* χ_A from X to $\{0,1\}$ such that,

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

In the case that an element has only partial membership of the set, we need to generalize this characteristic function to describe the membership grade of this element in the set. Note that larger values denote higher degrees of the membership. For a fuzzy subset

A of X, this function is defined from X to $[0,1]$ and called as the *membership function (MF)* denoted by μ_A, and the value $\mu_A(x)$ is called the *degree of membership* of x in A. Thus we can characterize A by the set of pairs as following:

$$A = \{(x, \mu_A(x)), x \in X\}.$$



L. Dimitriou et al. / Transportation Research Part C 16 (2008) 554–573

Figure 3.1: An illustration of MFs for the variable "Volume". -

Figure 3.1 illustrates a characteristic example of a MF for the representation of traffic flow through employing three different properties, i.e., "Low", "Medium" and "High", to describe the variable "Volume". Each fuzzy set is uniquely defined by a membership function. In literature there are several frequently used membership functions such as: triangular membership function, Gaussian membership function, trapezoidal membership function and discrete membership function (see Figure 3.2).

For the purpose of describing the fuzzy logic mathematically we need to generalize also the usual Boolean operators. Let X be the universe set and A be a fuzzy set associated with a membership function $\mu_A : X \to [0,1]$. If $y = \mu_A(x_0)$ is a point in $[0,1]$, representing the truth value of the proposition "x_0 is A", then the truth value of the proposition "x_0 is not A" is given by $1 - \mu_A(x_0)$. Therefore, we have the fuzzy set $\sim A$ with membership function $\mu_{\sim A} = 1 - \mu_A$ for "not being A". Let A, B two

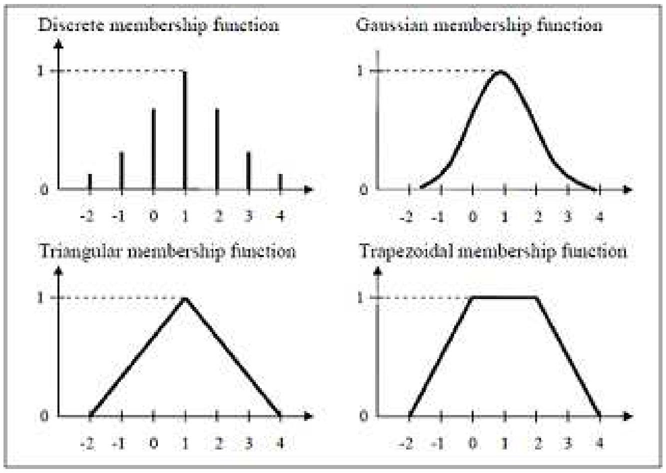


Figure 3.2: Some common shapes of membership functions. -

linguistic terms on the universe X, and let $x, y \in X$. Similar to Boolean logic, in fuzzy logic we are able to give a truth value for the proposition "x is A" AND "y is B" by taking the minimum of the truth values of the propositions "x is A", "y is B". In this way we can represent the proposition "x is A" AND "y is B" as the fuzzy set $A \wedge B$ with the membership function depending on the two variables x, y defined by,

$$\mu_{A\wedge B}(x, y) = \min\{\mu(x), \mu(y)\}.$$

We can define also the other logical operators "or", "implication" and "equivalence" as follows:

$$\begin{aligned}
\mu_{A\vee B}(x, y) &= \max\{\mu_A(x), \mu_B(y)\} \\
\mu_{A\Rightarrow B}(x, y) &= \min\{1, 1 + \mu_B(y) - \mu_A(x)\} \\
\mu_{A\Leftrightarrow B}(x, y) &= 1 - |\mu_A(x) - \mu_B(y)|
\end{aligned}$$

There are many other ways to define the membership functions of these operators (See [11]). For instance for the $\Rightarrow$ operator one can define,

$$\mu_{A\Rightarrow B}(x, y) = \min\{1, 1 + \mu_B(y) - \mu_A(x)\}$$

$$\mu_{A\Rightarrow B}(x, y) = \max\{\min\{\mu_A(x), \mu_B(y)\}, 1 - \mu_A(x)\}$$

$$\mu_{A\Rightarrow B}(x, y) = \max\{1 - \mu_A(x), \mu_B(y)\}$$

$$\mu_{A\Rightarrow B}(x,y)=\begin{cases}1 & \text{if } \mu_A(x)\le\mu_B(y),\\ \frac{\mu_B(y)}{\mu_A(x)} & \text{if } \mu_A(x)>\mu_B(y).\end{cases} \text{ “Goguen's formula”}$$

It is not difficult to see that all these formulae are compatible with Boolean logic in the sense that if A, B are Boolean properties $\mu_{A\Rightarrow B}(x,y)$ is the usual truth table for the operator $\Rightarrow$.

3.1.2 Fuzzy System Modeling

A fuzzy system is a system where inputs and outputs of the system are modeled as fuzzy sets or their interactions are represented by fuzzy relations. A fuzzy system can be described either as a set of fuzzy logical rules or a set of fuzzy equations.

Several situations may be encountered from which a fuzzy model can be derived:

- a set of fuzzy logical rules can be built directly;
- there are known equations that can describe the behavior of the process, but parameters cannot be precisely identified;
- too complex equations are known to hold for the process and are interpreted in a fuzzy way to build, for instance a linguistic model;
- input-output data are used to estimate fuzzy logical rules of behavior.

The basic unit for capturing knowledge in many fuzzy systems is a fuzzy IF-THEN rule. A fuzzy rule has two components: an IF-part (referred to as the antecedent) and a THEN-part (referred to as the consequent). The antecedent and the consequent are both fuzzy propositions. The antecedent describes a condition, and the consequent describes a conclusion that can be drawn when the condition holds.

Consider the black-box vision of an input-output system where we assume that the internal structure of the system is unknown but qualitative knowledge about its behavior is available with the form of a collection of rules involving fuzzy concepts. These rules are presented as an IF-THEN form as following:

Let $A_1, ..., A_n$ and B be fuzzy subsets with membership functions $\mu_{A_1}, ..., \mu_{A_n}$ and μ_B, respectively. A general fuzzy IF-THEN rule has the form,

$$\text{“IF } x_1 \text{ is } A_1 \text{ AND...AND } x_n \text{ is } A_n \text{ THEN } y \text{ is } B\text{”} \tag{3.1}$$

Fuzzy systems operating with these rules are called as *Fuzzy Rule-Based Systems (FRBS)*. There are two main types of these systems: the Mamdani system [15, 39, 40] which we consider in the fuzzy IF-THEN rule 3.1 and the Takagi–Sugeno–Kang (TSK) system [14]. The TSK system is introduced to reduce the number of rules required by the Mamdani model. The main difference between the Mamdani and TSK fuzzy systems lies on the fact that the consequent are fuzzy and crisp sets, respectively. In other words, in the consequent part of the TSK system, it is used a function (equation) of the input variables which results with a crisp value.

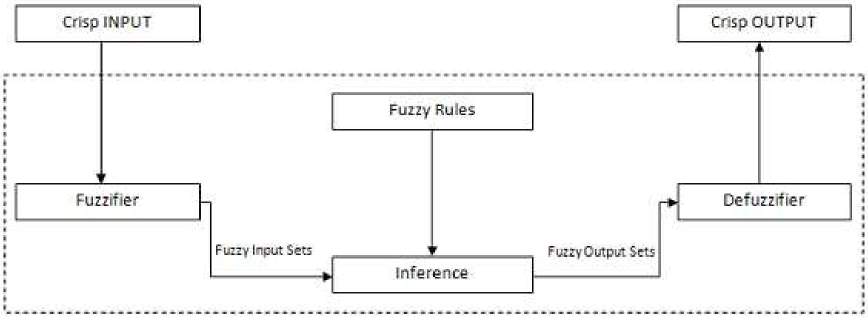


Figure 3.3: General structure of a fuzzy rule-based system. -

In our model, we use a Mamdani type FRBS to try to mimic human behavior in the driving process. Therefore, we consider a vehicle like a black-box receiving inputs from the environment and responding using the variable "acceleration" which we consider as the unique output of the "vehicle system". From the intuition behind the basic behavior of drivers, we derive the fuzzy IF-THEN rules described in Section 4.3. Thus our aim is to try to obtain this output by modeling the "vehicle system" with a Mamdani fuzzy system described by these rules. There are three steps involved in the design of a Mamdani model (see Figure 3.3):

Step 1: The Fuzzifier Module The input data of a fuzzy logic system are a set of crisp values. The function of the fuzzifier is to transform these crisp values into a set of fuzzy values. For example, recall the fuzzy rule defined in 3.1,

"IF x_1 is A_1 AND...AND x_n is A_n THEN y is B"

where $x_1, \ldots, x_n$ are the variables representing the n inputs and $A_1, \ldots, A_n$ are n fuzzy sets with membership functions $\mu_{A_1}, \ldots, \mu_{A_n}$. If the fuzzy system receives the values $\bar{x}_1, \ldots, \bar{x}_n$ as input, the task of the fuzzifier is to calculate

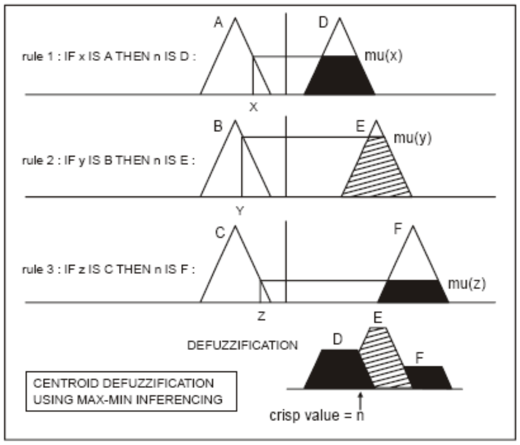
$\mu_{A_1}(\bar{x}_1), \ldots, \mu_{A_n}(\bar{x}_n)$. The membership of each fuzzy input variable is used in evaluating the weights of the rules.

Step 2: Fuzzy Inference Module Fuzzy inference is the key component of the fuzzy logic system. Using the degrees of membership determined during fuzzification, the rules are evaluated according to the fuzzy logic AND operation defined in Subsection 3.1.1. Consider the fuzzy rule defined in 3.1 with inputs $\bar{x}_1, \ldots, \bar{x}_n$, we can associate to this rule the following weight representing the degree of "fulfilment":

$$\mu_{A_1 \wedge \ldots \wedge A_n}(\bar{x}_1, \ldots, \bar{x}_n) = \min\{\mu_{A_1}(\bar{x}_1), \ldots, \mu_{A_n}(\bar{x}_n)\}$$

The output of the inference module is a fuzzy set that is some clipped version of the output fuzzy set. The height of this clipped set is $\mu_{A_1 \wedge \ldots \wedge A_n}(\bar{x}_1, \ldots, \bar{x}_n)$, hence we essentially cut off from the membership function μ_B the subgraph of the points whose ordinates are greater or equal to the height $\mu_{A_1 \wedge \ldots \wedge A_n}(\bar{x}_1, \ldots, \bar{x}_n)$. After collecting all these fuzzy outputs for each rule, we need to combine them to obtain a crisp value. Note that it is possible to elaborate the information more by giving some extra weights for each rule.

Step 3: Defuzzification Defuzzification is a mathematical process used to convert a fuzzy set or fuzzy sets to a crisp value (output). Fuzzy sets generated by the fuzzy inference module must be somehow mathematically combined to give one single number as the output of a model. A commonly used defuzzification method to evaluate the crisp output is the center-of-gravity (COG) method (see Figure 3.4), for more detail see [8]. There are several common defuzzification techniques such as the weighted average formula (WAF), the center-of-area, maxima methods, etc., (see [1]). However, Zadeh [63] pointed to the problem that different defuzzifications have different relative performance measures with different benchmark tests, and there is no general method which can gain satisfactory performance in all conditions. For instance, the COG method is computationally difficult for complex membership functions since it is working on the area of the aggregated output fuzzy set and as a consequence it requires integral evaluations. However, we would like to use a simple method that is less computationally time consuming so we introduce a new defuzzification method similar to the WAF in Section 4.3.



COG	$U = \dfrac{\int_{\mathrm{Min}}^{\mathrm{Max}} u\,\mu(u)\,du}{\int_{\mathrm{Min}}^{\mathrm{Max}} \mu(u)\,du}$

Figure 3.4: An example of the center-of-gravity defuzzification method. -

3.2 Cellular Automata

A cellular automaton represents a discrete dynamic system consisting of a lattice of cells that change their states depending on the states of their neighbors, according to a local update rule. Formally, a CA consists of the 4-tuple

$$(\mathcal{L}, S, N, f)$$

where:

- $\mathcal{L}$ is a discrete lattice,
- S is a (finite) set of cell states. In literature, the cellular automata where S is an infinite set are referred as continuous cellular automata (CCA) or couple map lattices (CML).
- N is the neighborhood, which is a function from the lattice to the set of subsets of the lattice $\mathcal{L}$ of dimension m for some fixed integer $m \geq 0$, i.e., $N : \mathcal{L} \to P_m(\mathcal{L})$.
- $f : S^m \to S$ is the local update rule (local transition function).

Frequently, the lattice is a finite or infinite discrete regular grid of cells on a finite number of dimensions. Each cell is defined by its state and by its discrete position in the lattice. Time is also discrete, so the cells change their states synchronously at discrete time steps. The future state of a cell (at time $t+1$) is a function of the present state (at time t) of a finite number of cells surrounding the observed cell called the neighborhood (N). In other words, the next state of each cell depends on the current states of the neighboring cells. The neighboring cells may be the nearest cells surrounding the cell, but more general neighborhoods can also be specified. Although CA neighborhoods in general may be defined arbitrarily, for the CA we will examine, the same relative neighborhood is defined for each cell. For instance, if the neighborhood of one cell consists of the two immediately adjacent cells, then this is also true for all other cells. Such CA are called as *homogeneous*. The instantaneous situation (the configuration, see Subsection 3.2.1) of a cellular automaton is completely specified by the states of each cell.

To summarize, cellular automata are;

- discrete in both space and time,
- homogeneous in space and time (same update rule at all cells at all times),
- local in their interactions.

In literature, a two dimensional cellular automaton usually consists of a uniform grid that can be square, hexagon or triangle. For a square grid, its neighborhood can be von Neumann neighborhood, which comprises the four cells orthogonally surrounding a central cell on a two-dimensional square lattice. Beside the von Neumann neighborhood, some of the most common neighborhoods of two dimensional lattice are the Moore neighborhood and the extended Moore neighborhood (with radius=2), see Figure 3.5.

3.2.1 Some Basic Definitions

In this section, we briefly state some basic definitions that are needed in the sequel. For a survey on the subject see [29].

Let $\mathcal{A} = (\mathbb{Z}^d, S, N, f)$ be a CCA. Note that from now on we will always consider cellular automata whose lattices are d-dimensional grids $\mathbb{Z}^d$ for some integer d. A

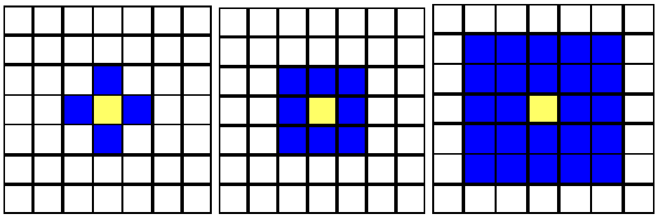


Figure 3.5: Common Neighborhoods - von Neumann, Moore and Extended Moore Neighborhood, respectively.

d-dimensional *neighborhood* (of size m) is a function

$$N : \mathbb{Z}^d \to P_m(\mathbb{Z}^d),$$

with

$$N(v) = (v + v_1, ..., v + v_m)$$

where $v \in \mathbb{Z}^d$ and $v_1, ..., v_m$ are fixed vectors with, $v_i \neq v_j$ for all $i \neq j, 1 \leq i, j \leq m$.

The local transition function (or the local rule) of a cellular automaton with a set of states S and size m neighborhood is defined by,

$$f : S^m \to S$$

that specifies the new state of each cell based on the old states of its neighbors.

A *configuration* of $\mathcal{A}$ is a function

$$c : \mathbb{Z}^d \to S$$

that specifies the states of all cells, i.e., $c(v)$ is the state of the cell $v \in \mathbb{Z}^d$. The set of all the configurations $S^{\mathbb{Z}^d}$ is denoted by $\mathcal{C}onf(\mathcal{A})$. A configuration should be interpreted as an instantaneous description of all the states in the system of cells at some moment of time.

Suppose that $\mathcal{A}$ is in the configuration c at time t and the cell in position v have neighbors $N(v) = (v + v_1, ..., v + v_m)$ with states $\sigma_i = c(v + v_i)$ for $i \in [1, m]$. The value $f(\sigma_1, ..., \sigma_m)$ is the state of the cell v at time $t + 1$. Therefore the local rule f determines the global dynamics of $\mathcal{A}$ and so we can extend f to a function

$$f^* : \mathcal{C}onf(\mathcal{A}) \to \mathcal{C}onf(\mathcal{A}),$$

called the *global transition function* of $\mathcal{A}$, which transforms c into the new configuration $f^*(c)$. The dynamic of $\mathcal{A}$ with initial configuration c is thus given by the iterated application of the global transition rule f^*. In this way we obtain an evolution of the system,

$$c \to f^*(c) \to f^{*^2}(c) \to f^{*^3}(c) \to \ldots \to f^{*^t}(c) \to \ldots$$

where the configuration at time step t is $f^{*^t}(c)$. From the computational point of view f^* can be computed only for particular configurations, namely, the configurations defined on a finite number of cells. For this purpose, we assume that S contains a special symbol $\perp$ representing the fact that the cells with this state must not be computed (in the sequel, we call such cells as empty cells). Formally, if a cell v is in the state $c(v) =\perp$ then $f^*(c)(v) =\perp$. A particular configuration is the empty configuration $\mathfrak{e}$ defined by $\mathfrak{e}(v) =\perp$ for all $v \in \mathbb{Z}^d$. The *support* of a configuration c is the set

$$Supp(c) := \{v \in \mathbb{Z}^d : \ c(v) \neq\perp\}.$$

The set of configurations with finite (compact) supports is denoted by $\mathcal{C}onf_c(\mathcal{A})$, it is easy to see that f^* restricts to $f^* : \mathcal{C}onf_c(\mathcal{A}) \to \mathcal{C}onf_c(\mathcal{A})$. Therefore this space is the natural environment to consider when we deal with the dynamic of $\mathcal{A}$ from the computational point of view. In the particular case $d = 1$ for any $c \in \mathcal{C}onf_c(\mathcal{A})$ there is a minimum integer i and a maximum integer j such that $c(k) =\perp$ for all $k \notin [i, j]$. In this way, we can represent c as a finite vector $(\sigma_i, ..., \sigma_j)$, and with this notation we can write the empty configuration as $\mathfrak{e} = ()$.

3.2.2 First CA Traffic Model Example: The Wolfram 184 Model

As we mentioned in Subsection 2.2.1, Rule 184 derived from Wolfram's naming scheme is one of the binary ECAs that are classified in [60]. In this deterministic ECA model (see [16]), a road is defined as a one-dimensional array with binary states where black square represents a state of 1 (an occupied cell) and a white square represents a state of 0 (an empty cell).

Let $\mathcal{W}$ be the Wolfram model defined as:

$$\mathcal{W} = (\mathbb{Z}, \mathcal{S}, \mathcal{N}, \mathcal{F})$$

where

- The lattice is set of integers.
- $\mathcal{S} = \mathbb{Z}_2 = \{0, 1\}$ is the set of cell states with:

  $$s_i(t) = (b_i(t))$$

  where the Boolean flag b_i represents the occupancy of a cell.
- The neighborhood of radius 1 is $N(i) = (i-1, i, i+1)$.
- The local transition function

  $$\mathcal{F} : \mathbb{Z}_2^3 \to \mathbb{Z}_2$$

  with $\mathcal{F} : (s_{i-1}(t), s_i(t), s_{i+1}(t)) \mapsto s_i(t+1)$ where $s_i(t)$ is the state of a central cell i at time step t, together with the states $s_{i-1}(t)$ and $s_{i+1}(t)$ of its two adjacent neighbors $i-1$ and $i+1$, respectively.

The graphical representation in Figure 3.6 provides us an illustration of the evolution of $\mathcal{F}$. For instance, if the local transition function maps (010) onto a state of 0, this has the physical meaning that a particle (black square) moves to the right if its neighbor cell is empty, leaving the central cell empty.

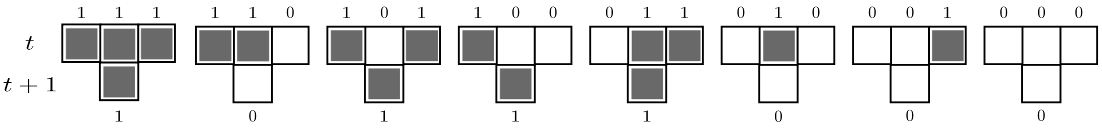


Figure 3.6: A graphical representation of the local transition function of Wolfram's rule 184 - All possible 8 configurations are set in descending order, representing the evolution of all cell's states in time, each based on its two adjacent neighbors.

3.2.3 Second CA Traffic Model Example: The NaSch Model

In literature, regarding the NaSch-type models, it is claimed that these kind of models are based on CA. However, it seems that there is no paper where the model is written using the standard formalization of a CA as a 4-tuple. Instead, the model is always presented in a standard way as a collection of rules stating how a vehicle, occupying one cell, should move.

In this subsection, we first describe the deterministic variant of the NaSch model (with open boundaries) where the randomization part introduced to slow down the vehicles at random is not considered (see Algorithm 1), and secondly, we describe the stochastic NaSch model (with open boundaries) with the randomization step (see Algorithm 2) which both are presented as a CA model.

Let $\mathcal{NS}$ be the deterministic NaSch CA defined as:

$$\mathcal{NS} = (\mathbb{Z}, \Theta, N, F)$$

where:

- The lattice is set of integers[1].
- $\Theta = \{0, \ldots, 5\} \times \{0, 1\} \cup \{\perp\}$ is the set of cell states with:

  $$\theta_i = (v_i, b_i)$$

  where $\{0, \ldots, 5\}$ represents the velocities of vehicles, and the Boolean flag represents the occupancy of a cell.
- The neighborhood is $N(i) = (i - 5, \ldots, i, \ldots, i + 5)$ and the function

  $$d(\theta_i, \ldots, \theta_{i+5}) = \min\{1 \leq k \leq 5 : b_{i+k} = 1\}$$

  returns the front distance of a vehicle.
- The local transition function

  $$F(\omega_{-5}, \ldots, \omega_0, \ldots, \omega_5)$$

  where $\omega_i = (v_i, b_i)$ for $-5 \leq i \leq 5$, is defined by Algorithm 1.

Note that the stochastic NaSch CA model requires a bigger set of states than the deterministic one since we need to record the information for the probability of slowing down to make this information available for all cells.

Let $\mathcal{NS}'$ be the stochastic NaSch CA defined as:

$$\mathcal{NS}' = (\mathbb{Z}, \Theta', N, F')$$

where:

---

[1] The closed boundary version has a finite set of integers and the local transition rule has to be modified to adapt to the periodic condition.

Algorithm 1 The pseudo-code for the one time step evolution of the local transition function F of the deterministic NaSch CA model.

```
procedure F($\omega_{-5}, \ldots, \omega_0, \ldots, \omega_5$)
    if $\omega_0 = \perp$ then
        $\omega_0 := \perp$
    else
        if $b_0 = 0$ then
            for $i = -1 \to -5$ do
                if $b_i = 1$ then
                    $d_i = d(\omega_i, \ldots, \omega_{i+5})$
                    $s_i = \min\{5, v_i + 1, d_i - 1\}$
                    if $s_i = |i|$ then
                        $v_0 := s_i$
                        $b_0 := 1$
                        break
                    end if
                end if
            end for
        else
            $d_0 = d(\omega_0, \ldots, \omega_5)$
            $s_0 = \min\{5, v_0 + 1, d_0 - 1\}$
            if $s_0 \neq 0$ then
                $v_0 := 0$
                $b_0 := 0$
            end if
        end if
    end if
end procedure
```

- The lattice is set of integers.
- $\Theta' = \{0, \ldots, 5\} \times \{0, 1\} \times \{0, 1\} \times \{0, 1\} \cup \{\perp\}$ is the set of cell states with:

  $$\theta'_i = (v_i, b_i, B_i, r_i)$$

  where $\{0, \ldots, 5\}$ represents the velocities of vehicles, the first Boolean flag represents the occupancy of a cell, the second one represents if we apply a Bernoulli process or not, and the last one represents the result of the Bernoulli process.
- The neighborhood is $N(i) = (i - 5, \ldots, i, \ldots, i + 5)$ and the function

  $$d(\theta'_i, \ldots, \theta'_{i+5}) = \min\{1 \leq k \leq 5 : b_{i+k} = 1\}$$

  returns the front distance of a vehicle.
- The local transition function

  $$F'(\omega'_{-5}, \ldots, \omega'_0, \ldots, \omega'_5)$$

  where $\omega'_i = (v_i, b_i, B_i, r_i)$ for $-5 \leq i \leq 5$, is defined by Algorithm 2.

Algorithm 2 The pseudo-code for the one time step evolution of the local transition function F' of the stochastic NaSch CA model.

```
procedure F'($\omega_{-5}, \ldots, \omega_0, \ldots, \omega_5$)
    if $B_0 = 1$ then
        Execute the Bernoulli Trial $r_0 \sim \mathcal{B}(2, p)$
        $B_0 := 0$
    else
        if $\omega_0 = \perp$ then
            $\omega_0 := \perp$
        else
            if $b_0 = 0$ then
                for $i = -1 \to -5$ do
                    if $b_i = 1$ then
                        $d_i = d(\omega'_i, \ldots, \omega'_{i+5})$
                        $s_i = \min\{5, v_i + 1, d_i - 1\} - r_i$
                        if $s_i = |i|$ then
                            $v_0 := s_i$
                            $b_0 := 1$
                            break
                        end if
                    end if
                end for
            else
                $d_0 = d(\omega'_0, \ldots, \omega'_5)$
                $s_0 = \min\{5, v_0 + 1, d_0 - 1\} - r_0$
                if $s_0 \neq 0$ then
                    $v_0 := 0$
                    $b_0 := 0$
                end if
            end if
        end if
    end if
end procedure
```

Chapter 4

A New Approach to Single-Lane CA Traffic Models via CCA

In this chapter, we first give the motivations for introducing a new single-lane CA traffic model, then we describe it in details in Section 4.2. This model has a different conception since we detach from the idea which is central in the other CA traffic models where cells represent the road space. Our approach uses the idea that cells represent vehicles, see Figure 4.1. The advantage of this vision is having much less cells to represent the same physical situation. Furthermore, with this approach we are able to introduce for the first time the continuity of space for a CA traffic model by using a CCA. In this context, our model is more close to the usual microscopic traffic flow models which adopt a semi-continuous space, formed by the usage of floating-point numbers compared to the classical CA traffic models (NaSch-type) where the space is coarse-grained. In this way, we are able to take the advantages of both usual microscopic models and the CA models which are computationally efficient and have fast performance. Moreover, in contrast to the gaseous models, the particles in the CA models can be intelligent and able to learn from past experience [12, 26, 59]. Therefore using the continuity introduced in our model, we have the possibility of incorporating behavioral and psychological aspects (e.g., stress) into the model using a multi-agent system based on fuzzy logic.

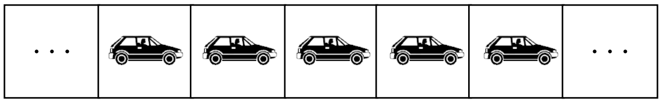


Figure 4.1: A representation of the fact that cells represent vehicles -

4.1 Introduction: Why a Different Model

In literature of the NaSch-type CA models, an important parameter to deal with is the physical representation of the cells. In the attempt of increasing the resolution of the model to have a smoother (finer) simulation, the lengths of the cells are decreased [9, 25, 33, 48]. In this way, the degree of freedom in the simulation is increased. For instance in the NaSch model [43], a vehicle has only 5 possible velocities while in the NaSch-type models where the length of a cell is less than 7.5 $m.$, this number representing the possible velocities is increased. However, decreasing the physical dimension of cells means increasing the number of cells if the length of the road is kept constant. Also if we want to embed the physical dimension of a vehicle into the model, this vehicle will not fit anymore in only one cell when the dimension of cells are decreased. Thus, there will be the need of more cells to represent this vehicle, e.g., in [33] each vehicle has a length of 5 cells. Furthermore, the dimension of the neighborhood must be increased when the cells are scaled, and in the ideal limit when the length of a cell tends to zero, the number of cells clearly tends to infinity with also the length of the neighborhood. Consequently, a bigger number of cells means a larger computational time even though the number of vehicles is kept constant, and the passage to the limit case when the space is continuous is clearly impossible to be implemented on a computer in the realm of NaSch-type models. Therefore, a natural question is whether or not it is possible to find a different CA traffic model in which the space is not anymore discrete but continuous. This question is interesting not only from the theoretical point of view, but also it makes possible to simulate driver behaviors by using a fuzzy logic-based system.

We now summarize our motivations and the features requested from the model as:

- We want a traffic CA model where the space is continuous or at least finer without an extreme rise on the number of cells.

- In the NaSch-type models, decisions are discrete, not smooth, not realistic. For instance in the NaSch model, vehicles accelerate independently of their velocity. The acceleration of one step is of 7.5 m/s^2, and vehicles can come to a stop from a maximum speed of 37.5 m/s in one second. Instead dealing with floating points, numbers for parameters such as velocity, space and acceleration in general can be useful in implementing a fuzzy logic-based system to try to mimic real driver behaviors.

- Using a fuzzy logic-based system, we can group the vehicles into types where they share common characteristics such as the same perception of distances. In this way we can study the influence of the heterogeneity of driver behaviors in road traffic.

- Open boundary CA traffic models of the NaSch-type where there is no destruction of vehicles imply an infinite number of cells (at least a big number if we want to approximate it with a closed boundary model). Therefore the question is whether or not it is possible to define an open boundary CA traffic model where the number of cells is equal to the number of vehicles. In this way, we also do not have any empty cell[1] to be computed like in the NaSch-type models.

- In terms of neighborhood concept, the NaSch-type models do not fit inside the CA models with the common classification of neighborhood (e.g., von Neumann, see Section 3.2). It easy to see that in such models the length of the neighborhood depends on the maximum velocity of the vehicles. The question is whether or not it is possible to define a CA model where the neighborhood is a constant number that does not depend on parameters, since the dimension of the neighborhood influences the computational speed.

Fitting in these requests has a “price” of passing from a CA model to a continuous CA model.

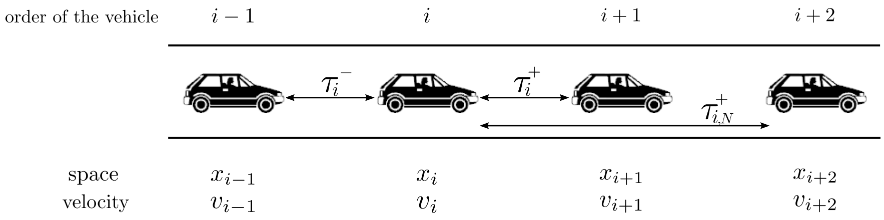


Figure 4.2: The illustration of the back vehicle, front vehicle and next front vehicle with respect to the i-th vehicle. -

4.2 Description of the Model

Our CA model of a single-lane traffic flow is a time-discrete, *space continuous* cellular automaton model which is combined with fuzzy decision rules with the purpose of simulating driver-vehicle behavior as much as possible. In our CCA model each non-empty cell corresponds to one vehicle, see Figure 4.1 and Figure 4.2 for the illustrations of the single-lane model. The proposed model is described as following (note that for the sake of simplicity, we fix the unit of time as one second, $u = 1\,\text{sec}$):

Consider the CCA model

$$\mathcal{SL} = (\mathbb{Z}, \Sigma, \mathcal{N}, \delta)$$

where:

- The discrete lattice is the set of integers.
- $\Sigma = (K \times \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \mathbb{R} \times \{L, 0, R\} \times \{L, 0, R\}) \cup \{\perp\}$ is the infinite set of cell states with the state of the i-th vehicle at time step t defined as:

  $$\sigma_i(t) = (k_i, x_i(t), v_i(t), s_i(t), d_i(t), d'_i(t)),$$

  where:

  - k_i represents the kind of the i-th vehicle. For instance, in our experiments we take into consideration two kinds of vehicles which are passenger vehicles and long vehicles (see Section 6.2). It is possible to introduce as many kinds as desired in the model, e.g., depending on the driver's characteristics (old, young, slow, fast, nervous, relaxed), types of the vehicles (passenger, trucks,

---

[1] In this context, we refer to the term "empty cell" which means a cell not occupied by any vehicle as in [43], so the cell is not really empty as defined in Subsection 3.2.1.

sport cars), weather conditions (good, bad) and time of the day (day-time, night-time). The kind consists of all the information (parameters) specified differently for each kind of vehicle, such as:

  - ∗ the maximum velocity (v_{max}), note that not necessarily there is speed limit,
  - ∗ the optimal velocity (v_{opt}): the velocity specified for each kind of vehicle with which they feel comfortable in traffic. It is introduced with the assumption that not all the vehicles move with the maximum velocity but with their optimal velocity,
  - ∗ the length (l_i),
  - ∗ the fuzzy membership functions (see Section 4.3),
  - ∗ the natural acceleration noise (A_N): a random variable defined with Gaussian distribution as, $A_N(t) = \mathcal{N}(0, \sigma^2)$, see [23].
  - ∗ the maximum stress (s_{max}),
  - ∗ the minimum stress (s_{min}),
  - ∗ the function used to calculate the probability of lane-changing to the right lane ($P_R(x)$),
  - ∗ the function used to calculate the probability of lane-changing to the left lane ($P_L(x)$).

- $x_i(t)$ is the position of the i-th vehicle, which is defined as the distance from the origin of the road to the middle point of this vehicle.

- $v_i(t)$ is the velocity of the i-th vehicle.

- $s_i(t)$ is the stress of the i-th vehicle, which is a variable to keep track of how much the driver is above or below of his optimal velocity. It is introduced to implement a more realistic driver behavior since in reality drivers tend to decelerate (or to change the lane in the multi-lane case) when they are moving with a velocity higher (or lower) than their optimal velocity.

- $d_i(t)$ is the variable which describes the desire of the i-th driver for:
  - ∗ lane-changing to the left, represented by L,
  - ∗ staying on his own-lane, represented by 0,
  - ∗ lane-changing to the right, represented by R.

- $d'_i(t)$ is the variable describing the trace of the i-th driver showing from which lane it is transferred, such as:
  - transferred from the left lane, represented by L,
  - not transferred, represented by 0,
  - transferred from the right lane, represented by R.

Note that for the purpose of the single-lane CCA model, there is no usage of the variables $d_i(t)$ and $d'_i(t)$, and $s_i(t)$ is used just to slow down the i-th vehicle in the case that it goes too much above its optimal velocity.

- $\mathcal{N}$ is a kind of one-dimensional extended Moore neighborhood, consisting of the cell itself, its adjacent cells and one next adjacent cell defined by $\mathcal{N} : \mathbb{Z} \to \mathcal{P}_4(\mathbb{Z})$ such that
$$\mathcal{N}(i) = (i-1, i, i+1, i+2).$$
- δ is the local transition function (local rule) defined by $\delta : \Sigma^4 \to \Sigma$ such that
$$\delta(\sigma_{i-1}(t), \sigma_i(t), \sigma_{i+1}(t), \sigma_{i+2}(t)) = (k_i, x_i(t+1), v_i(t+1), s_i(t+1), d_i(t+1), 0).$$

This rule acts upon a cell and its direct neighborhoods such that the cell's state changes from one discrete time step to another (i.e., the system's iterations). The CCA evolves in time and space as the rule is subsequently applied to all the cells in parallel. In our model, we define the position and the velocity of the i-th vehicle at time $t+1$ as following:

$$x_i(t+1) = x_i(t) + v_i(t+1),$$

$$v_i(t+1) = \min(v_{max}, \Delta x_i^+(t), \max(0, v_i(t) + A_i(t) + A_N(t))),$$

where $A_i(t)$ is the acceleration calculated by the fuzzy decision modules which are described in Section 4.3, and $\Delta x_i^+(t)$ is the distance with front vehicle defined in Equation 4.1. The constraint $\Delta x_i^+(t)$ in the choice of the updated velocity is introduced to make the model collision-free as in [43]. However, it is unrealistic and the application of it in our model is used in the borderline situations where extreme decelerations are involved. Indeed we tried to avoid this constraint as much as possible by introducing more fuzzy rules which makes the system more reactive to reduce these extreme situations. Finally, the stress and the desire of lane-changing of the i-th vehicle at time $t+1$ are defined as:

$$s_i(t+1) = AddStr(k_i, s_i(t), v_i(t), v_{i+1}(t), x_i(t), x_{i+1}(t)),$$

$$d_i(t+1) = Eval_{(\mathcal{L},\mathcal{R})}(k_i, v_i(t), s_i(t)),$$

where $AddStr$ and $Eval_{(\mathcal{L},\mathcal{R})}$ are explained in detail in Section 5.1 related to the multi-lane CCA model. The two parameters $\mathcal{L}, \mathcal{R}$ are Boolean variables which are used to describe whether or not there exists a lane on the left or on the right, respectively. For instance, $(\mathcal{L}, \mathcal{R}) = (0, 1)$ represents a lane having no lane on its left, but having a lane on its right. Since $\mathcal{SL}$ depends on these two parameters, we can write

$$\mathcal{SL}_{(\mathcal{L},\mathcal{R})} = (\mathbb{Z}, \Sigma, \mathcal{N}, \delta_{(\mathcal{L},\mathcal{R})})$$

to make this dependency clear. This dependency is important only when we consider the multi-lane case, since the variable $d_i(t)$ which is updated depending on $(\mathcal{L}, \mathcal{R})$, is not used in the single-lane case. Therefore, from the simulation point of view of the single-lane case the models $\mathcal{SL}_{(a,b)},\ a, b \in \{0, 1\}$ are all equivalent, and so it is appropriate to use the notation $\mathcal{SL}$ only for the single-lane case.

We assume that $d'_i(t+1) = 0$, because at each time step we need to have a "reset" situation at the beginning of the changing lane process. This variable and its usage will also be explained in detail in Section 5.1.

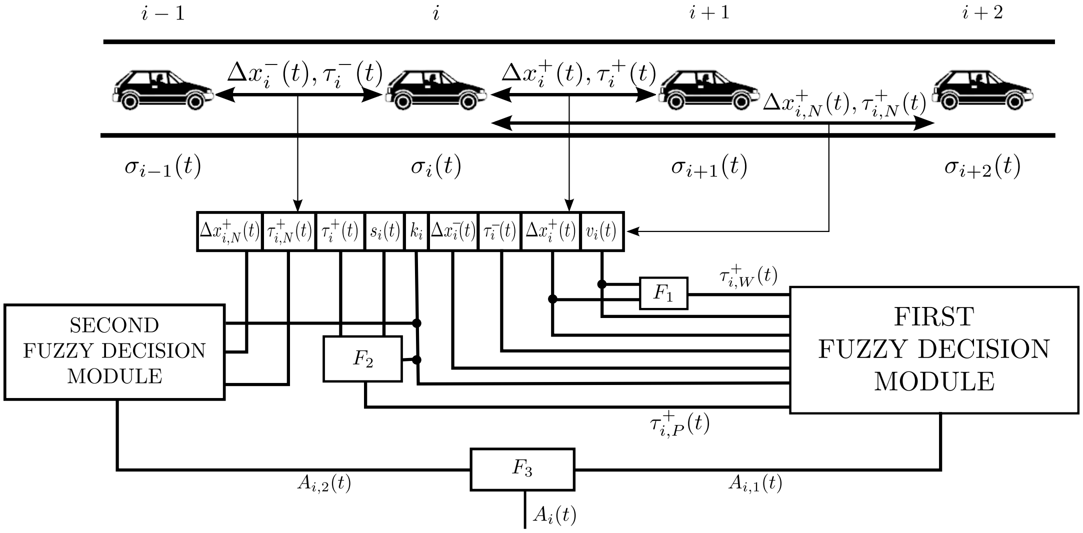


Figure 4.3: Block diagram of the decision process for the acceleration $A_i(t)$. -

The acceleration $A_i(t)$ is depending on the kind k_i, the velocity $v_i(t)$ and the variables defined as following (see Figure 4.3):

1. Back Distance (BD), the distance between $i-1$-th vehicle and i-th vehicle:

$$\Delta x_i^-(t) = x_i(t) - x_{i-1}(t) - \frac{l_i}{2} - \frac{l_{i-1}}{2}.$$

2. Front Distance (FD), the distance between i-th vehicle and $i+1$ -th vehicle:

$$\Delta x_i^+(t) = x_{i+1}(t) - x_i(t) - \frac{l_{i+1}}{2} - \frac{l_i}{2}. \tag{4.1}$$

3. Next Front Distance (NFD), the distance between i-th vehicle and i+2-th vehicle:

$$\Delta x_{i,N}^+(t) = x_{i+2}(t) - x_i(t) - \frac{l_{i+2}}{2} - \frac{l_i}{2}.$$

   Note that, in our model the distance between vehicles is considered as the distance from front bumper to rear bumper.

4. Perceived Front Collision Time ($PFCT$):

$$\tau_{i,P}^+(t) = \begin{cases} \zeta_i(t) & \text{if } \tau_i^+(t) < 0, \\ \min(\zeta_i(t), \tau_i^+(t)) & \text{otherwise.} \end{cases}$$

   (see the function F_2 in Figure 4.3) where $\zeta_i(t)$ is the parameter for slowing down that is depending on the stress as:

$$\zeta_i(t) = \frac{s_{max} - s_i(t)}{v_i(t)},$$

   and $\tau_i^+(t)$ is the Front Collision Time (FCT), the time that passes for the i-th vehicle to reach to (to collide with) the front vehicle. If it is negative it means that i-th vehicle is slower and will not reach to the front vehicle with the configuration at time t:

$$\tau_i^+(t) = \frac{\Delta x_i^+(t)}{v_i(t) - v_{i+1}(t)}.$$

   PFCT is a parameter which is a combination between the FCT and an auxiliary time defined to keep the velocity close to the optimal velocity v_{opt}.

5. Worst Front Collision Time ($WFCT$), the time that passes for i-th vehicle to collide with the front vehicle in the case where the front vehicle suddenly stops (introduced for safety reasons). This is calculated by:

$$\tau^+_{i,W}(t) = \frac{\Delta x^+_i(t)}{v_i(t)},$$

(see the function F_1 in Figure 4.3).

6. Next Front Collision Time ($NFCT$), the time that passes for i-th vehicle to reach to the next front ($i+2$-th) vehicle. It is clear that a vehicle never reaches to its next front vehicle, but $NFCT$ is introduced to anticipate the braking manoeuvre in the case where the next front vehicle suddenly brakes:

$$\tau^+_{i,N}(t) = \frac{\Delta x^+_{i,N}(t)}{v_i(t) - v_{i+2}(t)}.$$

7. Back Collision Time (BCT), the time that passes for $i-1$-th vehicle to reach to the front (i-th) vehicle:

$$\tau^-_i(t) = \frac{\Delta x^-_i(t)}{v_{i-1}(t) - v_i(t)}.$$

This quantity is introduced to take into account the phenomenon where the back driver forces the front driver to accelerate which we call as pushing effect.

In the case the cell $i-1$ is empty, we assume that the back distance and the back collision time of the i-th vehicle converge to infinity (in the simulation process, we use a sufficiently fixed big number). Instead, if the cells $i+1$ and $i+2$ are empty, we assume that the front distance, the next front distance, the front collision time and the next front collision time of the i-th vehicle converge to infinity. If only the cell $i+2$ is empty, we assume that the next front distance and the next front collision time of the i-th vehicle converge to infinity.

In the sequel, we consider configurations with a physical meaning. In other words, these configurations are formed in such a way that the order of the cells have the same physical order of the vehicles according to their positions. It is easy to see that the set consisting of such configurations which will play an important role in Chapter 5 has the property $\Delta x^+_i(t) \geq 0$ for any cell i. We denote this set by $\mathcal{C}onf_p(\mathcal{SL}_{(\mathcal{L},\mathcal{R})})$ or simply $\mathcal{C}onf_p(\mathcal{SL})$, since $\mathcal{C}onf_p(\mathcal{SL}_{(a,b)}),\ a, b \in \{0,1\}$ are all the same set.

Example Let $c, c' \in \mathcal{C}onf_c(\mathcal{SL})$ such that $c = (\sigma_0, \sigma_1)$ and $c' = (\omega_0, \omega_1)$ where $\sigma_0 = \omega_1 = (k_0, 5, v_0, s_0, d_0, d'_0)$ and $\sigma_1 = \omega_0 = (k_1, 10, v_1, s_1, d_1, d'_1)$, see Figure 4.4. Clearly, $c \in \mathcal{C}onf_p(\mathcal{SL})$ and $c' \notin \mathcal{C}onf_p(\mathcal{SL})$.

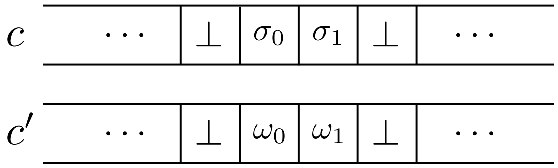


Figure 4.4: The representation of two configurations c and c'. -

4.3 Fuzzy Decision Modules

The approach of embedding fuzzy logic while dealing with a system described by continuous variables is already introduced, indeed, there are several works based on fuzzy logic systems in car-following models [7] and in lane-changing models [18].

In this section, we define the sets of fuzzy rules that determine the behavior of the vehicles in traffic stream. More specifically, the fuzzy modules, after receiving the inputs determined by the environment, return the decided acceleration ($A_i(t)$ for the i-th vehicle at time t) that is obtained by the two fuzzy sets of rules at each time step (see Figure 4.3). These rules are formed based on some common sense of driver behaviors including some experiences and examples. Although the rules are the same for each kind of vehicle, they have different "weights" depending on the definition of the membership functions of different kinds. In this way, it is possible to give a description of a variety of behaviors, such as the behavior of a long vehicle driver or a sportive driver or a person with low reflexes. For instance, a person with low reflexes perceives a time of collision of 5 seconds as a very short time, however a person with higher reflexes probably would feel comfortable with that time of collision.

As we mentioned before, there are three steps in the fuzzy decision module which will determine the act of the driver (decision making) depending on the kind. These steps, the fuzzifier, the fuzzy inference and the defuzzification are described as following:

4.3.1 Fuzzifier

In the fuzzifier step, each crisp value (input) $\tau_{i,P}^{+}$(t), $\tau_{i,W}^{+}(t)$, $\tau_{i,N}^{+}(t)$, $\tau_{i}^{-}(t)$, $\Delta x_{i}^{+}(t)$, $\Delta x_{i,N}^{+}(t)$, $\Delta x_{i}^{-}(t)$, $V_i(t)$, for each vehicle i at time t is transformed into fuzzy values, which essentially means associating an input to a degree of having a property involved in the fuzzy rules. For example, if we receive the information from the environment such that the perceived front collision time of the i-th vehicle at time t is given as, $\tau_{i,P}^{+}(t) = 9$ *sec.* which we decide as a big time with the truth-value of 0.8, and assume that $\mu_{BIG}(x)$ is the membership function of the property "being big", then the fuzzifier simply transforms the crisp value 9 into the pair $(9, \mu_{BIG}(9))$ which is $(9, 0.8)$. All the fuzzy values must be decided so that they can be used for evaluating the weights of each fuzzy rule.

4.3.2 Fuzzy Inference

We note that in this subsection, to avoid cumbersome notations, we omit the dependency of t from all the variables.

The fuzzy system involves eight inputs: "perceived front collision time ($\tau_{i,P}^{+}$)", "worst front collision time ($\tau_{i,W}^{+}$)", "next front collision time ($\tau_{i,N}^{+}$)", "back collision time (τ_{i}^{-})", "front distance (Δx_{i}^{+})", "next front distance ($\Delta x_{i,N}^{+}$)", "back distance (Δx_{i}^{-})", and "velocity (V_i)". We need the "velocity" as an input only for jam situations.

The $WFCT$ is introduced to keep a safety distance between the vehicles. In other words, it is used in the case where the front vehicle suddenly stops, so we can assume that if the $WFCT$ is very small, there is a dangerous situation. Moreover, the $NFCT$ and the NFD are introduced to have a better perception of the driver behaviors of the next front vehicle since in reality drivers observe not only the vehicle just in front of them but also the vehicles ahead.

The set of rules of the first module have more importance since they are based on the information related to the front and the back vehicles while the other set of rules have less importance since they are related to the next front vehicle. As a consequence of having two different fuzzy sets of rules, we have two fuzzy outputs: the output came out from the first set of rules is the "first acceleration value ($A_{i,1}$)" and the output came out from the second set of rules is the "second acceleration value ($A_{i,2}$)". After the defuzzification step we evaluate the final decision A_i which is determined by a function of $A_{i,1}$ and $A_{i,2}$ (see Figure 4.3). Let us define this function as:

$$F_3(A_{i,1}, A_{i,2}) = \begin{cases} \min(A_{i,1}, A_{i,2}) & \text{if } A_{i,1} \leq 0, \\ \frac{A_{i,1}+A_{i,2}}{2} & \text{if } A_{i,1} > 0 \wedge A_{i,2} \leq -0.25, \\ A_{i,1} & \text{otherwise.} \end{cases} \tag{4.2}$$

where we give more weight to the decision taken by the first module. Indeed, in the simulation, we have noticed that without these kind of constraints the vehicles were tending to slow down too much. This is also the reason of splitting the fuzzy module into two parts. Moreover, the second module and as a consequence the second acceleration gains significance only in the case of emergencies such as a sudden breakdown or deceleration of the next front vehicle.

In regard to introducing the membership functions below, we could have used different membership functions for the three different front collision times, and for the two different front distances. However, for the sake of simplicity, we use the same membership functions with respect to the properties for all types of front collision times, and similarly for both types of front distances. Also because for instance, in reality the perception of the time of collision of the next front vehicle would be the same with that of the front vehicle if there would not exist the front vehicle since in this case the next front vehicle would take the place of the front vehicle. Similarly, we define the same membership functions with respect to the properties for both $A_{i,1}$ and $A_{i,2}$. Let us define now the input fuzzy sets in our model:
The input fuzzy sets for both $PFCT$ and $NFCT$ are,

$$FrCT\ VERY\ SMALL, FrCT\ SMALL, FrCT\ MEDIUM, FrCT\ BIG$$

The input fuzzy set of BCT is,

$$BackCT\ VERY\ SMALL$$

The input fuzzy sets for both FD and NFD are,

$$FrD\ VERY\ SMALL, FrD\ SMALL, FrD\ MEDIUM, FrD\ BIG$$

The input fuzzy set of BD is,

$$BackD\ VERY\ SMALL$$

The input fuzzy set of V is,

$$VEL\ SMALL$$

The output fuzzy sets for the accelerations in both modules are,

$$ACC\ PB, ACC\ PM, ACC\ PS, ACC\ Z, ACC\ NS, ACC\ NM, ACC\ NB$$

where

$$PB = POSITIVE\ BIG, PM = POSITIVE\ MEDIUM,$$
$$PS = POSITIVE\ SMALL, Z = ZERO, NS = NEGATIVE\ SMALL,$$
$$NM = NEGATIVE\ MEDIUM, NB = NEGATIVE\ BIG$$

and the zero acceleration means “keeping the velocity constant”. While listing the fuzzy set of rules, we use some shortcuts. For instance, instead of writing “perceived front collision time is $FrCT\ SMALL$” we write the shorter notation “$PFCT$ is $SMALL$”.

First Fuzzy Set of Rules (First Fuzzy Decision Module):

Rule 1: IF $PFCT$ is BIG AND FD is BIG AND V is NOT $SMALL$ THEN A is PM.

Rule 2: IF $PFCT$ is BIG AND FD is $MEDIUM$ AND V is NOT $SMALL$ THEN A is PS.

Rule 3: IF $PFCT$ is BIG AND FD is $SMALL$ THEN A is Z. (Zero Acceleration)

Rule 4: IF $PFCT$ is BIG AND FD is $VERY\ SMALL$ THEN A is Z.

Rule 5: IF $PFCT$ is $MEDIUM$ AND FD is BIG THEN A is Z.

Rule 6: IF $PFCT$ is $MEDIUM$ AND FD is $MEDIUM$ THEN A is Z.

Rule 7: IF $PFCT$ is $MEDIUM$ AND FD is $SMALL$ THEN A is NS.

Rule 8: IF $PFCT$ is $MEDIUM$ AND FD is $VERY\ SMALL$ THEN A is NS.

Rule 9: IF $PFCT$ is $SMALL$ AND FD is BIG THEN A is NM.

Rule 10: IF $PFCT$ is $SMALL$ AND FD is $MEDIUM$ THEN A is NM.

Rule 11: IF $PFCT$ is $SMALL$ AND FD is $SMALL$ THEN A is NM.

Rule 12: IF $PFCT$ is $SMALL$ AND FD is $VERY\ SMALL$ THEN A is NM.

Rule 13: IF $PFCT$ is $VERY\ SMALL$ AND FD is BIG THEN A is NB.

Rule 14: IF $PFCT$ is $VERY\ SMALL$ AND FD is $MEDIUM$ THEN A is NB.

Rule 15: IF $PFCT$ is $VERY\ SMALL$ AND FD is $SMALL$ THEN A is NB.

Rule 16: IF $PFCT$ is $VERY\ SMALL$ AND FD is $VERY\ SMALL$ THEN A is NB.

Rule 17: IF BCT is $VERY\ SMALL$ AND BD is $VERY\ SMALL$ AND $PFCT$ is BIG AND FD is BIG THEN A is PS. (Pushing Effect)

Rule 18: IF BCT is $VERY\ SMALL$ AND BD is $VERY\ SMALL$ AND $PFCT$ is BIG AND FD is $MEDIUM$ THEN A is PS.

Rule 19: IF BCT is $VERY\ SMALL$ AND BD is $VERY\ SMALL$ AND $PFCT$ is $MEDIUM$ AND FD is BIG THEN A is PS.

Rule 20: IF BCT is $VERY\ SMALL$ AND BD is $VERY\ SMALL$ AND $PFCT$ is $MEDIUM$ AND FD is $MEDIUM$ THEN A is PS.

Rule 21: IF $PFCT$ is BIG AND V is $SMALL$ THEN A is PB. (The vehicle is in a jam situation)

Rule 22: IF $WFCT$ is $VERY\ SMALL$ AND FD is $VERY\ SMALL$ THEN A is NM.

Rule 23: IF $WFCT$ is $VERY\ SMALL$ AND FD is $SMALL$ THEN A is NM.

Rule 24: IF $WFCT$ is $VERY\ SMALL$ AND FD is $MEDIUM$ THEN A is NS.

Second Fuzzy Set of Rules (Second Fuzzy Decision Module):

Rule 1: IF $NFCT$ is $VERY\ SMALL$ and NFD is $VERY\ SMALL$ THEN A is NB.

Rule 2: IF $NFCT$ is $VERY\ SMALL$ and NFD is $SMALL$ THEN A is NB.

Rule 3: IF $NFCT$ is $VERY\ SMALL$ and NFD is $MEDIUM$ THEN A is NB.

Rule 4: IF $NFCT$ is $VERY\ SMALL$ and NFD is BIG THEN A is NM.

Rule 5: IF $NFCT$ is $SMALL$ and NFD is $VERY\ SMALL$ THEN A is NM.

Rule 6: IF $NFCT$ is $SMALL$ and NFD is $SMALL$ THEN A is NM.

Rule 7: IF $NFCT$ is $SMALL$ and NFD is $MEDIUM$ THEN A is NS.

Rule 8: IF $NFCT$ is $SMALL$ and NFD is BIG THEN A is NS.

Rule 9: IF $NFCT$ is $MEDIUM$ and NFD is $VERY\ SMALL$ THEN A is NS.

Rule 10: IF $NFCT$ is BIG and NFD is $VERY\ SMALL$ THEN A is NS.

We now evaluate the fuzzy decision rules according to the compositional rule of inference by using the degree of memberships determined in the fuzzifier step. Therefore we obtain a result (an output fuzzy set) for each rule by taking the minimum value of the images of the inputs in each rule. We describe this evaluation in general terms as following:

Let $B_1^j, ..., B_{k_j}^j$ and C^j be fuzzy subsets with the membership functions $\mu_{B_1^j}, \dots, \mu_{B_{k_j}^j}$ and μ_{C^j}, respectively, and let R^j be the fuzzy rules defined as:

$$R^j : IF\ x_1^j \text{ is } B_1^j\ AND\ \dots\ AND\ x_{k_j}^j \text{ is } B_{k_j}^j\ THEN\ y\ is\ C^j$$

with

$$\mu_{B_1^j}(x_1^j) \wedge\ \dots\ \wedge \mu_{B_{k_j}^j}(x_{k_j}^j) = \min\left\{\mu_{B_1^j}(x_1^j),\ \dots\ , \mu_{B_{k_j}^j}(x_{k_j}^j)\right\}$$

for $1 \leq j \leq m$, where m is the number of fuzzy rules.

In our fuzzy system, there are also rules including the form: x_r^j is not B_r^j with the degree of membership $\mu_{\sim B_r^j}(x_r^j)$ for some $r \in [1, k_j]$. Recall from Section 3.1.1 that for an input x_r^j, the degree of membership of not having a property B_r^j ("not being B_r^j") can be written as:

$$\mu_{\sim B_r^j}(x_r^j) = 1 - \mu_{B_r^j}(x_r^j).$$

Note that this is required only for the first two rules of the first fuzzy decision module with the purpose of emphasizing the vehicle is not in a jam situation.

4.3.3 Defuzzification

The output of the inference process so far is a fuzzy set, so we should convert our fuzzy output set obtained from the fuzzy inference step into one single number as the output of the fuzzy system, which is the "acceleration" in our case. Recall that since we have two fuzzy modules, there are two outputs of the system: $A_{i,1}$ and $A_{i,2}$ which are used to evaluate $A_i(t)$ at time t by the function F_3, see Equation 4.2. The driver's decision making system for his velocity of the next time step is updated by this acceleration value.

As we mentioned in Subsection 3.1.2 , there are many defuzzification techniques to obtain a crisp value, such as the center-of-gravity (COG) and the weighted average formula (WAF) method. In our model, since we would like to use a simple method that does not require too much computational power, we define a new method and call it as generalized weighted average formula (GWAF), i.e., we generalize the WAF to the case where the membership functions are not necessarily symmetric. We describe it now in general terms by using the notations used in the previous subsection:

Suppose that each j-th rule receives the values $\overline{x}_1^j, \ldots, \overline{x}_{k_j}^j$ as inputs. Let

$$w^j = \min\left\{\mu_{B_1^j}\,(\overline{x}_1^j),\ \ldots\ , \mu_{B_{k_j}^j}\,(\overline{x}_{k_j}^j)\right\}$$

be the weights for each rule j and let $P^j = \mu_{C^j}^{-1}(w^j)$ be the preimage of the weight of j-th rule. We assume that μ_{C^j} does not have any plateau, since we do not want P^j to contain intervals (this is done to have a discrete set). The defuzzified output is thus calculated by:

$$\overline{y} = \frac{\sum\limits_{j=1}^{m} w^j \sum\limits_{z \in P^j} z}{\sum\limits_{j=1}^{m} |P^j|\, w^j}.$$

Chapter 5

A Multi-Lane Stochastic CCA Traffic Model

In this chapter, we extend the single-lane model to the case of multi-lane. This extension is not trivial as it is in the NaSch-type models where adding a lane means simply adding an array of cells and where the local transition function can naturally be extended. This is a consequence of having a clear physical interpretation of the model given by the fact that space is represented by cells.

In our single-lane model, cells represent vehicles and the union of two arrays of cells that represent a road with two lanes is a natural candidate for a multi-lane model. However, since we do not have the cell-space correspondence as it is in NaSch-type models, we do not have the natural order which makes the extension of the local transition function so easy to achieve. For this reason, we first present our multi-lane model as a union of interacting single-lane CCA where the interaction is a transfer operation, and then we prove in Proposition 5.3.1 that this model can actually be simulated by a stochastic CCA.

The process of lane-changing is based on safety criterion which checks the possibility of executing a lane-changing by considering the situation in the target lane. This criterion guarantees that after the lane-changing, there will be no danger (avoidance of collision) and as less disturbing as possible with the back and the following vehicle in the target lane. This criterion used in our model is described in Section 5.2 together with some operators useful in describing the lane-changing process. Moreover, we assume that the precedence for the lane-changing is given to the vehicles moving from the left to the right, since on the highways that apply European rules where the overtaking is only allowed on the left, right-most lane is the slowest and the left-most lane is the fastest lane.

In Chapter 6 we have implemented our multi-lane stochastic CCA model (see Appendix A for the code) with some other extra features like on- and off-ramps and on- and off-toll plazas to test this model. However, for the sake of simplicity, we do not introduce these features into the mathematical formulation of the model described in this chapter.

5.1 The Update of Stress and the Desire of Lane-Changing

In this section, we describe the functions

$$d_i(t+1) = Eval_{(\mathcal{L},\mathcal{R})}(k_i, v_i(t), s_i(t))$$
$$s_i(t+1) = AddStr(k_i, s_i(t), v_i(t), v_{i+1}(t), x_i(t), x_{i+1}(t))$$

which we mentioned in Section 4.2 to update at each step respectively the desire of lane-changing and the stress. The decision process for lane-changing is depending on the stress variable $s_i(t)$ representing how much the driver is above or below of his optimal velocity v_{opt}. If the driver has positive stress meaning that he is driving too much above of his optimal velocity, the driver would tend to change the lane to the right (slower lane). If the stress is negative meaning that the driver is nervous and he desires to go faster to recover the distance that he has lost going slower than his optimal velocity, then he would tend to change the lane to the left (faster lane). We first describe how the stress is updated. Let us define the accumulated stress as,

$$s_{acc}(t) = s_i(t) + (v_i(t+1) - v_{opt}) \cdot \mathbb{X}(t)$$

where $\mathbb{X}(t)$ is a random variable distributed uniformly, $\mathbb{X}(t) \sim \mathcal{U}(0,1)$. Therefore we calculate a stress parameter,

$$str_i(t+1) = \begin{cases} \xi(s_{acc}(t), k_i, \tau_i^+(t), \Delta x_i^+(t)) & \text{if } \frac{s_{min}}{2} < s_{acc}(t) < 0, \\ s_{acc}(t) & \text{otherwise.} \end{cases}$$

where $\tau_i^+(t), \Delta x_i^+(t)$ are respectively the FCT and the FD defined previously in Section 4.2, ξ will be described below in detail and s_{min} is the maximum amount of negative stress that a driver can tolerate. Another consideration that we take into account is that the stress cannot be increased (or decreased) arbitrarily. For this reason, we have two parameters inside the kind k_i: s_{max} (the maximum amount of positive stress that a driver can tolerate) and s_{min}, limiting the stress parameter from above and below. Thus the update is performed in the following way:

$$AddStr(k_i, s_i(t), v_i(t), v_{i+1}(t), x_i(t), x_{i+1}(t)) = \begin{cases} s_{min} & \text{if } str_i(t+1) < s_{min}, \\ s_{max} & \text{if } str_i(t+1) > s_{max}, \\ str_i(t+1) & \text{otherwise.} \end{cases}$$

The function $\xi(s_{acc}(t), k_i, \tau_i^+(t), \Delta x_i^+(t))$ is defined to avoid too much frequent changes of lane in the case of a jam situation, if the queue is moving. We achieve this task by simply decreasing the stress from s_{acc} to $s_{acc}/2$ (see Equation 5.1). We also embed the strategy of trying to change lane instead of braking in the case the front vehicle is close and tends to brake. We model this effect considering a factor Φ which is calculated by the membership functions for the fuzzy sets $FrCT\ VERY\ SMALL, FrCT\ SMALL$ and $FrD\ MEDIUM, FrD\ SMALL$ (we do not consider the case $FrD\ VERY\ SMALL$ because in this case the maneuver of lane-changing could be dangerous). Let us denote by $\mu_{tvs}, \mu_{ts}, \mu_{dm}, \mu_{ds}$ the membership functions of $FrCT\ VERY\ SMALL$, $FrCT\ SMALL$, $FrD\ MEDIUM$, $FrD\ SMALL$, respectively. The factor Φ, which represents how much the i-th vehicle is close to the situation where the front vehicle is close and it is going much more slower than the i-th vehicle, can be calculated by:

$$\begin{aligned} \Phi \quad = \quad & \max(\min\{\mu_{tvs}(\tau_i^+(t)), \mu_{dm}(\Delta x_i^+(t))\}, \min\{\mu_{tvs}(\tau_i^+(t)), \mu_{ds}(\Delta x_i^+(t))\}, \\ & \min\{\mu_{ts}(\tau_i^+(t)), \mu_{dm}(\Delta x_i^+(t))\}, \min\{\mu_{ts}(\tau_i^+(t)), \mu_{ds}(\Delta x_i^+(t))\}) \end{aligned}$$

This value is simply the membership function of the formula,

$$\begin{gathered} (\tau_i^+(t) \text{ is } FrCT\ VERY\ SMALL \wedge \Delta x_i^+(t) \text{ is } FrD\ MEDIUM)\vee \\ (\tau_i^+(t) \text{ is } FrCT\ VERY\ SMALL \wedge \Delta x_i^+(t) \text{ is } FrD\ SMALL)\vee \\ (\tau_i^+(t) \text{ is } FrCT\ SMALL \wedge \Delta x_i^+(t) \text{ is } FrD\ MEDIUM)\vee \\ (\tau_i^+(t) \text{ is } FrCT\ SMALL \wedge \Delta x_i^+(t) \text{ is } FrD\ SMALL) \end{gathered}$$

and it is used to increase the amount of stress. In this way the vehicle will have more probability of changing lane. Hence ξ can be calculated by:

$$\xi(s_{acc}(t), k_i, \tau_i^+(t), \Delta x_i^+(t)) = \begin{cases} \frac{s_{acc}(t)}{2} & \text{if } \tau_i^+(t) < 0, \\ s_{acc}(t) \cdot (1 + \Phi) & \text{if } \tau_i^+(t) \geq 0. \end{cases} \tag{5.1}$$

We now describe the stochastic function $Eval_{(\mathcal{L},\mathcal{R})}(k_i, v_i(t), s_i(t))$ which returns the desired action (moving to the left or right lane, or staying in the same lane) at each time step. The decision of this action is made by means of a Bernoulli process $\mathcal{B}(2, p)$. The probabilities of changing lane to the left (p_L) and to the right (p_R) which are in general different, are calculated by two functions $P_L(x) : [0, 1] \to [0, 1]$ and $P_R(x) : [0, 1] \to [0, 1]$ contained in the kind k_i. The reason is that some vehicles tend to change lane more than others. For instance, long vehicles prefer to go to right lane more than left, i.e., they tend to stay more on the right lane. The variable used to calculate such probabilities is the normalized stress $ns_i(t)$ defined as,

$$ns_i(t) = \begin{cases} \frac{s_i(t)}{s_{max}} & \text{if } s_i(t) \geq 0, \\ \\ \frac{s_i(t)}{s_{min}} & \text{otherwise.} \end{cases}$$

In the decision process, we take into consideration also the jam situation where simply the driver randomly moves to the left or to the right for the purpose of finding an emptier lane. In this case, the parameters $(\mathcal{L}, \mathcal{R})$ are used to make a decision. More precisely, if there is no lane on the left of the driver, i.e., $\mathcal{L} = 0$, clearly he moves to the right lane, similarly if there is no lane on the right of the driver, i.e., $\mathcal{R} = 0$, clearly he moves to the left lane. If there are lanes both on the left and right, the choice of which lane he will move to is obtained by means of a Bernoulli process $\mathcal{B}(2, 0.7)$ where the decision is made as: moving to the left lane with the probability of 0.7 and moving to the right lane otherwise. These probabilities are decided to be different since we assume that the drivers that want to go faster usually tend to move to the left lane more than the right lane.

Another consideration is the evaluation of whether or not there is a jam situation. A parameter used to evaluate this situation on a highway can be the velocity, indeed if the velocity of the vehicle is small, with high probability it means that this vehicle is jammed. We need the fuzzy set $VEL\ SMALL$ with the membership function denoted by μ_{vels}, to perform this evaluation which is done by means of a Bernoulli process $\mathcal{B}(2, \mu_{vels}(v_i(t)))$. We present the pseudo-code which returns the value $Eval_{(\mathcal{L},\mathcal{R})}(k_i, v_i(t), s_i(t))$ in Algorithm 3.

Algorithm 3 The pseudo-code for evaluating $Eval_{(\mathcal{L},\mathcal{R})}$.

```
procedure $Eval_{(\mathcal{L},\mathcal{R})}(k_i, v_i(t), s_i(t))$
    if $s_i(t) \geq 0$ then
        $ns_i(t) = s_i(t)/s_{max}$
        Execute the Bernoulli Trial $\mathbb{X} \sim \mathcal{B}(2, P_R(ns_i(t))$
        if $\mathbb{X} = 1$ then
            $d_i(t+1) = R$
        else
            $d_i(t+1) = 0$
        end if
    else
        $ns_i(t) = s_i(t)/s_{min}$
        Execute the Bernoulli Trial $\mathbb{Y} \sim \mathcal{B}(2, P_L(ns_i(t))$
        if $\mathbb{Y} = 1$ then
            Execute the Bernoulli Trial $\mathbb{Z} \sim \mathcal{B}(2, \mu_{vels}(v_i(t)))$
            if $\mathbb{Z} = 1$ then
                if $\mathcal{R} = 0$ then
                    $d_i(t+1) = L$
                end if
                if $\mathcal{L} = 0$ then
                    $d_i(t+1) = R$
                end if
                Execute the Bernoulli Trial $\mathbb{W} \sim \mathcal{B}(2, 0.7)$
                if $\mathbb{W} = 1$ then
                    $d_i(t+1) = L$
                else
                    $d_i(t+1) = R$
                end if
            else
                $d_i(t+1) = L$
            end if
        else
            $d_i(t+1) = 0$
        end if
    end if
end procedure
```

5.2 The Lane-Changing Process

In this section, we describe the conditions for a vehicle to perform a lane-changing and some basic operators which will be useful to describe the multi-lane model presented in Section 5.3. These operators are fundamental to describe the transfer of vehicles from a lane to an adjacent one, and for the description of the CCA the transfer is seen as a copying and erasing process.

Deciding on whether or not to perform a lane-changing is depending on two steps. We first check if a driver desires to change lane. If a lane-changing is indeed desirable, then the second step proceeds to check if it is possible to perform such a lane-changing with respect to safety and collision avoidance.

The transfer of a vehicle from one lane to another clearly changes the configuration of the single-lane CA, thus we need to introduce some operations to describe this process. Let $c \in \mathcal{C}onf(\mathcal{SL})$ and let $\sigma \in \Sigma$ be a state, the *inserting operator* at position $i \in \mathbb{Z}$ is the function

$$Ins_i : \Sigma \setminus \{\perp\} \times \mathcal{C}onf(\mathcal{SL}) \to \mathcal{C}onf(\mathcal{SL}),$$

such that

$$Ins_i(\sigma, c)(j) = \begin{cases} c(j) & \text{if } j < i, \\ \sigma & \text{if } j = i, \\ c(j-1) & \text{if } j > i. \end{cases}$$

This operator simply shifts all the cells starting at the i-th position of one step and it sets the state of the i-th cell to the value σ. The right-inverse of Ins_i is the *deleting operator* at position i which is the function

$$Del_i : \mathcal{C}onf(\mathcal{SL}) \to \mathcal{C}onf(\mathcal{SL}),$$

such that

$$Del_i(c)(j) = \begin{cases} c(j) & \text{if } j < i, \\ c(j+1) & \text{if } j \geq i. \end{cases}$$

The index i gives the position where we insert or erase the content of the i-th cell, and physically, it depends on the relative position of the vehicle represented by the state σ. For this reason we need to consider the configurations which have a physical meaning where the order of the cells is in correspondence with the physical order of the vehicles. Thus, from now on we consider the configurations belonging to $\mathcal{C}onf_p(\mathcal{SL})$ (see Section 4.2).

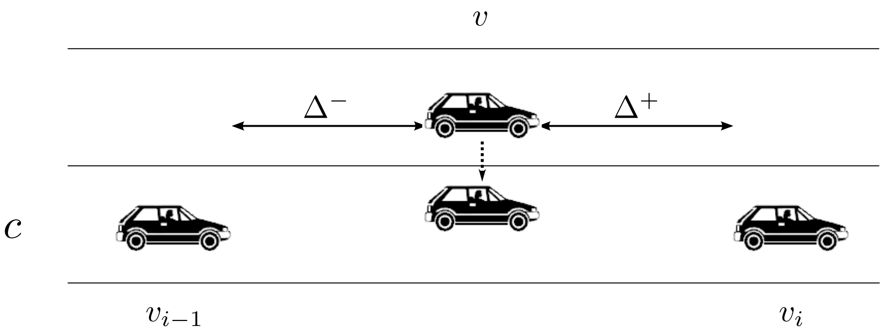


Figure 5.1: Inserting a vehicle into a lane with the configuration c. -

Given a vehicle represented by a state $\sigma \in \Sigma \setminus \{\perp\}$ and a configuration $c \in \mathcal{C}onf_p(\mathcal{SL})$, we define the *index operator* as the function

$$Indx : \Sigma \times \mathcal{C}onf_p(\mathcal{SL}) \to \mathbb{Z},$$

which returns the relative position of σ with respect to the vehicles in c. Suppose that $\sigma = (k, x, v, s, d, d')$ and $c = (\sigma_m, \ldots, \sigma_M)$ where $\sigma_j = (k_j, x_j, v_j, s_j, d_j, d'_j)$ for $j \in [m, M]$, then

$$Indx(\sigma, c) = \begin{cases} M+1 & \text{if } x \geq x_j, \forall j \in [m, M], \\ \min\{j \in [m, M] : x \leq x_j\} & \text{otherwise.} \end{cases}$$

In the case c is the empty configuration $\mathfrak{e}$, we define $Indx(\sigma, \mathfrak{e}) = 0$. The integer $i = Indx(\sigma, c)$ represents the index where the vehicle with state σ would go if we would try to insert into the configuration c. However, there are some restrictions. Suppose that the vehicle represented by σ has length l, and the $i-1$-th and i-th vehicles with the states σ_{i-1}, σ_i have lengths l_{i-1}, l_i, respectively[1]. The front and the back distances of a vehicle (see Figure 5.1) with state σ with respect to σ_{i-1} and σ_i are defined by:

$$\begin{aligned} \Delta^+ &= x_i - x - \frac{l_i}{2} - \frac{l}{2}, \\ \Delta^- &= x - x_{i-1} - \frac{l}{2} - \frac{l_{i-1}}{2}. \end{aligned}$$

The first condition of performing a lane-changing is due to a physical reason because of the fact the vehicle should fit between the $i-1$-th and the i-th vehicles. This condition is expressed by the following inequalities:

$$\Delta^+ > 0, \ \Delta^- > 0. \tag{5.2}$$

---

[1] The border cases where either $\sigma_{i-1} = \perp$ or $\sigma_i = \perp$ can be treated analogously.

Clearly this is a basic condition while dealing with the transfer of a vehicle from one lane to another one. However, there must be some conditions that a driver has to take into consideration when he is changing the lane. These conditions inspired by the rules in [24] and also by some experiments in our simulation, are introduced for safety reasons (avoiding accidents) in the following way:

$$\Delta^- > v_{i-1}^{1.2} - v + |v_{i-1} - v| + 3, \tag{5.3}$$

$$\Delta^+ > v^{1.25} - v_i + 3. \tag{5.4}$$

In general, the inequality 5.3 has more importance for the vehicles that are entering from the right to the left lane (faster lane) and the inequality 5.4 has more importance for the vehicles that are entering from the left to the right lane (slower lane). The last two conditions that we impose for the vehicle with the state $\sigma = (k, x, v, s, d, d')$ to be transferred into the configuration c is that σ must have the desire to change lane, i.e., $d \neq 0$ and σ must have not already been transferred, i.e., $d' = 0$. The latter condition is done to avoid multiple transfer of a vehicle in one computation step of the multi-lane CCA that we will describe later, e.g., a situation where a vehicle jumps from the first lane to the third lane instantaneously. We sum all the conditions that σ has to fulfill to be transferred into c defining the Boolean operator $trans(\sigma, c)$ which is true if and only if the following condition is satisfied:

$$(d \neq 0) \wedge (d' = 0) \wedge (\Delta^- > 0) \wedge (\Delta^- > v_{i-1}^{1.2} - v + |v_{i-1} - v| + 3) \wedge \\ (\Delta^+ > 0) \wedge (\Delta^+ > v^{1.25} - v_i + 3). \tag{5.5}$$

Depending on where a vehicle wants to be transferred, we define the Boolean operators $trans_L(\sigma, c)$ and $trans_R(\sigma, c)$ which are true if and only if this vehicle wants to be transferred to the left and to the right, respectively. Formally:

$$trans_L(\sigma, c) \Longleftrightarrow trans(\sigma, c) \wedge (d = L),$$
$$trans_R(\sigma, c) \Longleftrightarrow trans(\sigma, c) \wedge (d = R).$$

In the process of lane-changing, we need to keep trace where the vehicle comes from. This is done because the transfer of a vehicle is seen as a process consisting of two steps, firstly we copy the state σ into c, and secondly we have to erase the original state σ. For this purpose we need the following notation, we define the copy of σ as:

$$\sigma^{cp} = \begin{cases} (k, x, v, s/5, L, R) & \text{if } \sigma = (k, x, v, s, L, 0), \\ (k, x, v, s/5, R, L) & \text{if } \sigma = (k, x, v, s, R, 0). \end{cases}$$

where the stress is reduced from s to $s/5$ because we make the assumption that when a vehicle changes lane there is a sense of satisfaction which reduces the stress. Another reason of decreasing the stress in the process of changing lane is to avoid, or at least to reduce, the ping-pong phenomenon (frequent lane-changings, see [32, 41, 50]) especially in the congestion situation.

Suppose that we want to transfer the vehicle with the state $\sigma = (k, x, v, s, d, d')$ into the configuration c. In the sequel we need to discriminate the states to transfer. For instance, if we want to transfer to the left (right) lane all the vehicles that desire to be transferred, we need to copy only the ones that have a desire to move on the left (right) lane. For this reason, if we want to copy σ into c only if it desires to go to the left (right), we have to update c into a new configuration denoted by $\sigma \rightarrowtail_L c$ ($\sigma \rightarrowtail_R c$) defined by the following equations:

$$\sigma \rightarrowtail_L c = \begin{cases} Ins_i(\sigma^{cp}, c) & \text{if } trans_L(\sigma, c),\ \text{where } i = Indx(\sigma, c) \\ c & \text{otherwise.} \end{cases}$$

$$\sigma \rightarrowtail_R c = \begin{cases} Ins_i(\sigma^{cp}, c) & \text{if } trans_R(\sigma, c),\ \text{where } i = Indx(\sigma, c) \\ c & \text{otherwise.} \end{cases}$$

Note that $\sigma \rightarrowtail_L c, \sigma \rightarrowtail_R c \in \mathcal{C}onf_p(\mathcal{SL})$.

On the other hand, if we consider σ as the state of a configuration c' which is on the left or on the right of the configuration c, it is clear that we need to update c' into a new configuration $c' \setminus \sigma$ by simply erasing the state σ that has already been copied into c.

In general, assume that we have a state $\omega = (h, y, w, r, p, p')$ of a vehicle, then ω is a copy of a vehicle at position $i = Indx(\omega, c')$ in the configuration c' coming from the left (right) if and only if $c'(i) = (h, y, w, 5r, R, 0)$ and $\omega = (h, y, w, r, R, L)$ ($c'(i) = (h, y, w, 5r, L, 0)$ and $\omega = (h, y, w, r, L, R)$). Thus the erasing procedure is accomplished by changing c' into the new configuration defined by:

$$c' \setminus \omega = \begin{cases} Del_i(c') & \begin{array}{r} \text{if } (c'(i) = (h, y, w, 5r, R, 0) \wedge \omega = (h, y, w, r, R, L)) \\ \vee(c'(i) = (h, y, w, 5r, L, 0) \wedge \omega = (h, y, w, r, L, R)), \\ \text{where } i = Indx(\omega, c') \end{array} \\ c' & \text{otherwise.} \end{cases}$$

Note that $c' \setminus \omega \in \mathcal{C}onf_p(\mathcal{SL})$.

We now extend the previous functions $\rightarrowtail_L, \rightarrowtail_R$ and $\setminus$ to operators

$$\rightarrowtail : \mathcal{C}onf_p(\mathcal{SL}) \times \mathcal{C}onf_p(\mathcal{SL}) \to \mathcal{C}onf_p(\mathcal{SL})$$

$$\setminus : \mathcal{C}onf_p(\mathcal{SL}) \times \mathcal{C}onf_p(\mathcal{SL}) \to \mathcal{C}onf_p(\mathcal{SL})$$

between the physical configurations to describe the operation of transfer of vehicles from a lane to another. We call $\rightarrowtail_L$ ($\rightarrowtail_R$) and $\setminus$ respectively the left (right) *copying* and *erasing* operator. Let $c, c' \in \mathcal{C}onf_p(\mathcal{SL})$ where $c = (\omega_{j_1}, \ldots, \omega_{j_M}), c' = (\sigma_{i_1}, \ldots, \sigma_{i_N})$, let us define the configurations $c' \rightarrowtail_L c$ inductively as follows: let $e_0 = c$ and $e_k := \sigma_{i_k} \rightarrowtail_L e_{k-1}$ for $k \in [1, N]$ then,

$$c' \rightarrowtail_L c := e_N$$

analogously for $\rightarrowtail_R$.

On the other hand, for $c' \setminus c$, let $g_0 = c'$ and $g_k := g_{k-1} \setminus \omega_{j_k}$ for $k \in [1, M]$ then,

$$c' \setminus c := g_M.$$

Using these operators it is easy to see that the process of the transfer of vehicles from c' to c, where c is the configuration of a lane on the left of the lane with configuration c', can be easily described by transforming c into $c' \rightarrowtail_L c$ and c' into $c' \setminus (c' \rightarrowtail_L c)$.

5.3 Description of the Multi-Lane Model

In this section, we present our model for a multi-lane road using the copying and erasing operators introduced in Section 5.2. Suppose that there are $M \geq 2$ number of lanes on a road. We model each lane using the single-lane CCA model defined in Section 4.2, thus we can associate to the road the ordered M-tuple:

$$\mathcal{SL}_{(0,1)}, \mathcal{SL}^1_{(1,1)}, \ldots, \mathcal{SL}^{M-2}_{(1,1)}, \mathcal{SL}_{(1,0)}$$

where if $M \geq 3$ we have $M - 2$ copies of $\mathcal{SL}_{(1,1)}$. Note that $\mathcal{SL}_{(0,1)}, \mathcal{SL}_{(1,0)}$ represents the left-most lane and the right-most lane, respectively. In the case $M = 2$, we consider just the pair $\mathcal{SL}_{(0,1)}, \mathcal{SL}_{(1,0)}$.

Suppose that these M number of CCA are in the configurations $(c_1, \ldots, c_M) \in \mathcal{C}onf_p(\mathcal{SL})^M$. In our multi-lane model, we scan each lane and we transfer the vehicles to the adjacent lanes. After this process, for each lane we apply the single-lane CCA model to update the configuration, this update is done by means of the global transition function (recall that the global transition function of $\mathcal{SL}_{(a,b)}$ is denoted by δ^*).

Algorithm 4 The pseudo-code for the one time step evolution of the multi-lane model.

```
procedure Update($c_0, \ldots, c_{M-1}$)
    for $i = 0 \to M-1$ do
        if $i = 0$ then
            $c_1 := (c_0 \rightarrowtail_R c_1)$
            $c_0 := c_0 \setminus (c_0 \rightarrowtail_R c_1)$
        end if
        if $0 < i < M-1$ then
            $c_{i-1} := (c_i \rightarrowtail_L c_{i-1})$
            $c_i := c_i \setminus (c_i \rightarrowtail_L c_{i-1})$
            $c_{i+1} := (c_i \rightarrowtail_R c_{i+1})$
            $c_i := c_i \setminus (c_i \rightarrowtail_R c_{i+1})$
        end if
        if $i = M-1$ then
            $c_{M-2} := (c_{M-1} \rightarrowtail_L c_{M-2})$
            $c_{M-1} := c_{M-1} \setminus (c_{M-1} \rightarrowtail_L c_{M-2})$
        end if
    end for
    for $i = 0 \to M-1$ do
        if $i = 0$ then
            $c_0 := \delta^*_{(0,1)}(c_0)$
        end if
        if $0 < i < M-1$ then
            $c_i := \delta^*_{(1,1)}(c_i)$
        end if
        if $i = M-1$ then
            $c_{M-1} := \delta^*_{(1,0)}(c_{M-1})$
        end if
    end for
end procedure
```

In this way we result with a new array of configurations $(c'_1, \dots, c'_M)$, and this process represents u seconds of simulation (recall that in Section 4.2 we made the assumption $u = 1$ sec.).

Moreover, the order with which the transfer is performed is from the left-most lane to the right-most one, and this is done to satisfy the precedence requirement in European roads. In Algorithm 4, it is described the updating process $Update : (c_1, \dots, c_M) \mapsto (c'_1, \dots, c'_M)$.

We now show that Algorithm 4 can be simulated by a CCA which implies that our multi-lane model is actually a CCA model. Thus we can conclude that it is possible to introduce a multi-lane CA model where the space is continuous. We state this fact in the following,

Proposition 5.3.1 *There is a stochastic CCA $\mathcal{ML}$ which simulates Algorithm 4.*

Proof We define the stochastic CCA:

$$\mathcal{ML} = (\mathbb{Z}, \Omega, \mathcal{M}, \Delta)$$

where

- $\Omega = (\mathcal{C}onf_p(\mathcal{SL}) \times \{copy, erase\} \times \mathbb{N}^3) \cup \{\perp\}$, where $\perp$ is the state associated to the empty cells[1].
- $\mathcal{M}(i) = (i-1, i, i+1)$ is the von Neumann neighborhood.
- $\Delta : \Omega^3 \to \Omega$ is the local transition function where:

  $$\Delta(\omega_{-1}, \omega_0, \omega_1) = \omega'_0$$

  defined in Algorithm 5, with

  $$\omega_j = (c_j, X_j, M_j, P_j, K_j),\ j = -1, 0, 1.$$

If we consider M lanes with the configurations $c_0, \dots, c_{M-1}$, we associate to $\mathcal{ML} = (\mathbb{Z}, \Omega, \mathcal{M}, \Delta)$ the configuration

$$\mathcal{C} = (\omega_0, \dots, \omega_{M-1})$$

---

[1] In this context, an empty cell represents the fact that in that cell there is no lane.

Algorithm 5 The pseudo-code to compute the local transition function Δ.

```
procedure $\Delta$ $(\omega_{-1}, \omega_0, \omega_1)$
    if $\omega_0 = \perp$ then
        $\omega'_0 = \perp$
    else
        if $K_0 = 0$ then
            if $X_0 = copy$ then
                if $P_0 = K_0 + 1$ then
                    $c_0 := (c_{-1} \rightarrowtail_R c_0)$
                end if
                $X_0 := erase$
            else
                if $P_0 = K_0$ then
                    $c_0 := (c_0 \setminus c_1)$
                end if
                $K_0 := K_0 + 1 \mod M_0$
                $X_0 := copy$
                exit
            end if
        end if
        if $0 < K_0 < M_0 - 1$ then
            if $X_0 = copy$ then
                if $P_0 = K_0 - 1$ then
                    $c_0 := (c_1 \rightarrowtail_L c_0)$
                end if
                if $P_0 = K_0 + 1$ then
                    $c_0 := (c_{-1} \rightarrowtail_R c_0)$
                end if
                $X_0 := erase$
            else
                if $P_0 = K_0$ then
                    $c_0 := (c_0 \setminus c_{-1})$
                    $c_0 := (c_0 \setminus c_1)$
                end if
                $K_0 := K_0 + 1 \mod M_0$
                $X_0 := copy$
                exit
            end if
        end if
```

```
        if $K_0 = M_0 - 1$ then
            if $X_0 = copy$ then
                if $P_0 = K_0 - 1$ then
                    $c_0 := (c_1 \rightarrowtail_L c_0)$
                end if
                $X_0 := erase$
            else
                if $P_0 = K_0$ then
                    $c_0 := (c_0 \setminus c_{-1})$
                end if
                if $P_0 = 0$ then
                    $c_0 := \delta^*_{(0,1)}(c_0)$
                end if
                if $0 < P_0 < M_0 - 1$ then
                    $c_0 := \delta^*_{(1,1)}(c_0)$
                end if
                if $P_0 = M_0 - 1$ then
                    $c_0 := \delta^*_{(1,0)}(c_0)$
                end if
                $K_0 := K_0 + 1 \mod M_0$
                $X_0 := copy$
            end if
        end if
    end if
end procedure
```

where $\omega_i = (c_i, copy, M, i, 0)$ for $i = 0, \ldots, M-1$. It is easy to see that applying $2M$ times the global transition function Δ^* to $\mathcal{C}$, we obtain a new configuration:

$$\Delta^{*^{2M}}(\mathcal{C}) = (\omega'_0, \ldots, \omega'_{M-1})$$

with $\omega'_i = (c'_i, copy, M, i, 0)$ for $i = 0, \ldots, M-1$ where

$$(c'_0, \ldots, c'_{M-1}) = Update(c_0, \ldots, c_{M-1}),$$

and $Update(c_0, \ldots, c_{M-1})$ is the function described in Algorithm 4.

□

The following example is an application of Algorithm 5 (an application of $\Delta^{*^{2M}}$ where $M = 3$). In the example, we denote “copy” as $\mathbf{c}$ and “erase” as $\mathbf{e}$.

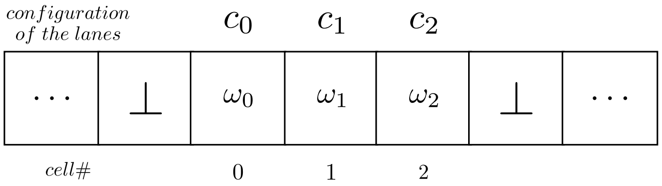


Figure 5.2: The configuration of $\mathcal{ML}$ of Example 5.3. -

Example Let $\mathcal{C} = (\omega_0, \omega_1, \omega_2)$ be the configuration of $\mathcal{ML}$ associated to c_0, c_1, c_2, respectively (see Figure 5.2) where,

$$\omega_0 = (c_0, \mathbf{c}, 3, 0, 0), \omega_1 = (c_1, \mathbf{c}, 3, 1, 0), \omega_2 = (c_2, \mathbf{c}, 3, 2, 0)$$

as defined in Proposition 5.3.1 then,

1. **cell 0:** $\Delta(\perp, (c_0, \mathbf{c}, 3, 0, 0), (c_1, \mathbf{c}, 3, 1, 0)) = (c_0, \mathbf{e}, 3, 0, 0)$

   cell 1: $\Delta((c_0, \mathbf{c}, 3, 0, 0), (c_1, \mathbf{c}, 3, 1, 0), (c_2, \mathbf{c}, 3, 2, 0)) = (c_1^{(1)}, \mathbf{e}, 3, 1, 0)$
   where,
   $$c_1^{(1)} = (c_0 \rightarrowtail_R c_1)$$

   cell 2: $\Delta((c_1, \mathbf{c}, 3, 1, 0), (c_2, \mathbf{c}, 3, 2, 0), \perp) = (c_2, \mathbf{e}, 3, 2, 0)$

   The new configuration: $\mathcal{C}^{(1)} = ((c_0, \mathbf{e}, 3, 0, 0), (c_1^{(1)}, \mathbf{e}, 3, 1, 0), (c_2, \mathbf{e}, 3, 2, 0))$.

2. **cell 0:** $\Delta(\perp, (c_0, \mathbf{e}, 3, 0, 0), (c_1^{(1)}, \mathbf{e}, 3, 1, 0)) = (c_0^{(1)}, \mathbf{c}, 3, 0, 1)$
   where,
   $$c_0^{(1)} = (c_0 \setminus c_1^{(1)})$$

   cell 1: $\Delta((c_0, \mathbf{e}, 3, 0, 0), (c_1^{(1)}, \mathbf{e}, 3, 1, 0), (c_2, \mathbf{e}, 3, 2, 0)) = (c_1^{(1)}, \mathbf{c}, 3, 1, 1)$

   cell 2: $\Delta((c_1^{(1)}, \mathbf{e}, 3, 1, 0), (c_2, \mathbf{e}, 3, 2, 0), \perp) = (c_2, \mathbf{c}, 3, 2, 1)$

   The new configuration: $\mathcal{C}^{(2)} = ((c_0^{(1)}, \mathbf{c}, 3, 0, 1), (c_1^{(1)}, \mathbf{c}, 3, 1, 1), (c_2, \mathbf{c}, 3, 2, 1))$.

   Note that these two steps simulate a transfer(copy-erase) from the lane represented by cell 0 to the lane represented by cell 1 (see Figure 5.3).

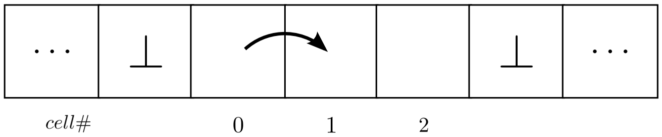


Figure 5.3: The first transfer. -

3. **cell 0:** $\Delta(\perp, (c_0^{(1)}, \mathbf{c}, 3, 0, 1), (c_1^{(1)}, \mathbf{c}, 3, 1, 1)) = (c_0^{(2)}, \mathbf{e}, 3, 0, 1)$
   where,
   $$c_0^{(2)} = (c_1^{(1)} \rightarrowtail_L c_0^{(1)})$$
   cell 1: $\Delta((c_0^{(1)}, \mathbf{c}, 3, 0, 1), (c_1^{(1)}, \mathbf{c}, 3, 1, 1), (c_2, \mathbf{c}, 3, 2, 1)) = (c_1^{(1)}, \mathbf{e}, 3, 1, 1)$

   cell 2: $\Delta((c_1^{(1)}, \mathbf{c}, 3, 1, 1), (c_2, \mathbf{c}, 3, 2, 1), \perp) = (c_2^{(1)}, \mathbf{e}, 3, 2, 1)$
   where,
   $$c_2^{(1)} = (c_1^{(1)} \rightarrowtail_R c_2)$$
   The new configuration: $\mathcal{C}^{(3)} = ((c_0^{(2)}, \mathbf{e}, 3, 0, 1), (c_1^{(1)}, \mathbf{e}, 3, 1, 1), (c_2^{(1)}, \mathbf{e}, 3, 2, 1)$.

4. **cell 0:** $\Delta(\perp, (c_0^{(2)}, \mathbf{e}, 3, 0, 1), (c_1^{(1)}, \mathbf{e}, 3, 1, 1)) = (c_0^{(2)}, \mathbf{c}, 3, 0, 2)$

   cell 1: $\Delta((c_0^{(2)}, \mathbf{e}, 3, 0, 1), (c_1^{(1)}, \mathbf{e}, 3, 1, 1), (c_2^{(1)}, \mathbf{e}, 3, 2, 1)) = (c_1^{(3)}, \mathbf{c}, 3, 1, 2)$
   where,
   $$c_1^{(2)} = (c_1^{(1)} \setminus c_0^{(2)})$$
   $$c_1^{(3)} = (c_1^{(2)} \setminus c_2^{(1)})$$
   cell 2: $\Delta((c_1^{(1)}, \mathbf{e}, 3, 1, 1), (c_2^{(1)}, \mathbf{e}, 3, 2, 1), \perp) = (c_2^{(1)}, \mathbf{c}, 3, 2, 2)$

   The new configuration: $\mathcal{C}^{(4)} = ((c_0^{(2)}, \mathbf{c}, 3, 0, 2), (c_1^{(3)}, \mathbf{c}, 3, 1, 2), (c_2^{(1)}, \mathbf{c}, 3, 2, 2)$.

   Note that the third and the fourth steps simulate a transfer(copy-erase) from the lane represented by cell 1 to the lanes represented by cell 0 and 2 (see Figure 5.4).

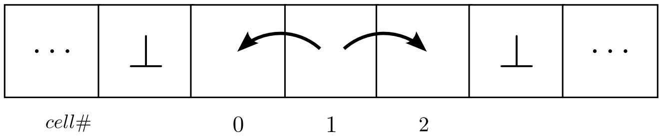


Figure 5.4: The second transfer. -

5. **cell 0:** $\Delta(\perp, (c_0^{(2)}, \mathbf{c}, 3, 0, 2), (c_1^{(3)}, \mathbf{c}, 3, 1, 2)) = (c_0^{(2)}, \mathbf{e}, 3, 0, 2)$

cell 1: $\Delta((c_0^{(2)}, \mathbf{c}, 3, 0, 2), (c_1^{(3)}, \mathbf{c}, 3, 1, 2), (c_2^{(1)}, \mathbf{c}, 3, 2, 2) = (c_1^{(4)}, \mathbf{e}, 3, 1, 2)$

where,

$$c_1^{(4)} = (c_2^{(1)} \rightarrowtail_L c_1^{(3)})$$

cell 2: $\Delta((c_1^{(3)}, \mathbf{c}, 3, 1, 2), (c_2^{(1)}, \mathbf{c}, 3, 2, 2), \perp) = (c_2^{(1)}, \mathbf{e}, 3, 2, 2)$

The new configuration: $\mathcal{C}^{(5)} = ((c_0^{(2)}, \mathbf{e}, 3, 0, 2), (c_1^{(4)}, \mathbf{e}, 3, 1, 2), (c_2^{(1)}, \mathbf{e}, 3, 2, 2)$.

6. **cell 0:** $\Delta(\perp, (c_0^{(2)}, \mathbf{e}, 3, 0, 2), (c_1^{(4)}, \mathbf{e}, 3, 1, 2)) = (c_0^{(3)}, \mathbf{c}, 3, 0, 0)$

where,

$$c_0^{(3)} = \delta^*_{(0,1)}(c_0^{(2)})$$

cell 1: $\Delta((c_0^{(2)}, \mathbf{e}, 3, 0, 2), (c_1^{(4)}, \mathbf{e}, 3, 1, 2), (c_2^{(1)}, \mathbf{e}, 3, 2, 2)) = (c_1^{(5)}, \mathbf{c}, 3, 1, 0)$

where,

$$c_1^{(5)} = \delta^*_{(1,1)}(c_1^{(4)})$$

cell 2: $\Delta((c_1^{(4)}, \mathbf{e}, 3, 1, 2), (c_2^{(1)}, \mathbf{e}, 3, 2, 2), \perp) = (c_2^{(3)}, \mathbf{c}, 3, 2, 0)$

where,

$$c_2^{(2)} = (c_2^{(1)} \setminus c_1^{(4)})$$

$$c_2^{(3)} = \delta^*_{(1,0)}(c_2^{(2)})$$

The new configuration: $\mathcal{C}^{(6)} = ((c_0^{(3)}, \mathbf{c}, 3, 0, 0), (c_1^{(5)}, \mathbf{c}, 3, 1, 0), (c_2^{(3)}, \mathbf{c}, 3, 2, 0)$.

Finally, these last two steps simulate a transfer(copy-erase) from the lane represented by cell 2 to the lane represented by cell 1 (see Figure 5.5).

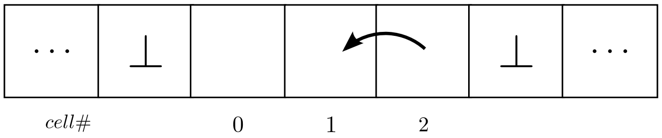


Figure 5.5: The third transfer. -

From the computational point of view a CA (CCA) model is more convenient since it can be easily parallelized, however, our multi-lane CCA model is essentially sequential. Indeed, we have to apply $2M$-times the global transition function Δ^* to perform all the transfer process. Therefore, our CCA model for multi-lane road can take advantage only if we decide not to apply the precedence rules or if we change the model to an asynchronous cellular automata model which is probably more appropriate to simulate a concurrent system like two vehicles requesting to move to the same position of the road. Besides, the number of lanes M are usually limited, thus in such kind of model we could gain just a factor M in the simulation speed.

Chapter 6

Simulation and Results

In this chapter, we implement our model and we run some simulations to study the general behavior of the model. Note that in our model we simulate just the vehicles and not the physical environment, i.e., we consider just the number of cells that are the number of vehicles, not the length of the road. This gives us the advantage of having a real-time simulation which does not depend on the length of the road.

6.1 The Simulator `ozsim`

In this section, we give a brief overview of the simulator `ozsim` that we have implemented in the code presented in Appendix A. The model we have presented is a CA model which is intrinsically parallel. Therefore, it can be implemented using for instance CUDA to take advantage of the power of the modern graphic cards to perform parallel computation. In the single-lane model it would be easy to parallelize the algorithm simply by giving to each thread of the GPU a cell representing in our case a vehicle. However, we decided to implement our model using the programming language Python since it is a high level language making the implementation of the model faster and easier. Indeed, during the phase of the development of writing the programme, we have used an object-oriented philosophy, especially while passing from the single-lane to multi-lane case, so that it has been easier to modify and rewriting some parts of the programme to tune it better. The code in Appendix A consists of four main classes:

- `vehicles()`: This class represents the kind of a vehicle introduced in Section 4.2. It contains all the properties such as; v_{max}, length, all the membership functions for the fuzzy modules, etc., and other properties used by the programme which are the name and the color of the kind of vehicle used by the visualization function `Real_Time_Visualizator`.

- `cars()`: This class essentially represents the set of state Σ of the single-lane CA $\mathcal{SL}$, see Section 4.2. Moreover, there are some other properties (data fields) such as a `timer` for each vehicle, used to keep track of the time passed. One important method defined in this class is the function `evalFeelings` which is the equivalent of the function $Eval_{(\mathcal{L},\mathcal{R})}(k_i, v_i(t), s_i(t))$, described in Section 5.1 to take the decision of which lane the vehicle wants to move.

- `external(cars)`: This class is a subclass of `cars()`. It is used to model the objects which are not vehicles, but they usually interact with them, e.g., obstacles, on- and off-ramps, etc. Some data fields of this class are;

    - `visibility`: is a Boolean parameter to specify whether or not an external object is visible to the other vehicles of the class `cars()`. This parameter can be used to make the vehicles slow down in the presence of an off-ramp or near an off-toll plaza.
    - `emissionRate`: is the parameter λ of a Poisson distribution which is used to model the emission phenomena. This parameter is used to model on-ramps and on-toll plaza.
    - `kindDistribution`: gives the probability distribution of the vehicle kinds used by the emitter to decide which kind of vehicle it will produce. For instance, if the `kindDistribution` is set as 10% of long vehicles and 90% of passenger vehicles then with 0.1 probability the emitter produces a long vehicle and otherwise it produces a passenger vehicle.
    - `initialVelocity`: the initial velocity that the vehicles are created with, by default we set this value as 12 m/s.
    - `absorptionProb`: gives the probability that a vehicle is absorbed by an off-ramp. Depending on how much the exit is used this probability is set up, i.e., the value is depending on the frequency of the usage of that exit. When this probability is 1, it corresponds to a sink (off-toll plaza), where all the vehicles are absorbed.

- `influenceRadius`: is the radius within which a vehicle is absorbed by an off-ramp. This parameter can be set up to change for instance the time used to process a vehicle by an off-toll plaza (the time required to slow down and make the payment). The smaller this parameter is, the more the time it takes for a vehicle to get absorbed within `influenceRadius`. `influenceRadius` accepts also negative parameters corresponding to the case where the vehicles do not see the off-ramp. This case is used to simulate electronic toll payments (open road tolling) in the off-toll plazas.
- `counter`: is used in many situations; the most common one is to count the number of cars to simulate loop detectors, the others are for ancillary usages.
- `buffer` and `bufferCapacity`: Buffer is a list that stores the vehicles absorbed by an off-ramp. This storing is used to analyze the information contained in each vehicle, for instance to check its timer which shows how much it took to reach to an off-ramp. Buffer capacity is simply the capacity of the buffer, and above this value the buffer is not able to store any more vehicles, so the off-ramp cannot absorb anymore. It is also used in the process of generating cars. If the vehicle can not be inserted into the road, it is stored in the buffer waiting for the road to be more empty. This can mimic the situation of entering a highway where there is traffic congestion and the on-ramp can store only a limited number of vehicles that want to enter.
- `sampRate`: is the number of cycles showing how much frequently the buffer is refreshed, i.e, every `sampRate` cycles the buffer is emptied so that it can analyze the information contained in the next pocket of vehicles arrived. The information to be analyzed are `throughput` and `avLatency`.
- `throughput`: is the number n of vehicles stored in buffer. For instance, it represents the number of vehicles processes every `sampRate` cycles in an off-toll plaza.
- `avLatency`: is the average of the information contained in the timer (latency) of the vehicles stored in buffer. If $t_1, \ldots, t_n$ are the values of the timers of the n vehicles, then

$$avLatency = \frac{t_1 + \ldots + t_n}{n}$$

- `lane()`: is the class characterizing a lane. It contains the configurations of a lane which is a list of objects `cars()`, two pointers `left` and `right` showing the left and right lanes, and a series of methods which are used to simulate the multi-lane CCA $\mathcal{ML}$ described in Chapter 5. In the program we essentially implement Algorithm 4 where the transfer of a vehicle is made step by step and not at once using the coping and erasing operators defined in Section 5.2. Furthermore, the transfer of the vehicles is not parallel but sequential from the leftmost to the rightmost lane. This sequentiality gives the precedence to the vehicles on the left-most lane and this precedence decreases towards the right-most lane. The updating is made using the following methods:
    - `_index`: finds the index where a vehicle should be inserted in the lane w.r.t its position considering also the length of it and its adjacent neighbors. This function is similar to $Indx(\sigma, c)$ described in Section 5.2 with the difference that `_index` check also the conditions in 5.2.
    - `_possibleCar`: calls `_index` to see whether the conditions in 5.2 are satisfied or not. If they are satisfied, it checks also the constraints in 5.3 and 5.4. If these are also satisfied, it returns the value given by `_index`.
    - `transfer`: transfers a vehicle from one lane to an adjacent lane, i.e., first it copies the vehicles that desire to move to a an adjacent lane and then it erases the original one. However before the transferring procedure, it calls `_possibleCar` to check if it is possible to make this transfer.
    - `evalChanges`: scans a lane and checks if a vehicle wants to move to an adjacent lane. In the affirmative case, it calls `transfer()` to perform this operation. In the case the element of the lane is an element of `external(cars)`, `evalChanges` calls the method `evalExternal`.
    - `evalExternal`: updates an external object of the class `external(cars)` in a lane. For instance, if this object is an emitter it performs all the operations such as the generation of the vehicles and the placement inside the lane. If it is a sink, it has to perform operations like counting the number of vehicles absorbed or the evaluation of the average latency.
    - `eval`: is the function that updates a lane. First it calls `evalChanges` to perform the transfer of the vehicles and after it updates the lane using the

global transition function of the single-lane CA model which is performed in the program by the function `transition_function`.

Moreover there are other methods which are used to create some external objects on a lane, such as on- and off-ramps (`createOnRamp` and `createOffRamp`), obstacles (`createObstacle`) and loop detectors (`createLoopDetector`).

Regarding the other functions available in `ozsim` we give an overview of the utilities of some of them:

- `updating_function`, `transition_function` are respectively the local transition function and its global version of the single-lane CCA model, i.e., they compute δ, δ^* described in Section 4.2.
- `initial_lane` is used to initialize a lane with some initial physical configuration (see Section 4.2).
- `createStreet` first creates the most-right lane with a given configuration, then it creates the other lanes with empty configurations linking them mutually using the `left` and `right` pointers each time we create a lane.
- `updateStreet` is the function that scans each lane from left to right and it updates it using `eval`.
- `lin_preimage`, `lin`, `fuzzy_agent` are the functions used to simulate the fuzzy decision modules (see Section 4.3).
- `draw_car`, `visual_position`, `Real_Time_Visualizator` are functions devoted to have a visualization tool for the real-time simulation (it requires the package Glumpy). During the running of the real-time simulation by clicking the left button of the mouse it is possible to call the function `randObstacle` that generates an obstacle randomly on some lane in front of the leading vehicle or the function `slowing_perturbation` that makes all the leading vehicles of each lane slow down with a factor 1/5. The latter function is useful in observing the back propagation wave formed during a jam situation (see Figure 6.18).

- `random_kind_array`, `random_position_array`, `random_velocity_array` are functions that generate a random array of kind of vehicles, positions and velocities respectively. The array of kind of vehicles is generated uniformly depending on an input array which gives the probability distribution. The array of position is created uniformly generating the spatial distance between each two vehicles (a uniform random number between a minimum distance and a maximum distance). Analogously the array of velocity is created by generating uniformly distributed numbers between a minimal velocity and a maximum velocity.
- `averageStreetVelocity`, `averageStreetDistance` calculate the average of the velocities of the vehicles and the average of the spatial distances between adjacent vehicles (from front bumper to rear bumper), respectively.
- `createRandHighway` creates a highway with some random parameters. It calls the functions `createOnToll` and `createOffToll` to generate on- and off-toll plazas, respectively, at the beginning and the end of this highway. Moreover, some on- and off-ramps or some obstacles can be added randomly.

A screenshot of the real-time simulator can be seen in Figure 6.1 where we have used `createRandHighway` to generate a random four-lane highway of length 22000 m with an on-toll plaza, an off-toll plaza, two on-toll ramps, two off-toll ramps and three obstacles. The simulation runs at a velocity of around 3 FPS with less than 1000 vehicles and around 1.5 frames per seconds with more than 2500 vehicles on a laptop equipped with a core $i5$ running on Windows 7 professional.

6.2 Setting the Kinds of Vehicles

In this section, we describe the kinds of vehicles we used for our experiments. In all our experiments, we use just two kinds of vehicles which we call as *passenger vehicles* and *long vehicles*. The vehicles have the following parameters (see the code line 1393 and 1427 in Appendix A):

- *passenger vehicles*: the maximum velocity is 36 m/s, the optimal velocity: 28 m/s, the length: 4 m, the natural acceleration noise: 0.2 m/s^2 (see [23]), the

Figure 6.1: A screenshot of the real-time simulator. -

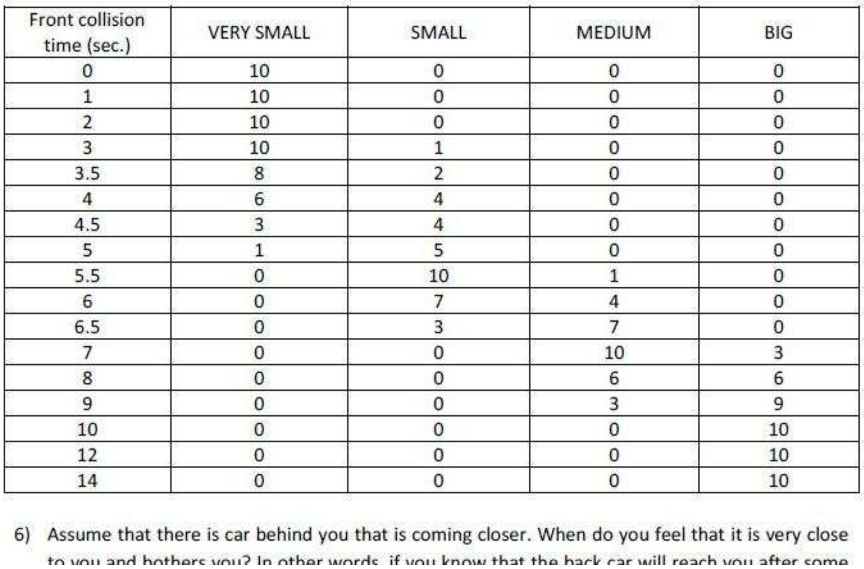
maximum stress: 500 m, the minimum stress: -450 m, the function of the probability of lane-changing to the right lane $P_R(x) = x$, the function of the probability of lane-changing to the left lane $P_L(x) = x$.

- *long vehicles*: the maximum velocity is 25 m/s, the optimal velocity: 20 m/s, the length: 9 m, the natural acceleration noise: 0.1 m/s^2 (see [23]), the maximum stress: 300 m, the minimum stress: -700 m, the function of the probability of lane-changing to the right lane $P_R(x) = x$, the function of the probability of lane-changing to the left lane $P_L(x) = x^{1.25}$.

Regarding the fuzzy membership functions, we have tuned them according to a questionary posed to a group of drivers. In each question, it is requested to give an integer value between $0, \ldots, 10$ which represents how much for the interviewee a claim related to their perception while driving. For instance, one of the question with regard to the front collision time is,

> Assume that there is a car in front of you (stopping or moving), and independently from the front distance and relative velocity, you know that some seconds later you will collide to that car, which is called as front collision time. For you, how many seconds of front collision time will be "very small"

amount of time? ”small” amount of time? ”medium” amount of time? and ”big” amount of time? Fill the table below with a rating between 0-10, that will give the ”meaning of front time to collision” is ”very small”, ”small”, ”medium” and ”big”.



Front collision time (sec.)	VERY SMALL	SMALL	MEDIUM	BIG
0	10	0	0	0
1	10	0	0	0
2	10	0	0	0
3	10	1	0	0
3.5	8	2	0	0
4	6	4	0	0
4.5	3	4	0	0
5	1	5	0	0
5.5	0	10	1	0
6	0	7	4	0
6.5	0	3	7	0
7	0	0	10	3
8	0	0	6	6
9	0	0	3	9
10	0	0	0	10
12	0	0	0	10
14	0	0	0	10

6) Assume that there is car behind you that is coming closer. When do you feel that it is very close to you and bothers you? In other words, if you know that the back car will reach you after some

Figure 6.2: A screenshot of the questionnaire. -

The answer of one of the interviewee can be seen in a screenshot of the questionary (see Figure 6.2). After we collect the data from 30 interviewees, we have taken the average, divided by 10 and normalized this data since the range of a membership function is in the interval $[0, 1]$. In this way, we consider the corresponding membership function by interpolating linearly these data. However, in the simulation, we do not use these functions since we have noticed a slowing down with respect to other simpler and more common membership functions like the triangular and trapezoidal ones. For this reason, we have approximated the membership functions obtained from the data with the triangular and trapezoidal membership functions (see Figures 6.3, 6.4 and 6.5). We do not think that this kind of approximation can influence too much the results of the simulation. Firstly, because the sample is too small and so the data obtained is subject to errors and secondly, the obtained membership functions are not so different from their approximated versions and the simulation can gain a boost of a factor 1.5.

The problem of tuning the membership functions is an interesting challenge which deserves a deeper analysis, however, in the simulation, we are interested in a first testing of this model and its general behavior.

Note that for long vehicles the effect of back collision time and back distance is not taken into consideration so much since long vehicles usually do not perceive the pushing effect. Therefore, in the questionary it is neglected this part and we made the assumptions for these membership functions as they are seen in Figures 6.3 and 6.4. In regard to the zero acceleration, it is not evaluated through the questionary, so we consider a triangular membership function representing the zero acceleration for both passenger and long vehicles as it is seen in Figure 6.5.

6.3 The Experiment Scenarios

In this section, we describe the scenarios of the simulations that we have run. The function `__main__` (line 1529 of the code in Appendix A) simulates a piece of highway of length L with M number of lanes, an on-toll plaza and an off-toll plaza. The ordered parameters passed to `__main__` are the followings:

- Length of the road L.
- Number of lane M.
- Number of iterations (the simulation time).
- Number of repetitions of the same experiment.
- Emission rate λ: the number of vehicles entering this piece of highway.
- Influence radius ρ of the off-toll plaza, recall that $\rho = -1$ means that there is an open road tolling.
- Obstacle: if this parameter is -1 an obstacle is placed on the left-most lane, if it is set to $+1$ then an obstacle is placed on the right-most lane, and if this parameter is missing, there is no obstacle.

For instance, if one wants to simulate an experiment with $L = 25000\ m$, $M = 3$, 10000 iterations, 1 repetition, $\lambda = 1\ veh/sec$, $\rho = -1$, an obstacle on the right-most lane (+1), then it should be lunched `python ozsim.py 25000 3 10000 1 1 -1 +1`.

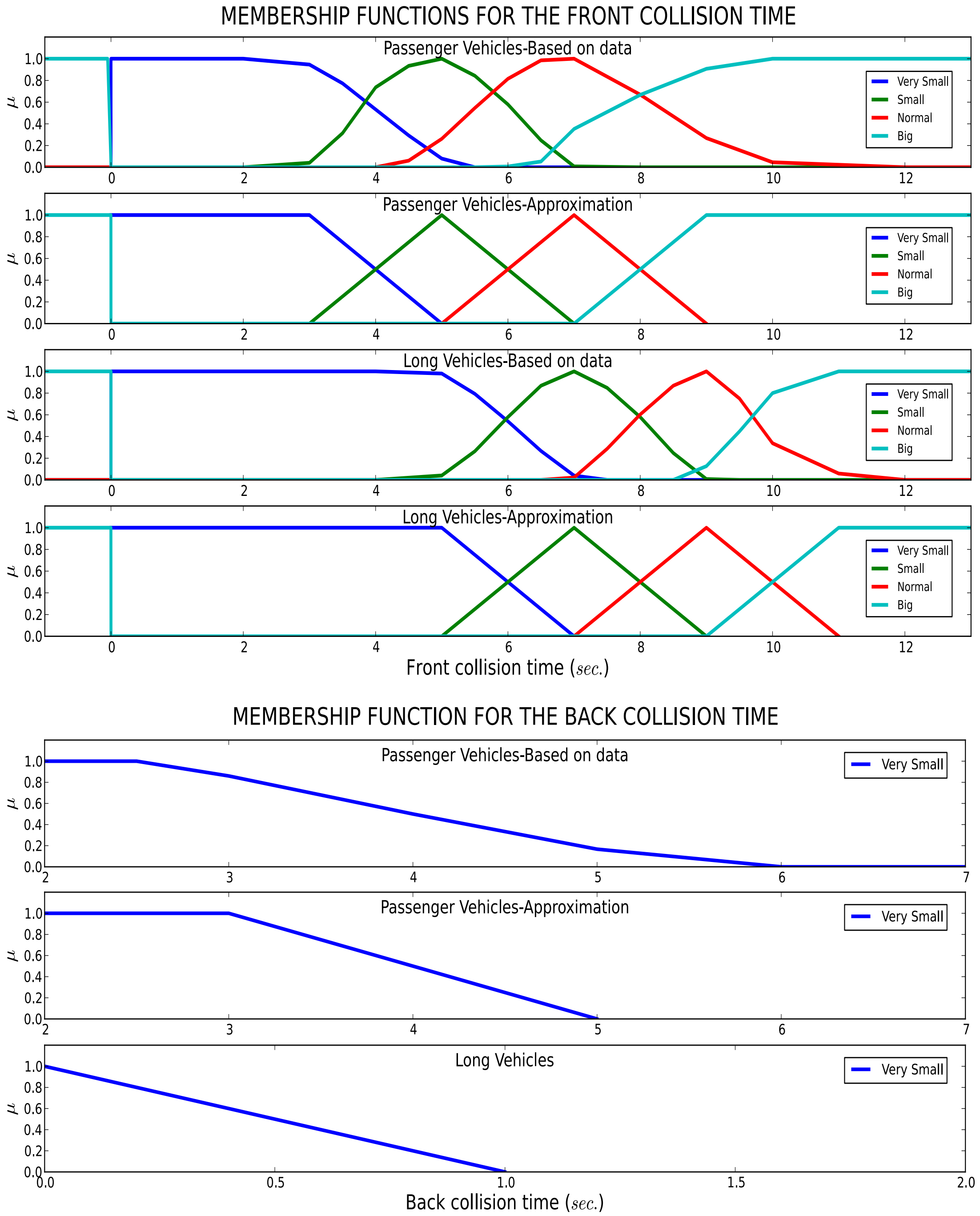


Figure 6.3: Fuzzy membership functions for Front Collision Time and Back Collision Time - Passenger vehicles and long vehicles with their approximated versions.

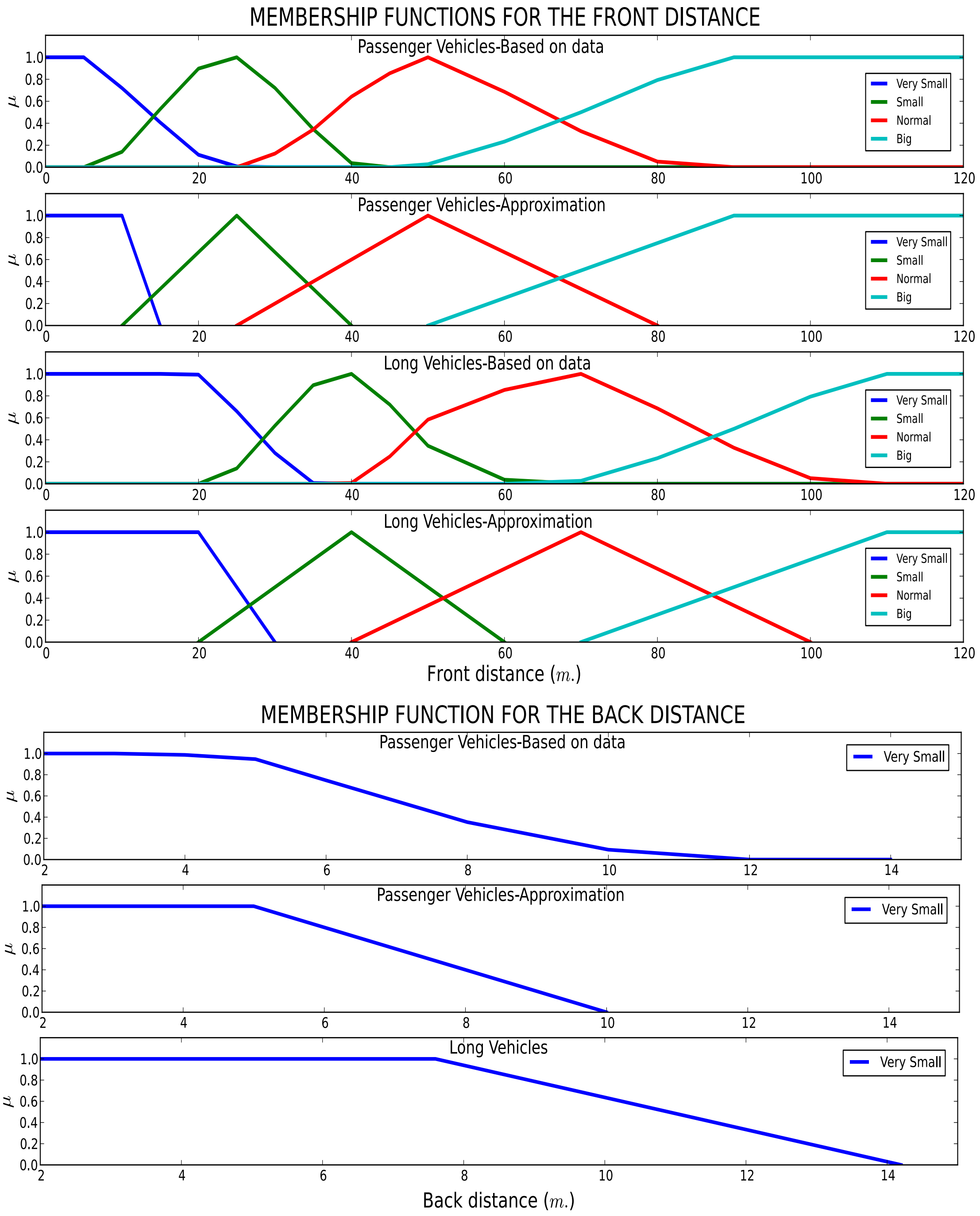


Figure 6.4: Fuzzy membership functions for Front Distance and Back Distance - Passenger vehicles and long vehicles with their approximated versions.

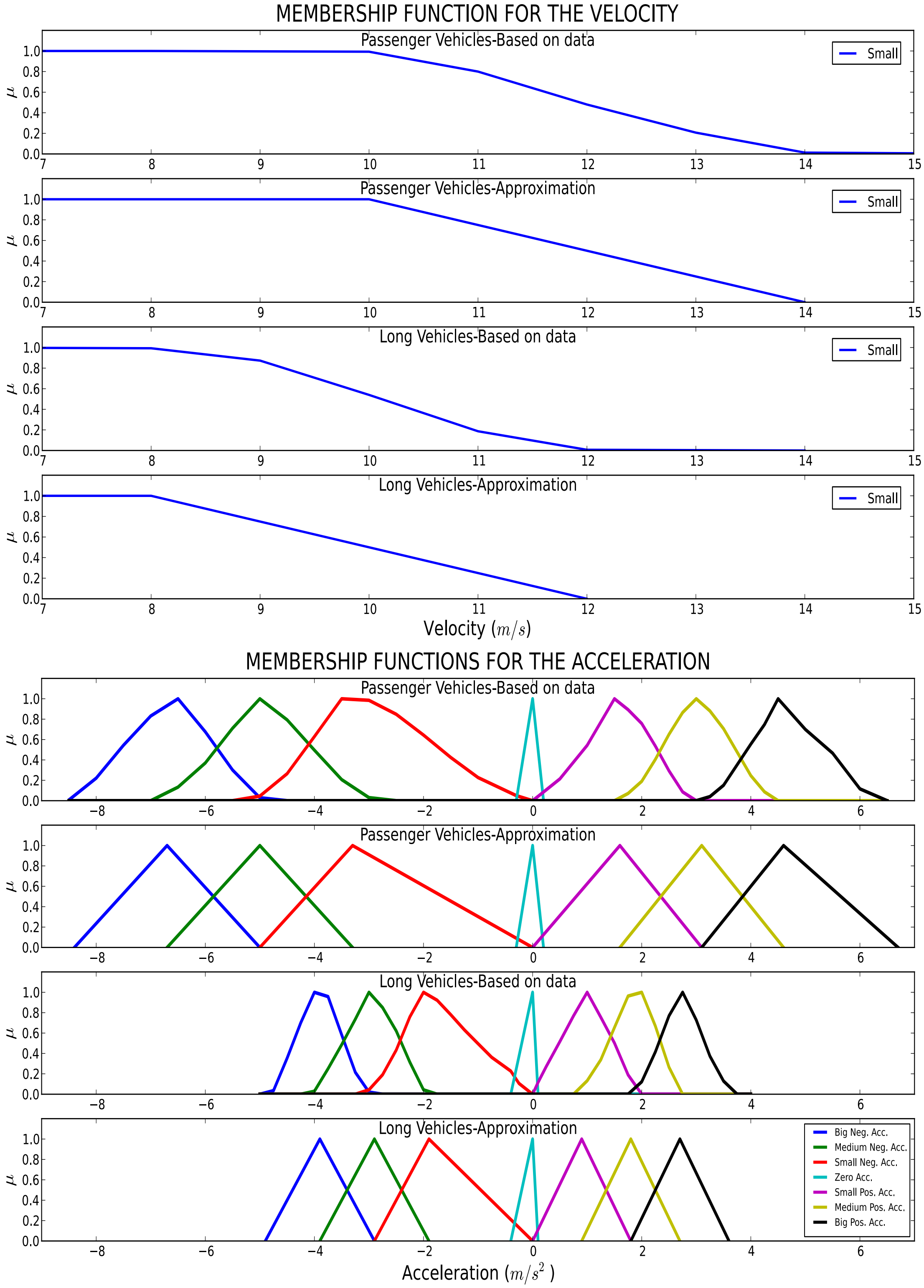


Figure 6.5: Fuzzy membership functions for Velocity and Acceleration - Passenger vehicles and long vehicles with their approximated versions.

The kinds of vehicles used are the passenger and the long vehicles whose parameters are described in Section 6.2. The percentages of long vehicles considered in the simulation are respectively of $0\%, 10\%, 20\%, 30\%$. We do not consider higher values since usually the percentage of long vehicles on a highway does not exceed the 40%. Therefore in the creation of the on-toll plaza we have used the function `createOnToll(leftMostLane, initialPosition, emissRate, kindDistribution)` defined in the code of Appendix A where `emissRate` is the average number of vehicles created every second in a single-lane and `kindDistribution` is the probability distribution of the two kinds of vehicles. The value `emissRate` is set to λ/M where we have considered $\lambda = 0.25, 0.5, 1, 1.5, 2$ vehicles per seconds. For instance, in the case $\lambda = 1.5$ vehicles per second with three lanes, the average number of vehicles entering the piece of highway is 90 vehicles per minute which means that each lane near the on-toll plaza is charged of around 30 vehicles per minute which corresponds to a situation of heavy traffic since the maximum capacity of a lane is considered around 40 vehicles per minute (see [34]). Regarding the off-toll plaza we have used the function `createOffToll(leftMostLane, position, absorptionProb, influenceRadius, bufferCapacity, sampRate)` where position is set to L, the absorption probability `absorptionProb` is set to 1 since all the vehicles exit from the road, the influence radius `influenceRadius` considered are $\rho = 10, 25, 50\ m$ or, -1 which is corresponding to the case where the off-toll plaza is not visible to the vehicles thus they pass through it without stopping. This situation corresponds to the case of an open road tolling payment system where the vehicles do not need to slow down and stop to make the payment. The last two parameters `bufferCapacity` and `sampRate` are set to 100 vehicles and 10 seconds, respectively. In other words, each lane in the off-toll plaza can store a maximum of 100 vehicles and every 10 seconds the simulator (`evalExternal`) checks the number of vehicles stored and it calculates the average latency, i.e., the average of the time spent to travel from the entrance to the exit, and finally it resets the buffer. If the last parameter passed in `__main__` is $1, -1$, an obstacle is placed on the right-most, left-most lane, respectively. This situation is made to analyze a bottleneck phenomenon. We have used the method `createObstacle(self, position, dimension, color = 0.45)` with parameters `position`$= L/2$, `dimension`$= L/5$ to place an obstacle of dimension $(2L)/5$ in the middle of the piece of highway. In Tables 6.1 and 6.2, the two scenarios of the experiments we have performed can be seen.

Table 6.1: Scenario №1

Experiment №	Road Length (m)	Lanes	Iterations (sec)	Repetitions	Emission Rate (veh/sec)	Influence Radius (m)	Obstacle
1	25000	2	10000	1	0.25	25	-
2	25000	2	10000	1	0.25	25	*R*
3	25000	2	10000	1	0.25	25	*L*
4	25000	2	10000	1	0.25	◇	-
5	25000	2	10000	1	0.25	◇	*R*
6	25000	2	10000	1	0.25	◇	*L*
7	25000	2	10000	1	1.0	25	-
8	25000	2	10000	1	1.0	25	*R*
9	25000	2	10000	1	1.0	25	*L*
10	25000	2	10000	1	1.0	◇	-
11	25000	2	10000	1	1.0	◇	*R*
12	25000	2	10000	1	1.0	◇	*L*
13	25000	3	10000	1	0.5	25	-
14	25000	3	10000	1	0.5	25	*R*
15	25000	3	10000	1	0.5	25	*L*
16	25000	3	10000	1	0.5	◇	-
17	25000	3	10000	1	0.5	◇	*R*
18	25000	3	10000	1	0.5	◇	*L*
19	25000	3	10000	1	1.0	25	-
20	25000	3	10000	1	1.0	25	*R*
21	25000	3	10000	1	1.0	25	*L*
22	25000	3	10000	1	1.0	◇	-
23	25000	3	10000	1	1.0	◇	*R*
24	25000	3	10000	1	1.0	◇	*L*
25	25000	3	20000	1	1.5	25	-
26	25000	3	20000	1	1.5	25	*R*
27	25000	3	20000	1	1.5	25	*L*
28	25000	3	20000	1	1.5	◇	-
29	25000	3	20000	1	1.5	◇	*R*
30	25000	3	20000	1	1.5	◇	*L*
31	25000	4	10000	1	1.0	25	-
32	25000	4	10000	1	1.0	25	*R*
33	25000	4	10000	1	1.0	25	*L*
34	25000	4	10000	1	1.0	◇	-
35	25000	4	10000	1	1.0	◇	*R*
36	25000	4	10000	1	1.0	◇	*L*
37	25000	4	10000	1	1.5	25	-
38	25000	4	10000	1	1.5	25	*R*
39	25000	4	10000	1	1.5	25	*L*
40	25000	4	10000	1	1.5	◇	-
41	25000	4	10000	1	1.5	◇	*R*
42	25000	4	10000	1	1.5	◇	*L*

◇: open road tolling, R: obstacle on the right-most lane, L: obstacle on the left-most lane.

Table 6.2: Scenario №2

Experiment №	Road Length (km)	Lanes	Iterations (sec)	Repetitions	Emission Rate (veh/sec)	Influence Radius (m)	Obstacle
1	5000	2	1000	100	1.0	10	-
2	5000	2	1000	100	1.0	10	*R*
3	5000	2	1000	100	1.0	25	-
4	5000	2	1000	100	1.0	25	*R*
5	5000	2	1000	100	1.0	25	*L*
6	5000	2	1000	100	1.0	50	-
7	5000	2	1000	100	1.0	50	*R*
8	5000	2	1000	1000	1.0	50	-
9	5000	2	1000	1000	1.0	50	*R*
10	5000	2	1000	100	1.0	⋄	-
11	5000	2	1000	100	1.0	⋄	*R*
12	5000	2	1000	100	1.0	⋄	*L*
13	5000	3	1000	100	0.5	25	-
14	5000	3	1000	100	0.5	25	*R*
15	5000	3	1000	100	0.5	25	*L*
16	5000	3	1000	100	0.5	⋄	-
17	5000	3	1000	100	0.5	⋄	*R*
18	5000	3	1000	100	0.5	⋄	*L*
19	5000	3	1000	100	1.0	25	-
20	5000	3	1000	100	1.0	25	*R*
21	5000	3	1000	100	1.0	25	*L*
22	5000	3	1000	100	1.0	⋄	-
23	5000	3	1000	100	1.0	⋄	*R*
24	5000	3	1000	100	1.0	⋄	*L*
25	5000	3	1000	100	1.5	10	-
26	5000	3	1000	100	1.5	10	*R*
27	5000	3	1000	100	1.5	25	-
28	5000	3	1000	100	1.5	25	*R*
29	5000	3	1000	100	1.5	25	*L*
30	5000	3	1000	100	1.5	50	-
31	5000	3	1000	100	1.5	50	*R*
32	5000	3	1000	100	1.5	⋄	-
33	5000	3	1000	100	1.5	⋄	*R*
34	5000	3	1000	100	1.5	⋄	*L*
35	5000	3	1000	100	2	10	-
36	5000	3	1000	100	2	10	*R*
37	5000	3	1000	100	2	50	-
38	5000	3	1000	100	2	50	*R*
39	5000	3	1000	100	2	⋄	*R*
40	5000	4	1000	100	1.5	25	-
41	5000	4	1000	100	1.5	25	*R*
42	5000	4	1000	100	1.5	25	*L*
43	5000	4	1000	100	1.5	⋄	-
44	5000	4	1000	100	1.5	⋄	*R*
45	5000	4	1000	100	1.5	⋄	*L*

⋄: open road tolling, *R*: obstacle on the right-most lane, *L*: obstacle on the left-most lane.

6.4 Analysis of the Experimental Results

Each experiment in Scenario 1 with 10000 iterations takes around 40 minutes and each experiment in Scenario 2 with 1000 iterations and 100 repetitions takes around 6.6 hours using one core of a computer equipped with a 16 core Xeon at 2.7GHz ($X5550$) with 16GB of RAM running Debian Linux (kernel 2.6.32).

The macroscopic parameters considered for the purpose of analyzing the general behavior of traffic in our experiments, are evaluated as following:

The density describing the number of vehicles per unit length of the piece of highway (measured in vehicles per meter) at time t is,

$$k(t) = \frac{N(t)}{L}$$

where $N(t)$ is the total number of vehicles at time t and L is the length of the road representing a piece of highway (25 km in the experiments without repetition and 5 km in the repeated experiments).

The average velocity, i.e., the averaged sum of the velocities at time t is,

$$v_{av}(t) = \frac{\sum_{i=1}^{N(t)} v_i}{N(t)}$$

Finally, the flow is defined as the number of vehicles passing by a specific point of the piece of highway per unit of time (measured in vehicles per second) as,

$$q(t) = k(t) \cdot v_{av}(t)$$

The analysis of traffic flow is typically performed by constructing the fundamental diagram (the flow-density diagram) that determines the traffic state of a roadway by showing the relation between flow and density. The equations described above show how to compute the variables at a particular time and they are used to plot the flow-time and density-time graphics (see Figure 6.8) and the fundamental diagrams (see for instance, Figure 6.6 and Figure 6.9). Note that in our experiments, the density can reach at 0.25 veh/m per lane as maximum since the length of a passenger vehicle is assumed to be 4 m. On the other hand, in the NaSch-type models, the density values used to plot the fundamental diagram is evaluated in a different way. In the sense that in our model the number of vehicles is not constant since the vehicles are entering (depending on a Poisson stochastic process) and exiting the piece of highway where the exiting rate is different than the entering one. Instead, for instance in NaSch model, the number of vehicles cannot change during the simulations since the model is defined with closed boundary conditions, meaning that the system has a constant density which is quite unrealistic. Therefore, in order to obtain different densities, they run the simulations by changing the length of the road (the number of cells) in these models.

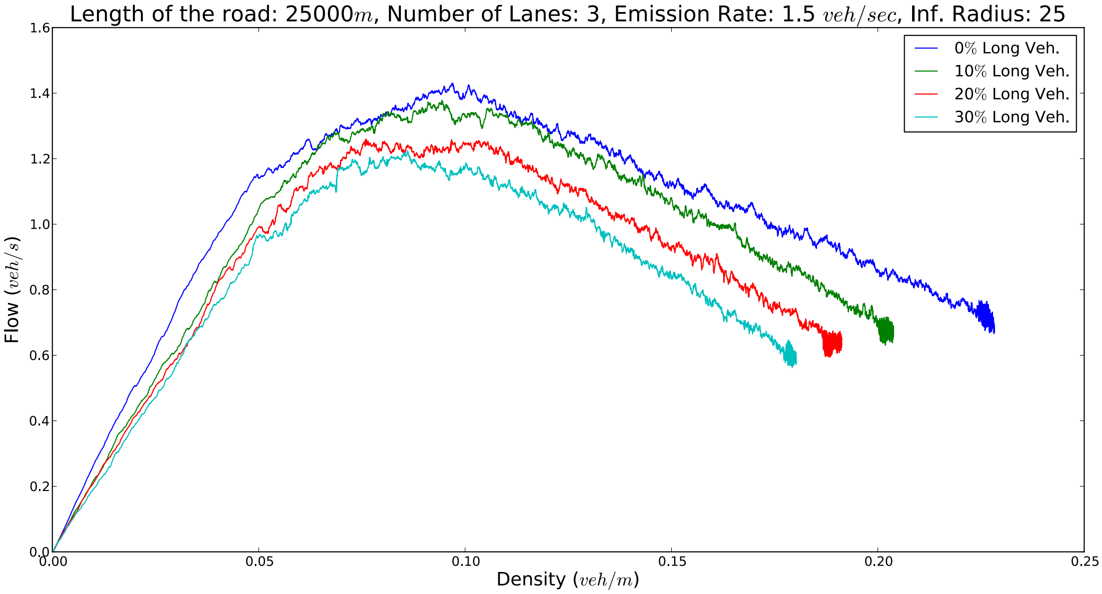


Figure 6.6: Fundamental diagram of flow with various percentages of long vehicles - This figure corresponds to Experiment 25 of Scenario 1.

In all the experiments performed, we have noticed that the heterogeneity is an important factor in influencing the flow. In the fundamental diagram of the flow in Figure 6.6, for instance, it is clear that adding even a small amount of long vehicles changes the diagram significantly.

Different composition of vehicles in traffic stream formed by changing the percentage of long vehicles effects the throughput (see Figure 6.7), as it is predictable since long vehicles are slower and so in the queue near the off-toll plaza it takes more time to move and get processed. For instance, without long vehicles the throughput is 7.7 vehicles per 10 seconds which becomes 6.7 when the amount of long vehicles is 20%. The effect of heterogeneity is seen also on the flow, density, average velocity, average distance and latency as we observe in the first plot of Figure 6.8. This issue is not a consequence of

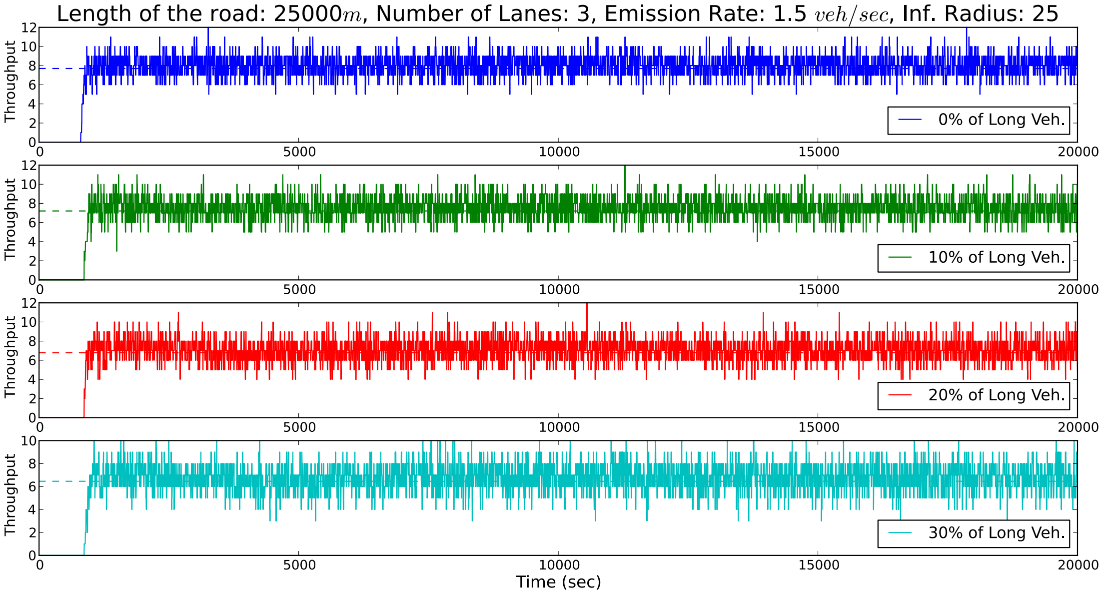


Figure 6.7: The throughputs according to the various percentages of long vehicles - This figure corresponds to Experiment 25 of Scenario 1.

the statistical fluctuation (see the second plot of Figure 6.8 with repetitions). Indeed, for instance Figure 6.9 shows the average of all the repeated experiments for 3 lanes and for 4 lanes in which it is seen that the flow-density relation has the dependency on the percentage of long vehicles.

The experiments show that the model is able to reproduce the typical traffic flow physical phenomena such as the three phases of traffic flow [30]: Free flow, synchronized flow and wide-moving jam, see Figure 6.10. Free flow corresponds to the region of low to medium density and weak interaction between vehicles. In general, the slope of the fundamental diagram in the free flow phase is related to the speed limit, meaning that in this phase vehicles can move almost at the speed limit. Instead, in the free flow phase of the fundamental diagram in our experimental results, the slope is related to the optimal velocity, meaning that in this phase the vehicles can move almost at the optimal velocity. The reason is, in our model the vehicles do not aim to reach to the maximum velocity, but they tend to go with their optimal velocity. Free flow is characterized with a strong correlation and quasi-linear relation between the local flow and the local density [45]. The synchronized flow presents medium and high density while the flow can behave free or jammed. In other words, it is defined by the interaction between the vehicles and

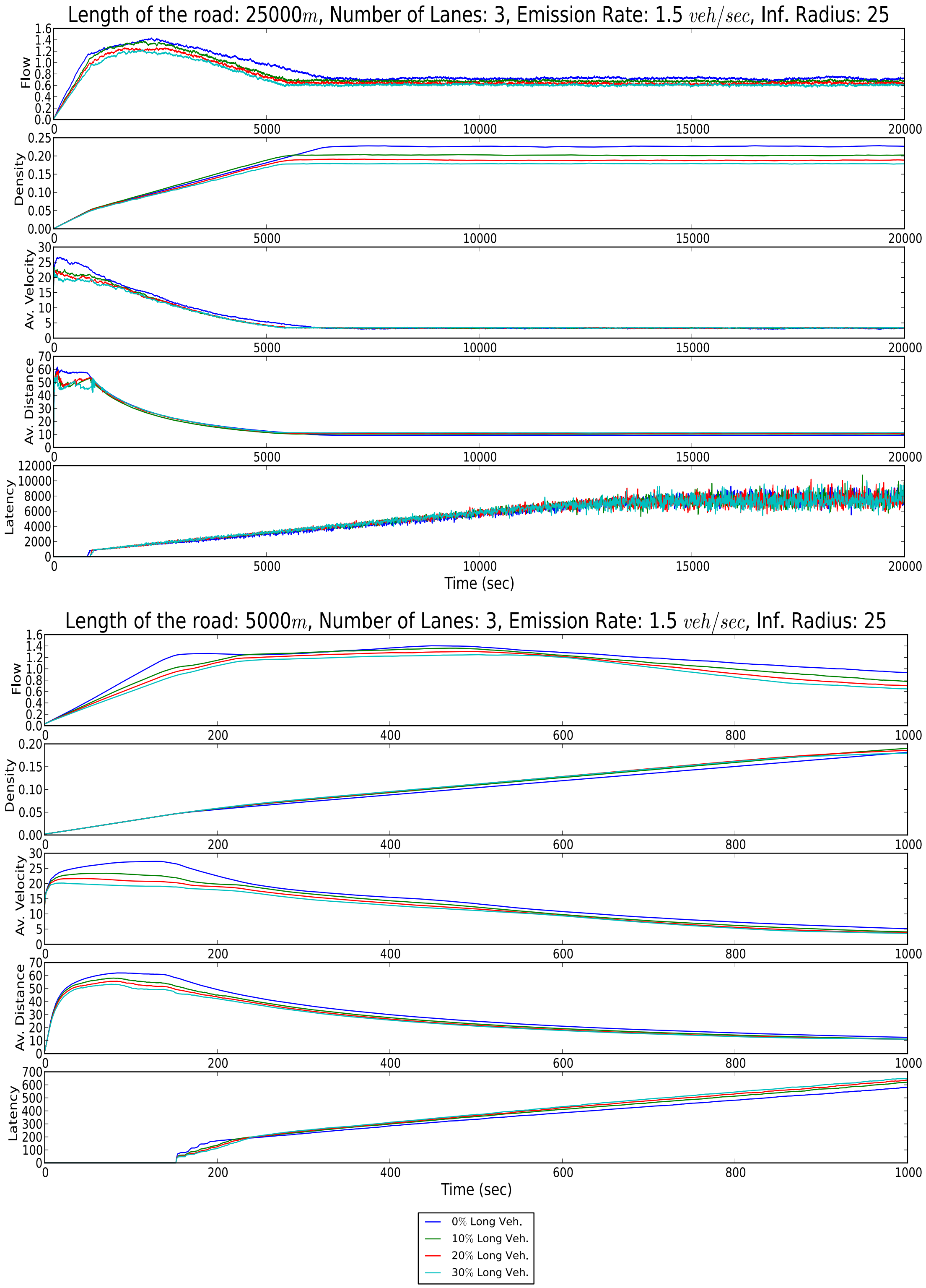
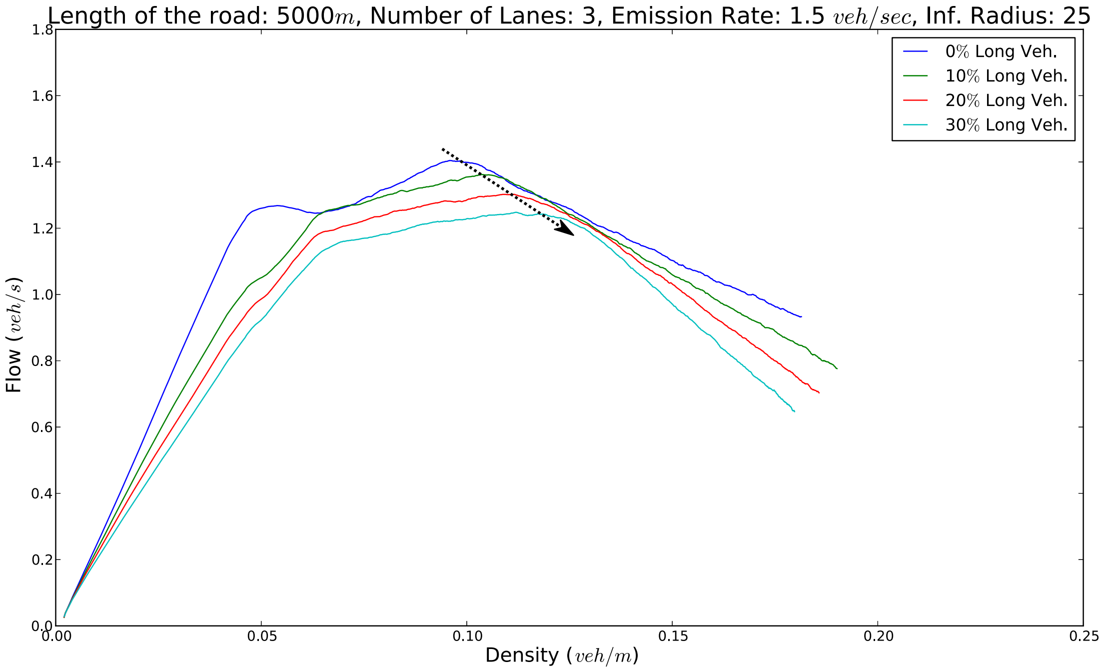


Figure 6.8: One of the typical diagrams of "plot.py" without and with repetitions, respectively, showing flow, density, average velocity, average distance and latency graphics with respect to time - This figure corresponds to Experiments 25 of Scenario 1 and 27 of Scenario 2, respectively.



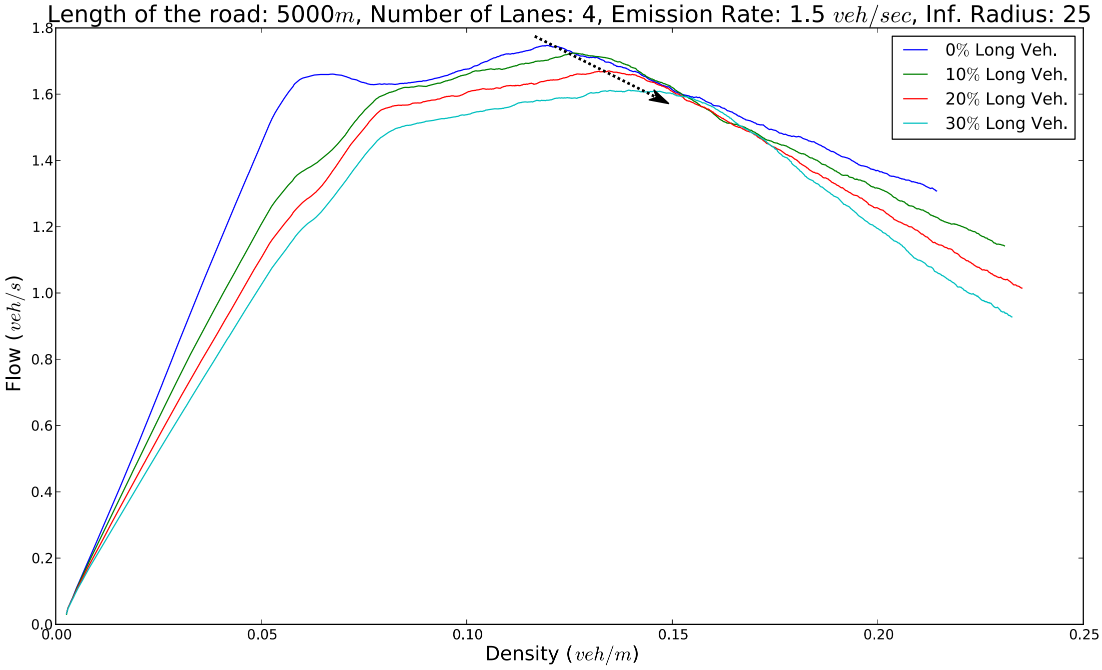


Figure 6.9: Fundamental diagrams with 100 repetitions - These figures correspond to Experiments 27 and 40 of Scenario 2, respectively.

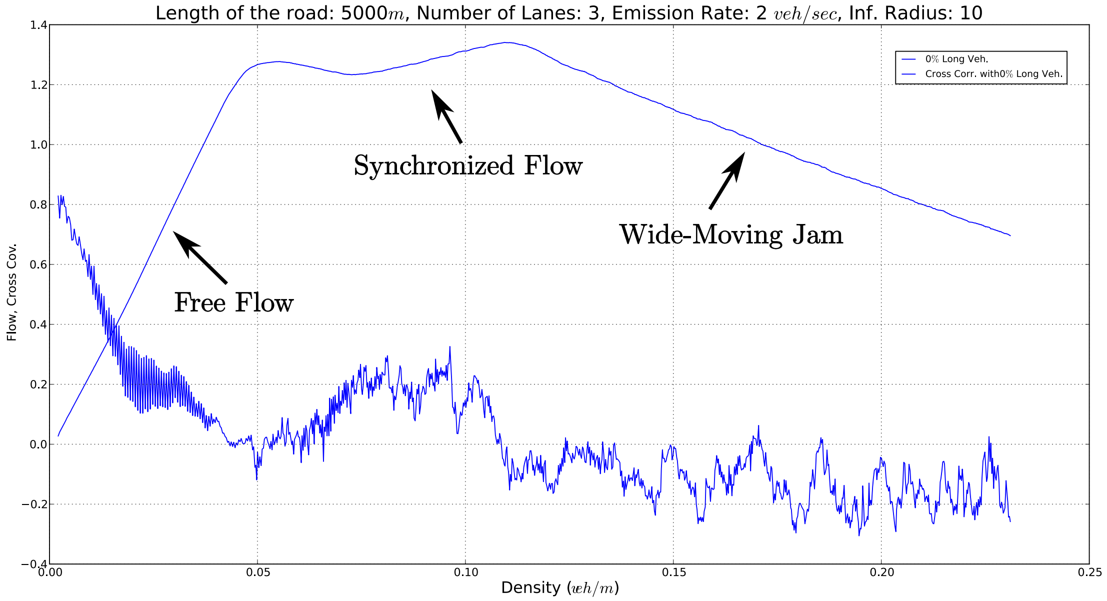


Figure 6.10: Traffic phases in the fundamental diagram: Free flow, synchronized flow and wide-moving jam, and the cross-covariance between the flow and density - This figure corresponds to Experiment 35 of Scenario 2.

is characterized by an uncorrelated flow-density diagram. However, this phase is not clearly understood in the context of CA and not observed in most of the NaSch-type CA models. It probably requires the presence of sources (on-ramps, on-toll plaza) and sinks (off-ramps, off-toll plaza), see [59]. The wide-moving jam phase represents the situation where the traffic is jammed (congested). In this phase, an increase in density results with a decrease in the flow. Let us now consider the cross-covariance between the flow $q(t)$ and the density $k(t)$, see Figure 6.10:

$$cc(q,k) = \frac{\langle q(t)k(t)\rangle - \langle q(t)\rangle\langle k(t)\rangle}{\sqrt{\langle q(t)^2\rangle - \langle q(t)\rangle^2}\sqrt{\langle k(t)^2\rangle - \langle k(t)\rangle^2}}$$

where the brackets $\langle\cdot\rangle$ indicate averaging the values obtained in all the experiments at time t. In the free flow phase, the flow is strongly related to the density indicating that the average velocity is nearly constant. For large densities, in the wide-moving jam phase, the flow is mainly controlled by density fluctuations. There is a transition between these two phases where the fundamental diagram shows a plateau when the cross-variance is close to zero. This situation with $cc(q,k) \approx 0$, is identified as synchronized flow [33, 45].

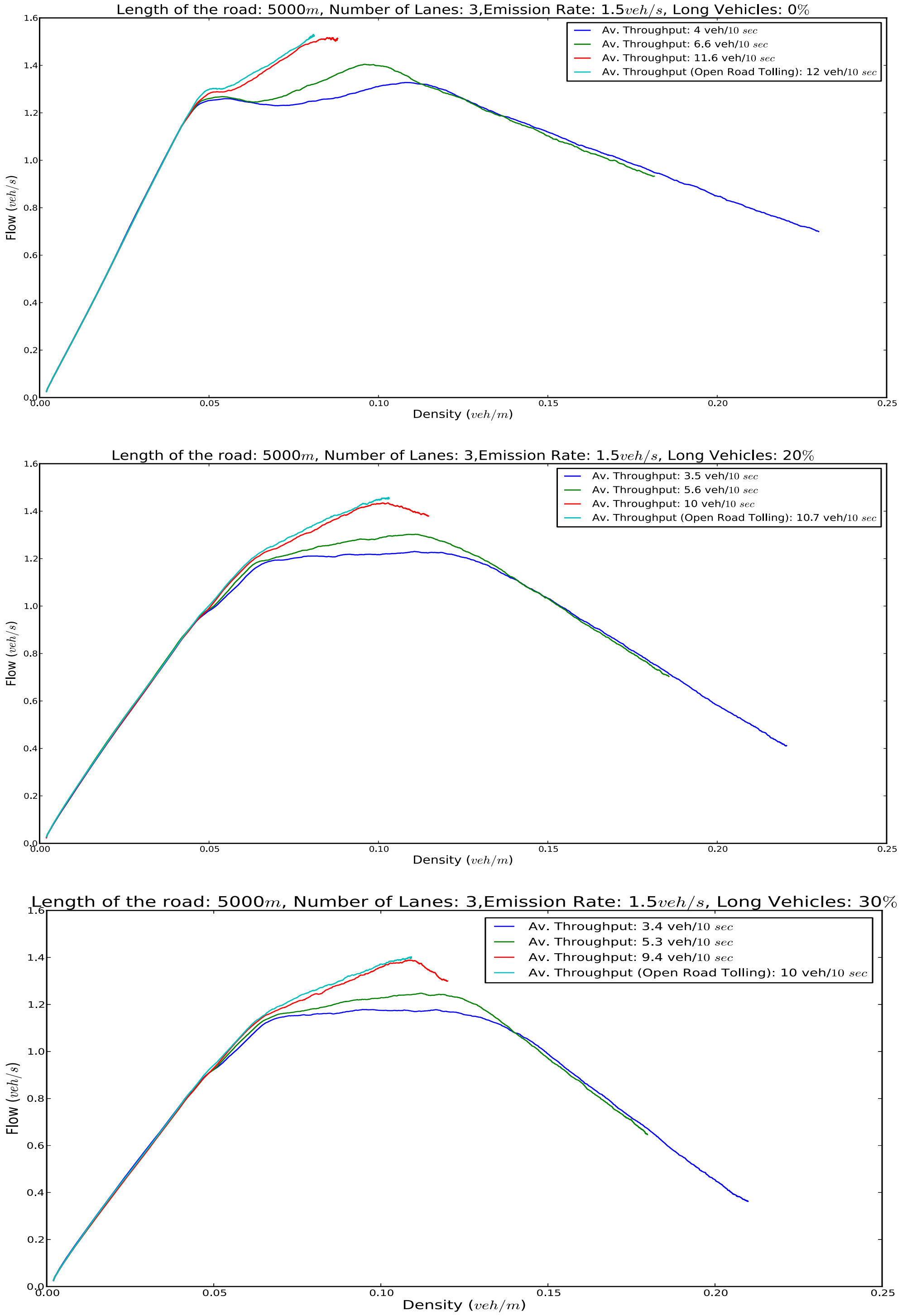


Figure 6.11: The fundamental diagrams depending on the different average throughputs - The figures are arranged according to the different percentages of long vehicles: 0%, 20% and 30%, respectively, and in each figure the plots from up to down correspond to Experiments 32, 30, 27 and 25 of Scenario 2, respectively.

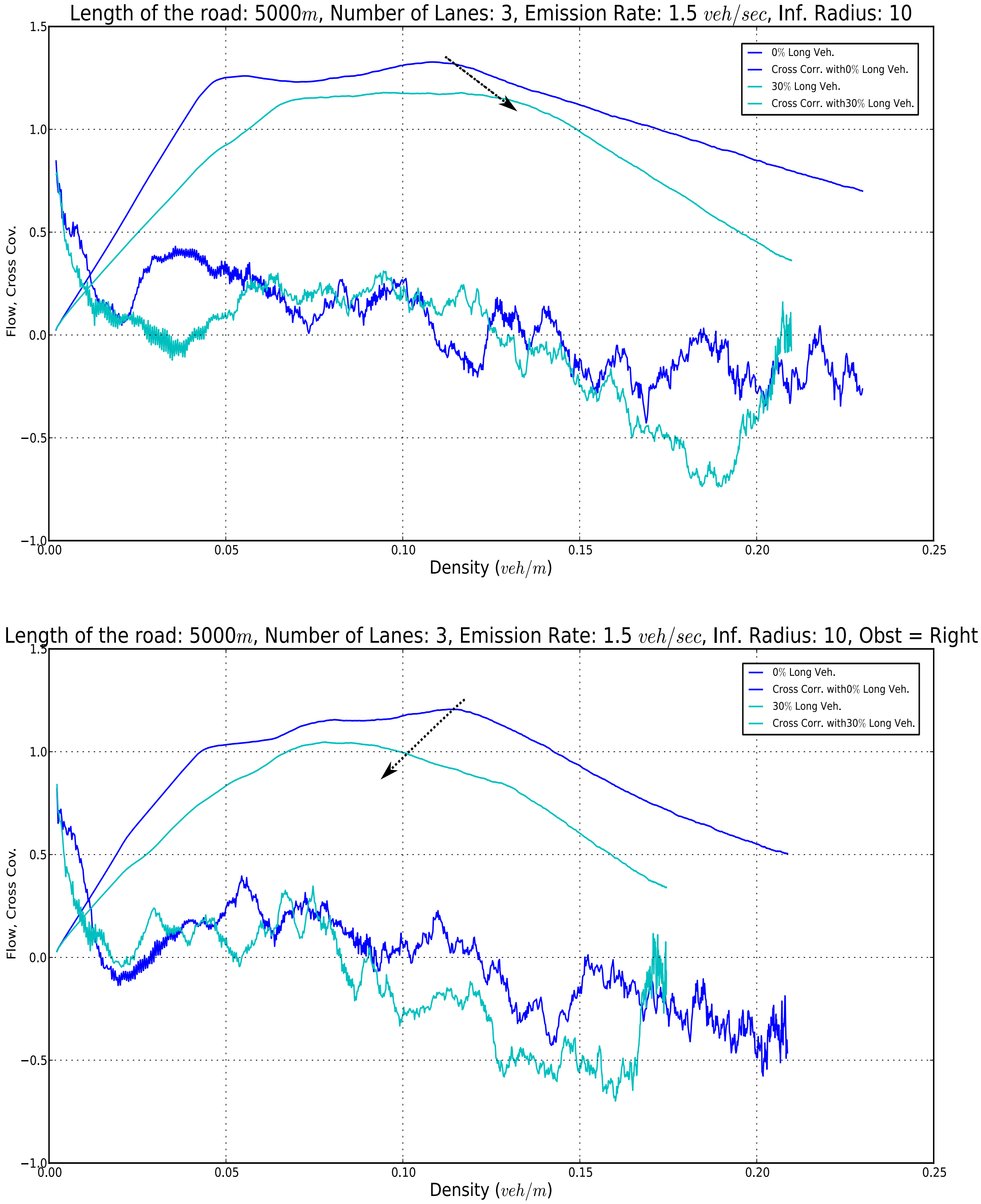


Figure 6.12: The fundamental and the cross-covariance diagrams without and with obstacle - These figures correspond to Experiments 25 and 26 of Scenario 2, respectively.

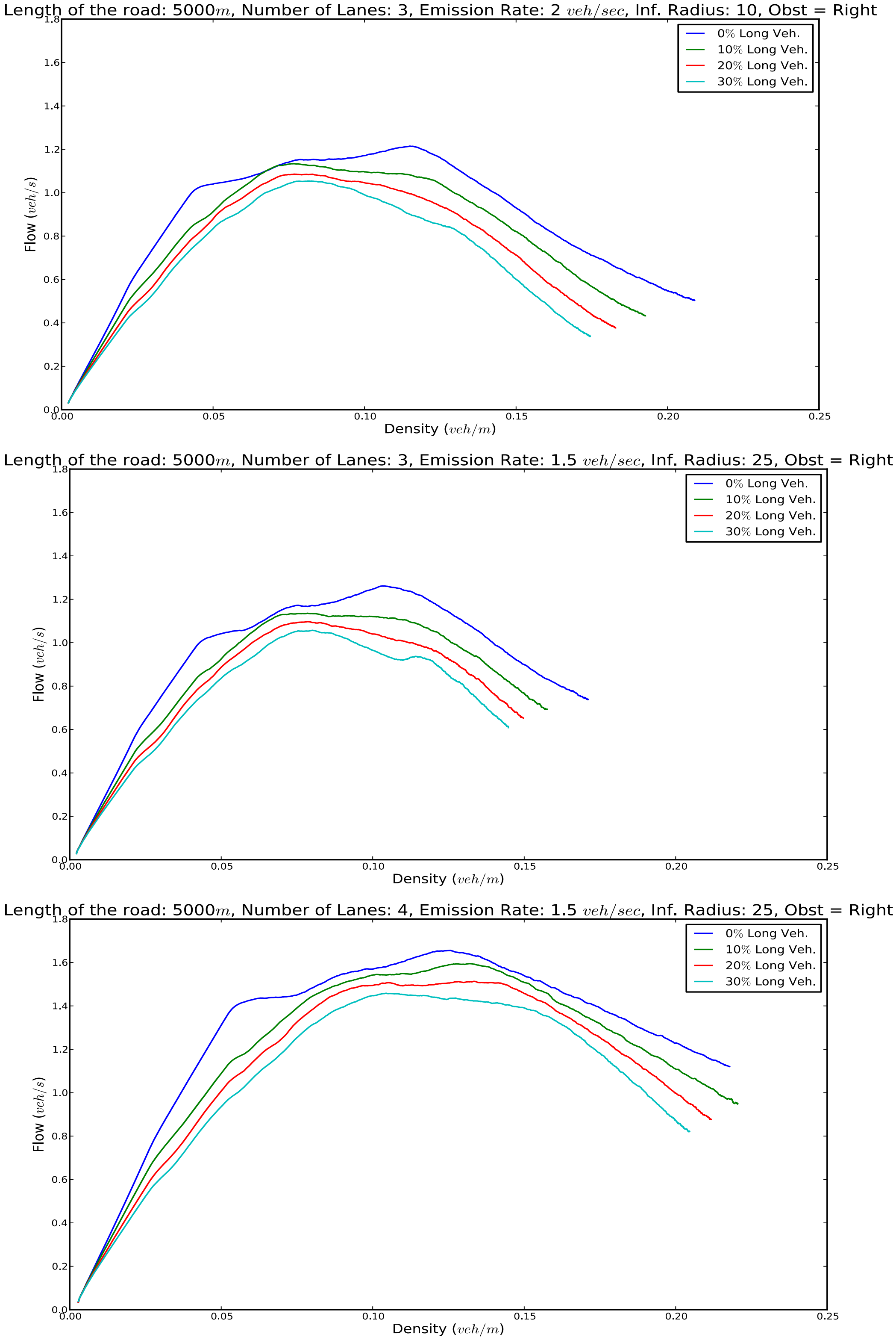


Figure 6.13: The slope in the wide-moving jam phase with the obstacles - These figures correspond to Experiments 36, 28 and 41 of Scenario 2, respectively.

As it is seen in Figure 6.10, we reproduce these relations where the free flow phase initiates with the situation of cross-covariance close to 1 and continues with a positive cross-covariance, the synchronized flow phase is in the region where the cross-covariance is close to zero and the wide-moving jam phase corresponds to the situation where the cross-covariance is negative (anticorrelation).

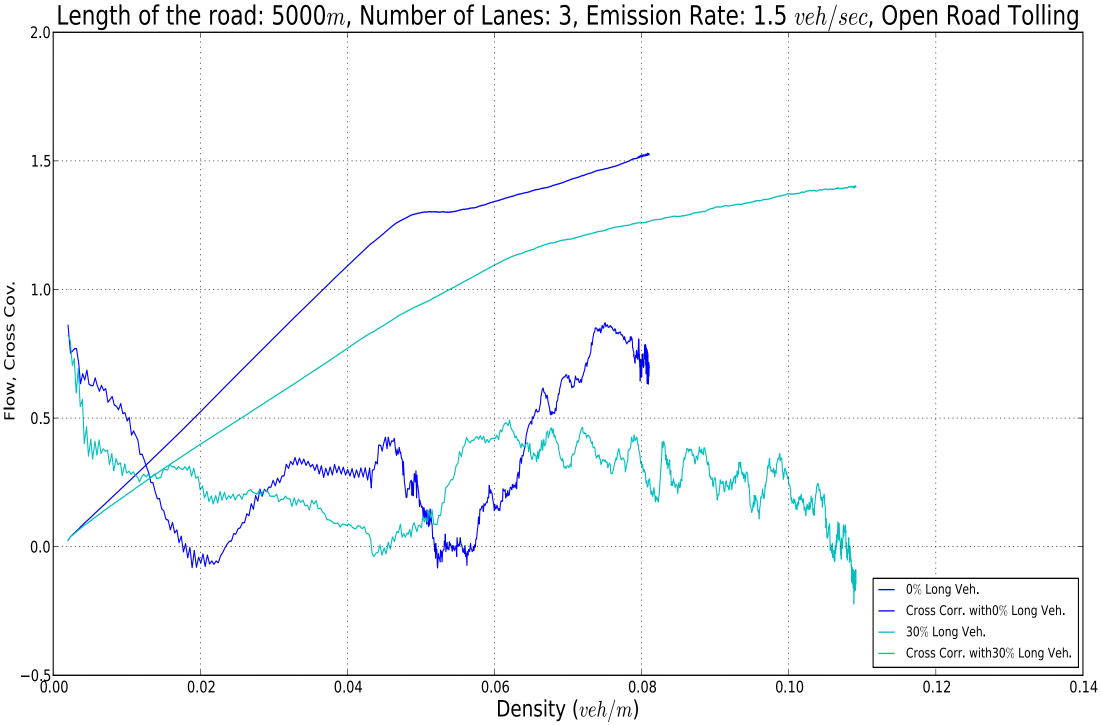
The plateau formation has the dependency also on the throughput. In Figure 6.11, it is seen this situation where we have compared the fundamental diagrams for different influence radii which are related to the average throughputs. If we increase the average throughput, there occurs a more immediate passage from the synchronized flow phase to the wide-moving jam phase. In other words, the phase-change between these two flows occurs with a higher density with the decrease of throughput. Similarly, when we compare the three figures in Figure 6.11, we see that with more long vehicle percentage we have the phase-change with a higher density. The same phenomena is shown with arrows in Figure 6.9.

This phenomena is inverted when an obstacle is placed (on the right-most lane[1]). More precisely, as it is seen in Figure 6.12, when an obstacle is set, we have observed that the phase-change occurs with a lower density when there are long vehicles, probably because when long vehicles get stuck this phase emerges faster. Besides, without an obstacle there is an increase at the slope (absolute value) of the fundamental diagram where there is the wide-moving jam phase, when the percentage of long vehicles is increased (see also Figure 6.9). In other words, the flow decreases faster in presence of long vehicles with the same increment of density, which is not observed in the case of placing an obstacle (the lines are almost parallel, see Figure 6.12). However, this parallelism occurs only when the road is enough saturated. For instance in Figure 6.13 we see that the parallelism occurs in the first and second figures, but not in the third figure since when we compare the second and the third figures, the lane number is increased and so the emission rate per lane is decreased, respectively, which means that in the third figure there is not enough saturation.

Another effect of setting an obstacle is a reduction on the traffic capacity as it is expected. For instance, in the experiments plotted in Figure 6.12, the maximum flow

---

[1] Note that when we placed an obstacle on the left-most lane, we have observed that the result is almost the same with placing it on the right-most lane, so when we make comparisons of having or not having obstacle we use the result of placing it on the right-most lane



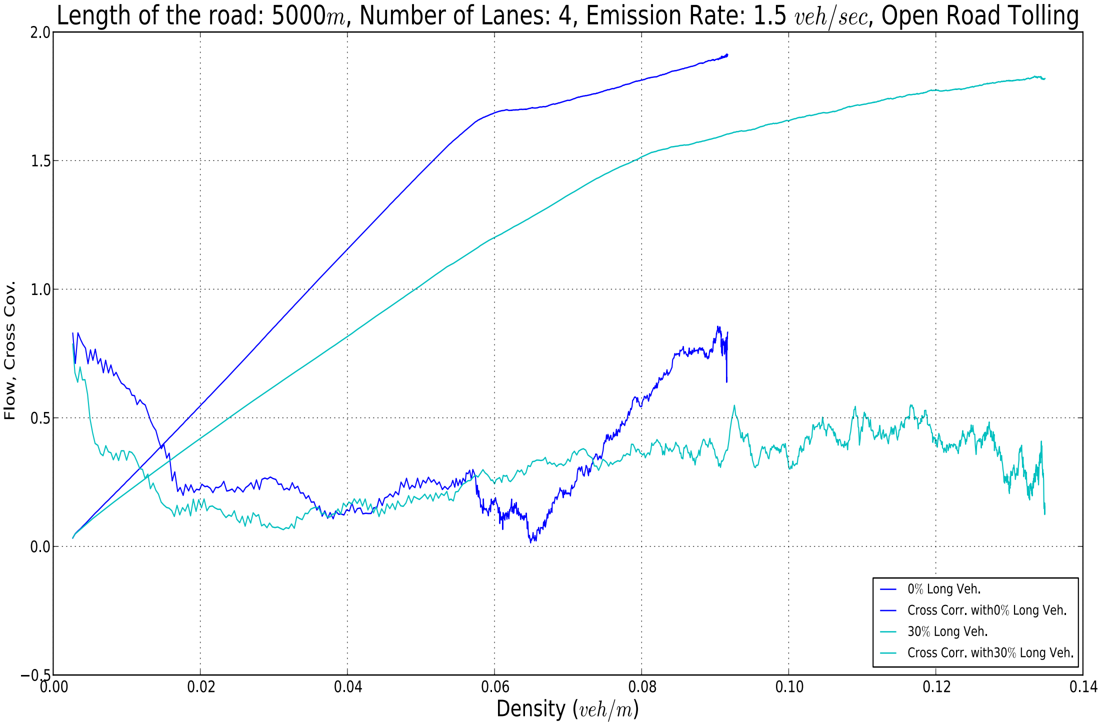


Figure 6.14: The effect of open road tolling on the flow phases with 3 lanes and 4 lanes - These figures correspond to Experiments 32 and 43 of Scenario 2, respectively.

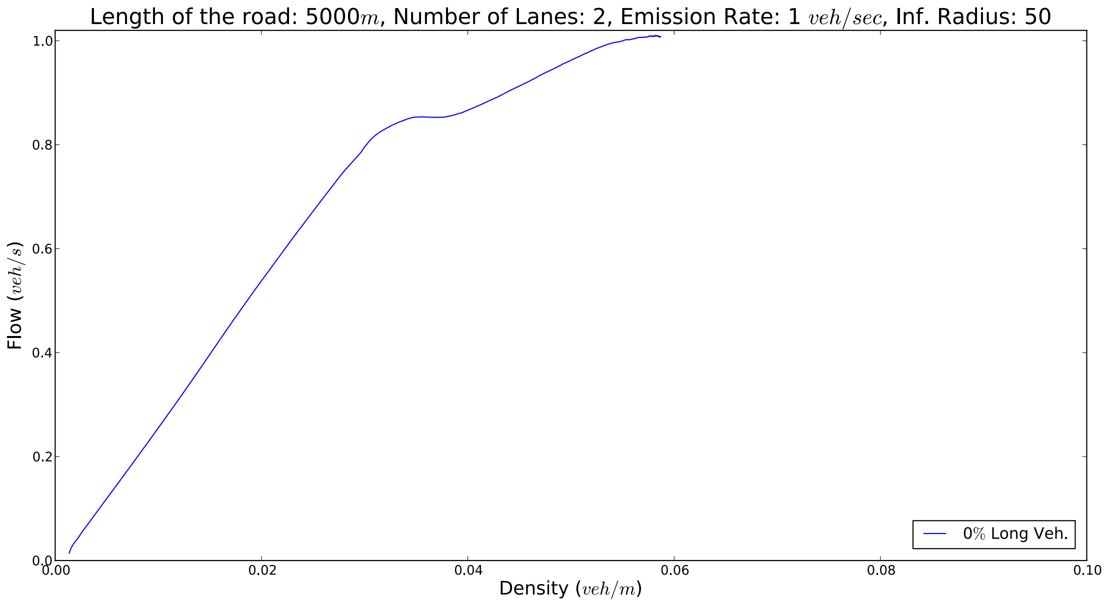


Figure 6.15: The absence of wide-moving jam phase in the case where the vehicles entering are less than the exiting ones - This figure corresponds to Experiment 8 of Scenario 2.

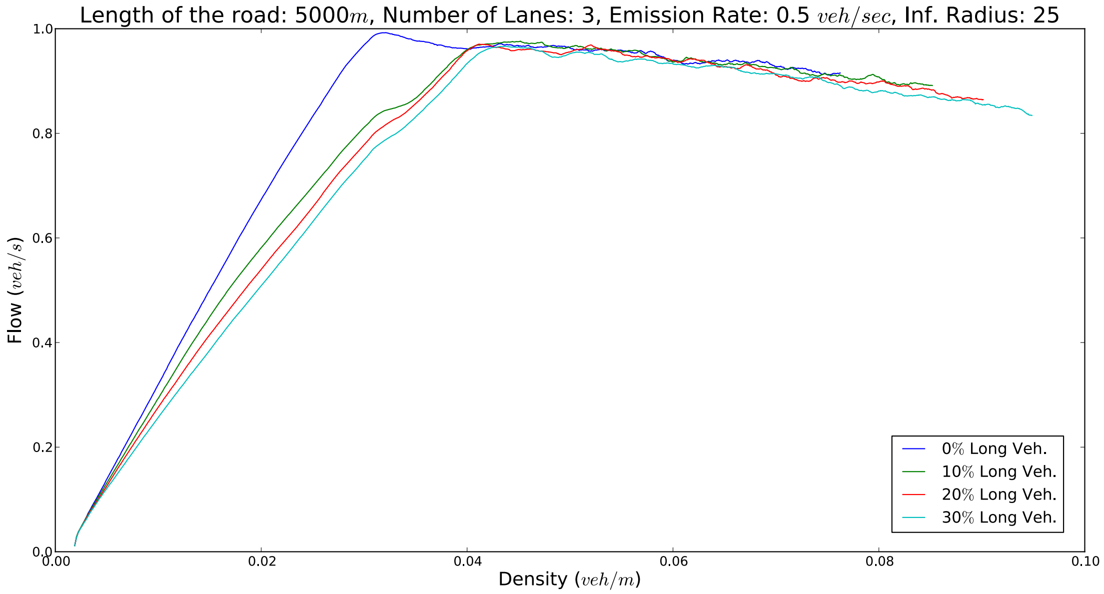


Figure 6.16: The absence of the heterogeneity in the synchronized flow phase in the case where the vehicles entering are less than the exiting ones - This figure corresponds to Experiment 13 of Scenario 2.

that can be reached is around 1.33 veh/sec without an obstacle and around 1.2 veh/sec with an obstacle placed on the right-most lane. It is also seen in Figure 6.12 that this reduction is more visible with the presence of long vehicles.

In the case that there is an open road tolling, the wide-moving jam phase does not take place, so there is no anticorrelation situation. As it is seen in Figure 6.14, there is just a transition between free flow and synchronized flow. The absence of wide-moving jam phase occurs in general with the situations where the emission rate is low with respect to the rate that the vehicles are processed (the influence radius or throughput) at the off-toll plaza as it is expected. For instance, in Experiment 8 of Scenario 2 plotted in Figure 6.15, the emission rate is 1 meaning that 1 passenger vehicle enters to the road each second, and the throughput is reaching around 11 $veh/10\ sec$ meaning that 1.1 passenger vehicles exit from the road each second. Moreover, in these situations also the heterogeneity is lost in the synchronized flow phase as it is seen in Figure 6.16 where the emission rate is 0.5 veh/sec and the number of exiting vehicles reaches around 0.8 veh/sec (throughput is around 8 $veh/10\ sec$).

The experiments also show that the model is able to reproduce the hysteresis phenomena in transition between free flow and synchronized flow phases (see in [30]) and in transition between synchronized flow and wide-moving jam phases as it is seen in Figure 6.17. The regions where there are saddles followed by a capacity drop, as in the Fig.9(e) of [21], are the regions of metastability in the phase-changes (from free flow to synchronized flow and from synchronized flow to wide-moving jam). These metastable states are visible also using the `Real_Time_Visualizator` considering an initial state of 200 vehicles in a free flow phase with velocity of 30 m/s and distance between passenger vehicles of 30 m. Applying a small perturbation to the leading car creates a back propagation wave that brings the system in a new state which is more stable, see Figure 6.18. This phenomena occurs very clearly in this example since it is an extreme situation where the system tends to react very severely. Indeed, in this situation the rules applied are the ones used by the NaSch model to avoid collision, however, in general the system reacts more smoothly and this phenomena is not so emphasized. This fact is also visible in the phase-changes of the fundamental diagrams where the saddles are smooth (curved), see Figure 6.17.

This metastability phenomenon is not so evident when there are long vehicles. More precisely, the heterogeneity of traffic effects the formation of the plateaus. In Figure 6.9

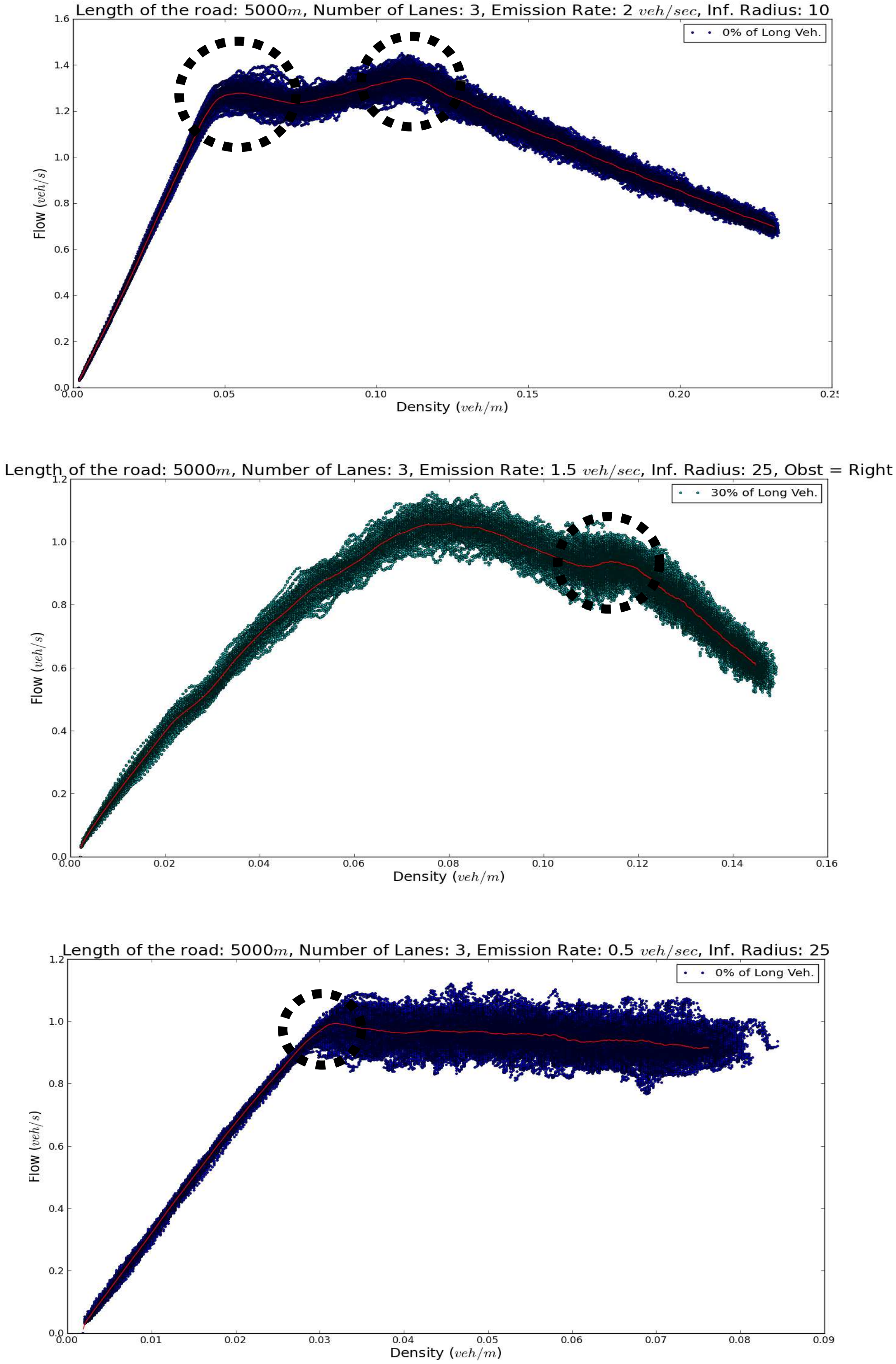


Figure 6.17: The scatter plots of the fundamental diagrams showing the metastability phenomenon in transitions between phases - These figures correspond to Experiments 35, 28 and 13 of Scenario 2, respectively.

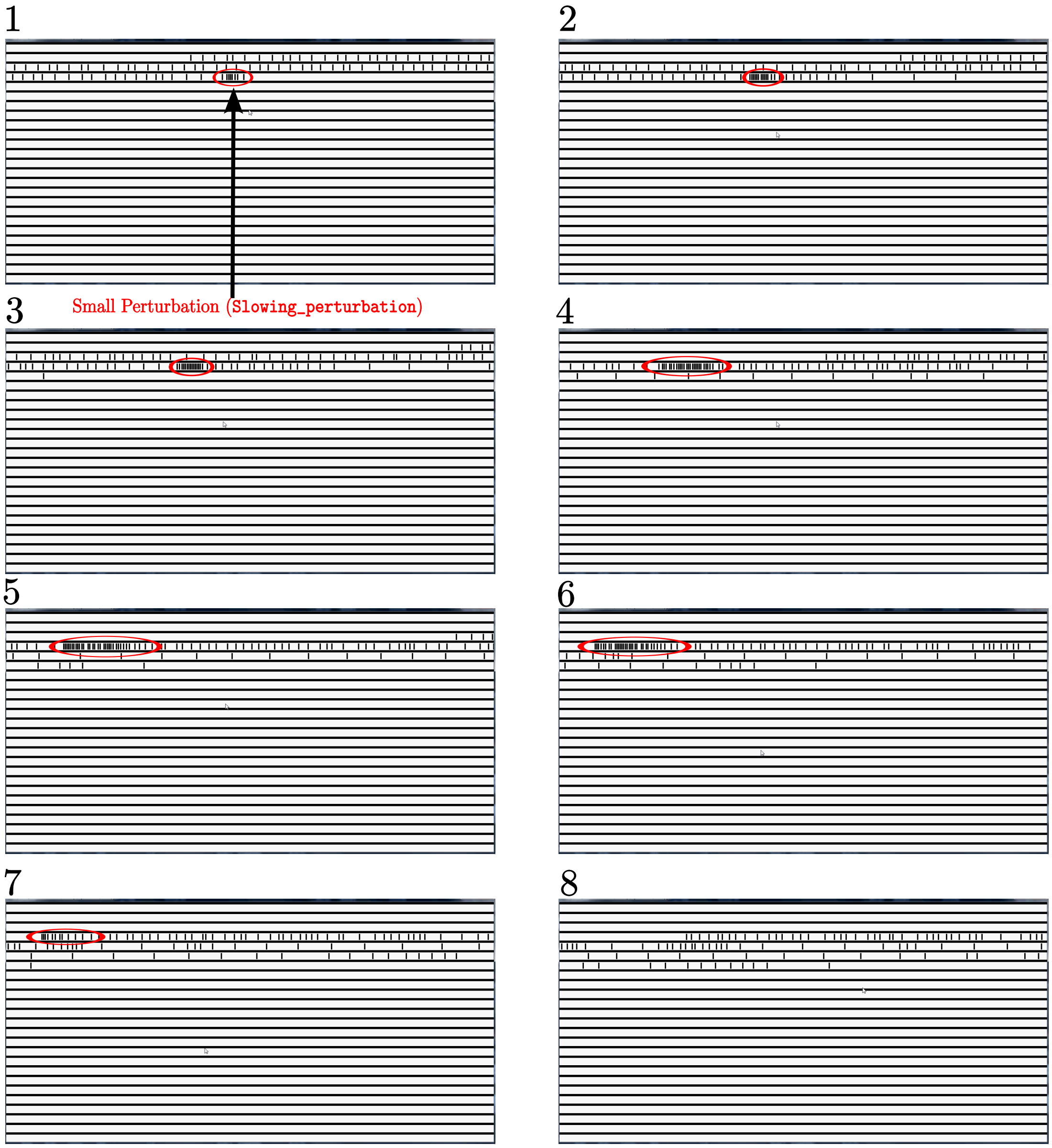


Figure 6.18: An example of a metastable state and a back propagation wave effect - The metastable state in frame 1 after a small perturbation evolves into a new stable state shown in frame 8.

and Figure 6.11, it is seen that in the case of the presence of long vehicles, the bumpy plateaus in the synchronized flow phase are replaced by more flattened plateaus, so the saddles in the phase-changes are not observed as they are in the absence of long vehicles. This is probably due to the fact that passenger vehicles are fast and so the flow of a traffic without long vehicles changes its phase more sharply. However, we observe that the presence of obstacles increases the number of the saddles even when there are long vehicles, making the fundamental diagram more bumpy especially in the synchronized phase (see Figure 6.13 and the second plot of Figure 6.17).

Another phenomenon is observed in the latency diagrams. In the experiments without repetition, it is seen that the latency oscillates and the amplitude of this oscillation increases as the traffic becomes more jammed (see the first and the second plot of Figure 6.19). It is clear that this oscillation is not seen when we average on many repetitions (see the third and the sixth plot of Figure 6.19), thus this phenomena is probably an amplification of the noise due to the traffic congestion: the more the traffic is congested, the more there are pockets of vehicles which arrive much more before than some others with differences of around 1000 seconds. Indeed, when there is open road tolling situation and there are only passenger vehicles (see the fifth plot of Figure 6.19 with 0% Long Vehicles) or when there is light traffic and there are only passenger vehicles (see the fourth plot of Figure 6.19 with 0% Long Vehicles) in the traffic stream, this phenomenon is not visible and there is no this amplification effect. However, it is observed that the presence of long vehicles creates some oscillations in the situations where there is open road tolling (see the fifth plot of Figure 6.19 with 30% Long Vehicles) or light traffic (see the fourth plot of Figure 6.19 with 30% Long Vehicles). We also observe that when there is open road tolling, the latency is almost constant, whereas in other situations it is increasing as it is expected.

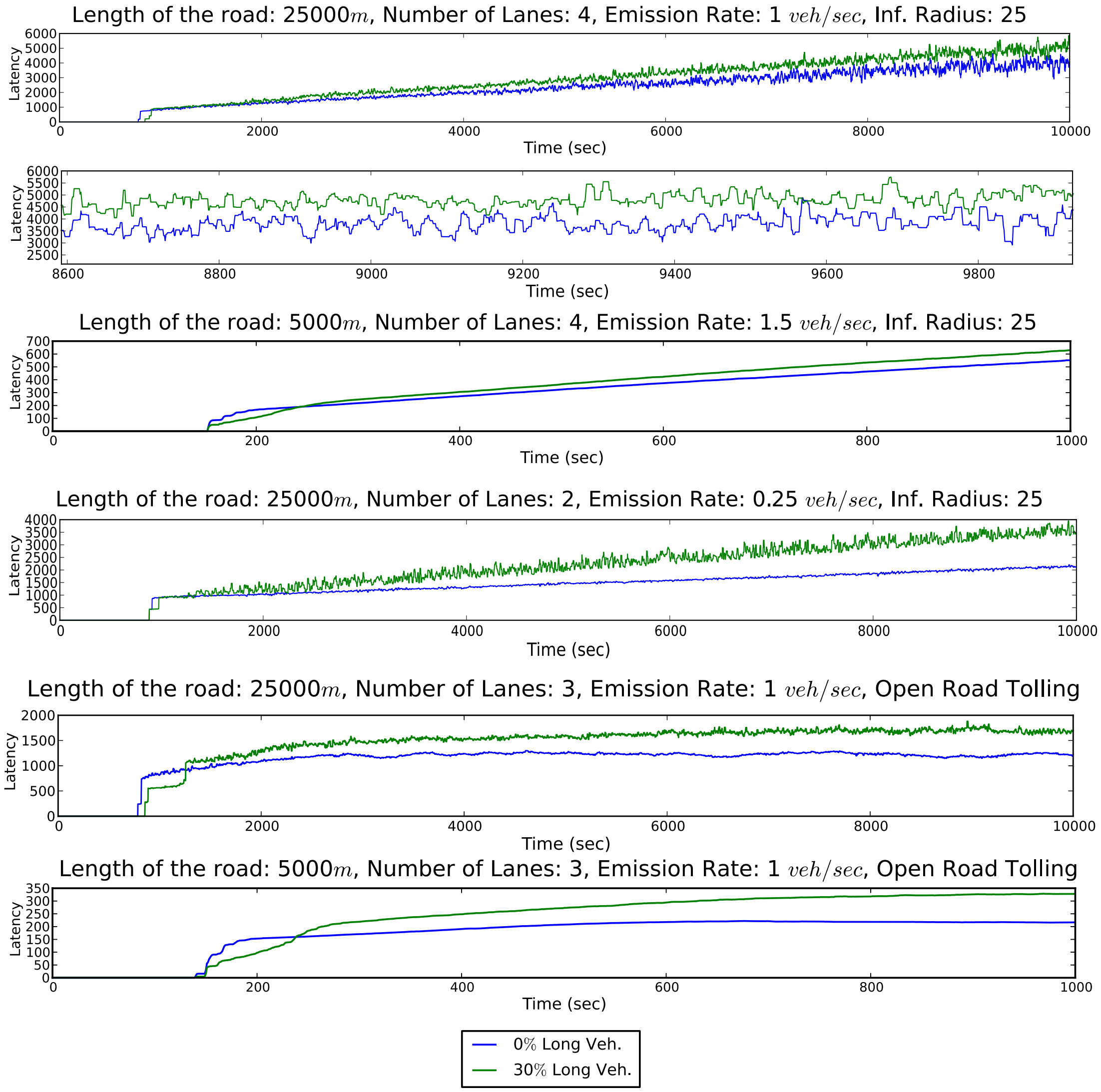


Figure 6.19: Latency in different situations - These figures correspond to Experiments: 31 of Scenario 1, the magnified version of previous experiment, 40 of Scenario 2, 1 and 22 of Scenario 1, and 22 of Scenario 2, respectively.

Chapter 7

Conclusion and Future Work

In this dissertation, we have introduced a new traffic model using continuous CA which is completely detached from the previous CA models defined in literature. The aim was to consider a hybrid between usual microscopic models, very accurate in predicting general traffic behavior but computationally expensive, and usual CA models very efficient due to their simplicity and intrinsic parallelism which make them natural to be implemented for parallel computing. This process of passing from the typical coarse-granularity usual of CA models to the continuity of the typical microscopic models gives us also the advantage of embedding a multi-agent system based on fuzzy decision rules to mimic different driver behaviors. This passage is done with a change of vision with respect to the other CA models. Indeed, instead of considering cells as space we have considered cells which are vehicles, in this way we have made independent the time of computation from the length of the road. Thus, we have first defined a continuous cellular automata model for a single-lane road and then we have extended this model to the multi-lane case. This extension, although natural, is non-trivial.

In the process of the extension, we have first presented the model as an array of communicating one-dimensional CCA, and successively we have proved that this model can be simulated by a suitable CCA. In this way, we have framed our multi-lane model inside the category of CCA. Finally, we have implemented the model using Python 2.7 in the simulator `ozsim.py` using an object-oriented philosophy of programming. Using a questionary we have set up two kinds of vehicles which we have used to run a series of experiments to give a first test to our model. Experimental work has been conducted and the results we have obtained seem promising. Analyzing the experimental results, we have focused on the behavior of heterogeneous traffic such as the effect of different composition of vehicles, and the influence of this heterogeneity on the macroscopic behavior of the traffic in order to study the typical traffic flow phenomena.

Although the simulator is not optimized, it can simulate several vehicles (around 1500) in a real-time visualization mode on a laptop equipped with a processor $i7$ intel© with a frame-rate of 2 FPS. On a more powerful machine[1], when the simulator does not run in real-time visualization mode, the speed increases to a factor of 4 with around 6600 vehicles.

The simulator is able to reproduce the basic traffic phenomena showing a variety of effects due to the heterogeneity in traffic. However, this analysis is not conclusive, but gives just an insight of the potentiality of our model. For this reason, we suggest the following tasks as future works and research directions to improve and validate the model and the simulator:

- We did not use all the potentiality of the code since our aim was to give a first evaluation to our model. However, it would be interesting to consider also the experiments involving on- and off-ramps and loop-detectors to analyze different and more realistic situations.

- The heterogeneity that we have considered is reduced to two kinds of vehicles. A natural question is how the system reacts introducing other kinds. For instance, in highway environments, motorcycles or sport vehicles can be added to the mixed traffic of passenger vehicles and long vehicles. This also brings with it the interesting issue of how to tune the membership functions for such new kinds.

- Explore the possibility of extending our approach to other kind of traffic models, such as city roads with many interactions, traffic lights, etc.

- The process of lane-changing is purely stochastic. However, in literature there are some attempts in microscopic models where the process of lane-changing is described by using a fuzzy logic-based system [7, 18], thus it would be interesting to extend our model to a model in which it is implemented a fuzzy logic-based system to refine the lane-changing rules.

---

[1] On a computer equipped with a 2.7GHz ($X5550$) processor, with 16GB of RAM, where 4 seconds in the simulation are simulated in 1 second.

- The model has to be compared with real data. In other words, a careful case study on specific scenarios with the data available is necessary for the validation by the community of people working on traffic flow theory, granular flow theory and traffic (transportation) engineering.

Table 7.1: Computation Time Comparison between CPU and GPU

Number of Vehicles (per lane)	CPU (sec)	GPU (sec)
0; 0; 1500	484	45,8
0; 0; 3000	958	104
0; 0; 5000	1608	194
0; 0; 10000	4270	556
5000; 5000; 5000	9679	1288

The code written in Python does not take advantage of the CA structure to run by using the parallel computing paradigm. For this reason, we have also adapted the code using PyCuda to parallelize the algorithm on GPU's and we have seen that it is possible to boost the speed of execution to have higher factors of simulation. This is important also to adapt the model for a forecasting usage. In Table 7.1, it is given the time of computation for 1000 *sec* of simulation using the CPU implemented by Python (`ozsim.py`) and the GPU implemented by PyCuda (`cozsim.py`, see Appendix B for a first draft of the simulator) on a laptop equipped with a processor *i*7 intel© and with a graphic card NVIDIA GeForce GT 555M. This computation time comparison is made simulating a road with 3 lanes and the number of vehicles are given with respect to the positions of the lanes.

Appendix A

The Python Code of the Simulator

```
# Ozsim a multi-lane roads simulator

from __future__ import division
import sys, os
sys.path.append(os.getcwd())


import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties
import numpy
from numpy import *
import random
import math
import time
import copy
import pickle
# glumpy is requested only by Real_Time_Visualizer
import glumpy


global AccNoise

# unit of time of the simulation
unitTime = 1
# it activates the fact of having an acceleration noise
# gaussian with standard deviation defined in class vehicles
AccNoise = True
```

```
# it actives the possibility of randomly slowing down
# the velocity of a value below 1 m/s.
slowNoise = False


"""definitions of classes"""

class vehicles():
  def __init__(self,name = None,color = None,
               optV = None,maxV = None,length = None,
               PVS = None, PS = None, PN = None, PB = None,NVS = None,
               VS = None, S = None, N = None, B = None,
               backS = None, jamVelS = None,
               accVS = None, accPS = None, accP = None, accPB = None,
               accNS = None, accN = None, accNB = None,
               accNoise = None, consuption = None,
               maxstress = None, minstress = None,
               LCRP = None, LCLP = None, engEff = None):
    self.name = name
    self.color = color
    self.optV = optV
    self.maxV = maxV
    self.length = length
    self.PVS = PVS
    self.PS = PS
    self.PN = PN
    self.PB = PB
    self.NVS = NVS
    self.VS = VS
    self.S = S
    self.N = N
    self.B = B
    self.backS = backS
    self.jamVelS = jamVelS
    self.accVS = accVS
    self.accPS = accPS
    self.accP = accP
    self.accPB = accPB
    self.accNS = accNS
    self.accN = accN
    self.accNB = accNB
    self.accNoise = accNoise
    self.consuption = consuption
    self.maxstress = maxstress
    self.minstress = minstress
```

```
    self.LCRP = LCRP
    self.LCLP = LCLP
    self.engEff = engEff


class cars(object):
  def __init__(self,position = None,velocity = None,
               kind = None,stress = 0):
    self.position = position
    self.velocity = velocity
    self.kind = kind
    self.stress = stress
    # if checked, it has already changed lane
    self.alreadyDone = False
    # external object or ‘‘not a vehicle mode” False
    self.extObj = False
    # the car is visible to the others
    self.visibility = True
    # internal timer set to zero
    self.timer = 0

  def addStress(self,ammount):
    self.stress += ammount
    if self.stress >= self.kind.maxstress:
      self.stress = self.kind.maxstress
    elif self.stress <= self.kind.minstress:
      self.stress = self.kind.minstress

  def evalFeelings(self, lane):
    # nstress is always positive
    if self.stress >= 0:
      if self.kind.maxstress is 0:
        nstress = 0
      else:
        nstress = self.stress/self.kind.maxstress
      if random.random() < self.kind.LCRP(nstress):
        return lane.right
    else:
      if self.kind.minstress is 0:
        nstress = 0
      else:
        nstress = (self.stress/self.kind.minstress)
      if random.random() < self.kind.LCLP(nstress):
        # if you are in a jam situation, try to change lane
        # to get out from the jam
```

```
        if random.random() < lin(self.velocity, self.kind.jamVelS):
          if lane.right is None:
            return lane.left
          if lane.left is None:
            return lane.right
          if random.random() < 0.7:
            return lane.left
          else:
            return lane.right
          # uncomment to reduce the ping-pong effect
          # if it is commented then the system seems more reactive
          # in a jam situation
          #self.stress = 0
        else:
          return lane.left
    return None


class external(cars):
  def __init__(self,position = None,kind = None,visibility = None,
               emissionRate = None, kindDistribution = None,
               initialVelocity = None, absorptionProb = None,
               influenceRadius = None,probe = False,bufferCapacity = None):
    self.velocity = 0
    self.position = position
    self.kind = kind
    # external object or ‘‘not a vehicle mode” True
    self.extObj = True
    # the car is visible to the others
    self.visibility = False
    self.emissionRate = emissionRate
    self.kindDistribution = kindDistribution
    self.initialVelocity = 12
    self.absorptionProb = absorptionProb
    self.influenceRadius = influenceRadius
    # loop detector modality off
    self.probe = probe
    self.counter = 0
    self.buffer = []
    self.bufferCapacity =  bufferCapacity

  def obstacle(self, dimension, color):
    obstacle = vehicles()
    obstacle.length = dimension
    obstacle.color = color
```

```
    obstacle.name = 'Obstacle'
    self.kind = obstacle
    self.visibility = True

  def onRamp(self,emissionRate,kindDistribution,bufferCapacity,color):
    ramp = vehicles()
    ramp.length = 3
    ramp.color = color
    ramp.name = 'On Ramp'
    self.kind = ramp
    self.visibility = False
    self.emissionRate = emissionRate
    self.kindDistribution = kindDistribution
    self.bufferCapacity = bufferCapacity

  def offRamp(self,absorptionProb,influenceRadius,bufferCapacity,
              sampRate,color):
    ramp = vehicles()
    ramp.length = 3
    ramp.color = color
    ramp.name = 'Off Ramp'
    self.kind = ramp
    self.absorptionProb = absorptionProb
    self.influenceRadius = influenceRadius
    self.bufferCapacity = bufferCapacity
    self.sampRate = sampRate
    self.avLatency = None
    self.throughput = None
    # if influence radius is negative then make it invisible
    # (in this case the vehicles do not slow down near the off ramp)
    if influenceRadius < 0:
      self.visibility = False
      self.influenceRadius = 50
    else:
      self.visibility = True

  def loopDetector(self, influenceRadius, color):
    ramp = vehicles()
    ramp.length = 1
    ramp.color = color
    ramp.name = 'Loop Detector'
    self.kind = ramp
    self.visibility = False
    self.influenceRadius = influenceRadius
    self.probe = True
```

```
class lane(object):
  def __init__(self,ilist = None,left = None,right = None):
    self.left = left
    self.right = right
    if ilist is not None:
      l=copy.deepcopy(ilist)
    else:
      l=[]
    dummy=vehicles()
    dummy.name = 'Dummy'
    dummy.maxV = 40
    dummy.length = 0
    first_dummy = cars(-200, 0, dummy)
    first_dummy.extObj = True
    self.first_dummy = first_dummy
    l.insert(0, first_dummy)
    if self.left is not None:
      left_dummy_position = self.left.last_dummy.position
    else:
      left_dummy_position = 0
    if self.right is not None:
      right_dummy_position = self.right.last_dummy.position
    else:
      right_dummy_position = 0
    if ilist is not None:
      length = len(ilist)
      last_vehicle = ilist[length - 1]
      last_vehicle_position = last_vehicle.position
    else:
      last_vehicle_position = 0
    potion_dummy=max(left_dummy_position, right_dummy_position,
                     last_vehicle_position + 1000)
    last_dummy = cars(potion_dummy, dummy.maxV, dummy)
    last_dummy.extObj = True
    self.last_dummy = last_dummy
    l.append(last_dummy)
    self.contents = l

  def setLeft(self, left):
    self.left = left

  def getLeft(self):
    return(self.left)
```

```
  def getRightmost(self):
    lane = self
    while lane.right is None:
      lane = lane.right
    return lane

  def setRight(self, right):
    self.right = right

  def getRight(self):
    return(self.right)

  def delete(self,car):
    del self.contents[self.contents.index(car)]

  def evalChanges(self):
    for c in self.contents:
      if not c.extObj:
        if not c.alreadyDone:
          newLane = c.evalFeelings(self)
          if newLane is not None:
            if self.transfer(c,newLane):
              self.delete(c)
      elif c is not self.first_dummy and c is not self.last_dummy:
        self.evalExternal(c)

  def eval(self):
    self.evalChanges()
    x, y = transition_function(self, AccNoise, slowNoise)
    self.LaneThroughput = x
    self.LaneLatency = y

  def transfer(self, car, tolane):
    if tolane is not None:
      i = tolane._possibleCar(car)
      if i is not None:
        tolane.contents.insert(i,car)
        car.alreadyDone = True
        # sense of satisfaction after changing lane
        car.stress /= 5
        return True
    return False

  def _possibleCar(self, car):
    indx, precIndx = self._index(car)
```

```
  if indx is not None:
      x = (car.position - self.contents[precIndx].position
          - car.kind.length - self.contents[precIndx].kind.length)
      y = (self.contents[indx].position - car.position
          - car.kind.length - self.contents[indx].kind.length)
     if ((x > (self.contents[precIndx].velocity)**1.2 - car.velocity +
          abs(self.contents[precIndx].velocity - car.velocity) + 3) and
         (3 + (car.velocity)**1.25 - self.contents[indx].velocity < y)):
        return indx
  else:
    return None

def _index(self, car):
  front_position = car.position + car.kind.length
  back_position = car.position - car.kind.length
  if not car.extObj:
    # it returns the indices of the closest
    # front and back visible cars
    precVehicle = 0
    for i,c in enumerate(self.contents):
     if (front_position <= c.position - c.kind.length and c.visibility):
      if (self.contents[precVehicle].position
          + self.contents[precVehicle].kind.length <= back_position):
        return (i, precVehicle)
      else:
        return (None, None)
     else:
      if c.visibility:
        precVehicle = i
    return (None, None)
  elif car.visibility:
    # if the car is a visible external object, it finds the position
    # among all the other cars (visible and invisible)
    precVehicle = 0
    for i,c in enumerate(self.contents):
      if (front_position <= c.position - c.kind.length):
        if (self.contents[precVehicle].position
            + self.contents[precVehicle].kind.length <= back_position):
          return (i, precVehicle)
        else:
          return (None, None)
      else:
        precVehicle = i
    return (None, None)
  else:
```

```
      # if it is invisible I choose the position in the list
      # according to the condition
      # front_position <= c.position - c.kind.length
      for i,c in enumerate(self.contents):
        if (front_position <= c.position - c.kind.length):
          return (i,0)


  def evalExternal(self, ext):
    # update the position (as index) on the lane
    self.delete(ext)
    newIndex = self._index(ext)[0]
    self.contents.insert(newIndex, ext)
    # if it is an emitter
    if ext.emissionRate is not None and ext.kindDistribution is not None:
      #print len(ext.buffer)
      if len(ext.buffer) < ext.bufferCapacity:
        pos = ext.position
        rate = ext.emissionRate
        # using a poisson distribution we calculate the probability
        # of having at least one occurrence of a vehicle in the interval
        # [0,ext.counter + 1]
        prob = 1 - (math.exp(-(rate * (ext.counter + 1) * unitTime)))
        # if the random test succeeds, randomly choose a kind of vehicle
        # distributed as kindDistribution
        if random.random() <= prob:
          Sum=0
          for (kind, percentage) in ext.kindDistribution:
            Sum += percentage
          rand = random.choice(range(Sum))
          rand += 1
          scan=0
          for (kind, percentage) in ext.kindDistribution:
            scan += percentage
            if rand <= scan:
              chosenVehicle = kind
              break
          newCar = cars(pos, ext.initialVelocity, chosenVehicle)
          ext.buffer.append(newCar)
          ext.counter = 0
        else:
          ext.counter += 1
        if ext.buffer != []:
          i = self._possibleCar(ext.buffer[0])
          if i is not None:
```

```
            ext.buffer[0].alreadyDone = True
            self.contents.insert(i, ext.buffer[0])
            del ext.buffer[0]
      else:
        if ext.buffer != []:
          i = self._possibleCar(ext.buffer[0])
          if i is not None:
            ext.buffer[0].alreadyDone = True
            self.contents.insert(i, ext.buffer[0])
            del ext.buffer[0]
        # not able to insert more vehicles in the buffer
        return False
    # if it is a sink
    if ext.absorptionProb is not None and ext.influenceRadius is not None:
      if len(ext.buffer) <= ext.bufferCapacity:
        # this case is used to simulate open road tolling system
        # in the off-toll plaza
        if ext.influenceRadius < 0:
          influencePosition = (ext.position - ext.kind.length - 50)
          for c in self.contents:
            if ((influencePosition <= c.position + c.kind.length)
              and not c.extObj):
              # if you capture it then store it in the buffer
              if random.random() <= ext.absorptionProb:
                if ext.bufferCapacity != 0:
                  ext.buffer.append(c)
                  self.delete(c)
                else:
                  # if bufferCapacity is 0 then the capacity
                  # of the buffer is infinite
                  self.delete(c)
              else:
                carPosition = c.position
                c.position = carPosition + ext.influenceRadius
                             + ext.kind.length
                # otherwise, teleport it beyond the ramp
                if self.transfer(c, self):
                  self.delete(c)
                # if you cannot do it leave it there
                else:
                  c.position = carPosition
        else:
          influencePosition = (ext.position - ext.kind.length
                               - ext.influenceRadius)
          for c in self.contents:
```

```
        if ((influencePosition <= c.position + c.kind.length
             <= ext.position - ext.kind.length) and not c.extObj):
          # if you capture it then store it in the buffer
          if random.random() <= ext.absorptionProb:
            if ext.bufferCapacity != 0:
              ext.buffer.append(c)
              self.delete(c)
            else:
              # if bufferCapacity is 0 then the capacity
              # of the buffer is infinite
              self.delete(c)
          else:
            carPosition = c.position
            c.position = carPosition + ext.influenceRadius
                         + ext.kind.length
            # otherwise teleport it beyond the ramp
            if self.transfer(c, self):
              self.delete(c)
            # if you cannot do, leave it there
            else:
              c.position = carPosition
  else:
    # not able to store in the buffer
    return False
  if ext.sampRate is not None:
    if ext.counter is ext.sampRate:
      T = 0
      for c in ext.buffer:
        T += c.timer
      if ext.buffer != []:
        ext.avLatency = T/len(ext.buffer)
        ext.throughput = len(ext.buffer)
      ext.buffer = []
      ext.counter = 0
    else:
      ext.counter += 1
# if it is a loop detector
if ext.probe:
  influencePosition = ext.position - ext.kind.length
                      - ext.influenceRadius
  # saving the vehicles within the influence radius
  j = newIndex - 1
  newList = []
  while 0 < j < newIndex:
    if not self.contents[j].extObj:
```

```
          if (influencePosition <= self.contents[j].position
              + self.contents[j].kind.length <= ext.position
              - ext.kind.length):
            newList.append(self.contents[j])
            j -= 1
          else:
            break
        else:
          j -= 1
      missingVehicles = len([x for x in ext.buffer if x not in newList])
      ext.counter += missingVehicles
      ext.buffer = newList

  def returnLane(self, num):
    # given a number and lane it returns
    # the lane of distance num from self
    lane = self
    if num >= 0:
      for i in range(num):
        lane = lane.getRight()
      return lane
    if num < 0:
      for i in range(num):
        lane = lane.getLeft()
      return lane

  def createObstacle(self, pos = None, dimension = 200, color = 0.45):
    if pos is None:
      l = len(self.contents)
      indexVehicles = range(l)
      indexVehicles.reverse()
      for j in indexVehicles:
        if not self.contents[j].extObj:
          pos = self.contents[j].position + 200
          break
    if pos >= self.last_dummy.position:
      self.last_dummy.position = pos + 1000
    obst = external(pos + dimension)
    obst.obstacle(dimension, color)
    i = self._index(obst)[0]
    if i is not None:
      self.contents.insert(i, obst)
      return pos
    else:
      return None
```

```

  def createOnRamp(self, pos = None, emissionRate = None,
                   kindDistribution = None, bufferCapacity = 100,
                   color = 0.33):
    l = len(self.contents)
    if pos is None:
      pos = cont[l-2].position + 200
    if pos >= self.last_dummy.position:
      self.last_dummy.position = pos + 1000
    OnRamp = external(pos)
    OnRamp.onRamp(emissionRate, kindDistribution, bufferCapacity, color)
    i = self._index(OnRamp)[0]
    if i is not None:
      self.contents.insert(i, OnRamp)
      return pos
    else:
      return None

  def createOffRamp(self, pos = None, absorptionProb = None,
                   influenceRadius = None, bufferCapacity = 0,
                   sampRate = None, color = 0.5):
    l = len(self.contents)
    if pos is None:
      pos = cont[l-2].position + 200
    if pos >= self.last_dummy.position:
      self.last_dummy.position = pos + 1000
    if (sampRate is not None and bufferCapacity is not None
        and bufferCapacity < sampRate):
      bufferCapacity = sampRate
    OffRamp = external(pos)
    OffRamp.offRamp(absorptionProb, influenceRadius, bufferCapacity,
                    sampRate, color)
    i = self._index(OffRamp)[0]
    if i is not None:
      self.contents.insert(i, OffRamp)
      return pos
    else:
      return None

  def createLoopDetector(self, pos, influenceRadius = 36, color = 0):
    l = len(self.contents)
    if pos is None:
      pos = cont[l-2].position + 200
    if pos >= self.last_dummy.position:
      self.last_dummy.position = pos + 1000
```

```
  loopDet = external(pos)
  loopDet.loopDetector(influenceRadius, color)
  i = self._index(loopDet)[0]
  self.contents.insert(i, loopDet)

def avDistance(self):
  # it gives the average distance between the vehicles
  l = 0
  lastPosition = 0
  sum = 0
  for car in self.contents:
    if not car.extObj and l > 0:
      distance = car.position - lastPosition - car.kind.length
      lastPosition = car.position + car.kind.length
      sum += distance
    l += 1
  if l <= 1:
    return 0
  else:
    return (sum/(l - 1), l-1)

def avVelocity(self):
# it gives the average velocity of the group of vehicles on a lane
  l = 0
  sum = 0
  for car in self.contents:
    if not car.extObj:
      sum += car.velocity
      l += 1
  if l is 0:
    return (0,0)
  else:
    return (sum/l, l)

def fuelCons(self):
# it gives the fuel consumption of the vehicles on a lane
  Sum = 0
  for car in self.contents:
    if not car.extObj:
      V = car.velocity
      kindVehicle = car.kind
      Sum += eval(kindVehicle.consumption)
  return Sum

```

```


”””operations on the cellular automaton”””

def updating_function(car, lane, frontDecidedVelocity,
                       frontDistance, backDistance, frontCollisionTime,
                       backCollisionTime, nextFrontCollisionTime,
                       nextFrontDistance, AccNoise, slowNoise):
  # the local transition function for the single-lane
  # updates the state of a vehicle on a lane
  kindVehicle = car.kind
  position = car.position
  velocity = car.velocity
  v_max = kindVehicle.maxV
  confortable_velocity = kindVehicle.optV
  # the vehicle tries to keep its optimal velocity if it cannot change
      lane
  # slowParameter: we introduce a fake collisionTime depending on
  # a slowParameter which simulates the presence of a front vehicle
  # to make the vehicle slow down in the case of having positive stress
  if not car.alreadyDone and car.stress > 0 and lane.right is None:
    slowParameter = (car.kind.maxstress - car.stress)/(0.1 + velocity)
    if frontCollisionTime < 0:
      collisionTime = slowParameter
    else:
      collisionTime = min(frontCollisionTime, slowParameter)
  else:
    collisionTime = frontCollisionTime
  # if checked it is introduced an acceleration noise
  if AccNoise:
    sigma = kindVehicle.accNoise
    rand=random.gauss(0, sigma)
    acceleration = fuzzy_agent(car, collisionTime, backCollisionTime,
                               frontDistance, backDistance,
                               nextFrontCollisionTime, nextFrontDistance)
                               + rand
  else:
    acceleration = fuzzy_agent(car, collisionTime, backCollisionTime,
                               frontDistance, backDistance,
                               nextFrontCollisionTime, nextFrontDistance)
  # the demanded power acceleration depends on the velocity
  if acceleration > 0:
    acceleration *= car.kind.engEff(car.velocity)
  # the fuzzy agent calculates the new velocity
  ChosenVelocity = max(0, velocity + (acceleration * unitTime))
```

```
  # if it does not collide, do not use NaSch!
  if (ChosenVelocity - frontDecidedVelocity + 1) < frontDistance:
    new_velocity = min(v_max, ChosenVelocity)
  else:
  # to avoid accidents due to sudden braking, we apply NaSch rule
    new_velocity = min(v_max, max(max(0, (frontDistance - 1)/unitTime),
                                         frontDecidedVelocity))
  new_position = position + new_velocity * unitTime
  if slowNoise:
    new_velocity = max(0, new_velocity - random.random())
  car.velocity = new_velocity
  car.position = new_position
  car.alreadyDone = False
  car.timer += 1
  car.addStress((velocity - confortable_velocity) * unitTime
                * random.random())
  # this part is devoted to help the system to take some decisions
  # using stress as control parameter
  if (car.kind.minstress/2) < car.stress < 0:
    if frontCollisionTime < 0:
    #sense of satisfaction if the queue is moving
      car.stress /= 2
    else:
    # try to avoid braking with the strategy of changing lane
    # if the front collision time is small or very small and
    # the front distance is normal or small, try to change lane
      CollVerySmall = lin(frontCollisionTime, car.kind.PVS)
      CollSmall = lin(frontCollisionTime, car.kind.PS)
      DistNormal = lin(frontDistance, car.kind.N)
      DistSmall = lin(frontDistance, car.kind.S)
      factor = max(min(CollVerySmall, DistNormal),
                   min(CollVerySmall, DistSmall),
                   min(CollSmall, DistNormal),
                   min(CollSmall, DistSmall))
      car.stress *= (1 + factor)


def transition_function(lane, AccNoise, slowNoise):
  # this function updates the CA for the single lane,
  # it corresponds to the global transition function
  LaneThroughput = 0
  LaneLatency = 0
  Latency = 0
  # state of the cellular automaton which simulates one lane
  state = lane.contents
```

```
numCars = len(state)
# numCars - 1 is the front dummy
prevBackPosition = (state[numCars - 1].position
                    - state[numCars - 1].kind.length)
prevVelocity = state[numCars - 1].velocity
prevNextBackPosition = (state[numCars - 1].position
                        - state[numCars - 1].kind.length)
prevNextVelocity =state[numCars - 1].velocity
# update the front dummy (just the position)
state[numCars - 1].position = (state[numCars - 1].position +
                               state[numCars - 1].velocity * unitTime)
indexVehicles = range(1, numCars - 1)
# we reverse the counter because we need to know the
# front decided velocity for a checking in case of a collision
indexVehicles.reverse()
for j in indexVehicles:
  if not state[j].extObj:
    # check for the closest back vehicle which is visible to him
    k = j - 1
    while k < j:
      if state[k].visibility:
        backVehicle = state[k]
        break
      else:
        k -= 1
    # check for the closest front vehicle which is visible to him
    i = j + 1
    while i > j:
      if state[i].visibility:
        frontVehicle = state[i]
        break
      else:
        i += 1
    frontDistance = (prevBackPosition - state[j].position
                     - state[j].kind.length)
    nextDistance = (prevNextBackPosition - state[j].position
                    - state[j].kind.length)
    backDistance = (state[j].position - backVehicle.position
                    - state[j].kind.length - backVehicle.kind.length)
    deltaVelocity = state[j].velocity - prevVelocity
    deltaNextVelocity = state[j].velocity - prevNextVelocity
    deltaBackVelocity = backVehicle.velocity - state[j].velocity
    if deltaVelocity == 0:
      frontCollisionTime = 999
    else:
```

```
        frontCollisionTime = frontDistance/deltaVelocity
      if deltaNextVelocity == 0:
        nextCollisionTime = 999
      else:
        nextCollisionTime = nextDistance/deltaNextVelocity
      if deltaBackVelocity == 0:
        backCollisionTime = 999
      else:
        backCollisionTime = backDistance/deltaBackVelocity
      # keep memory for the next step
      prevNextBackPosition = prevBackPosition
      prevNextVelocity = prevVelocity
      prevBackPosition = state[j].position - state[j].kind.length
      prevVelocity = state[j].velocity
      updating_function(state[j], lane, frontVehicle.velocity,
                        frontDistance, backDistance,
                        frontCollisionTime, backCollisionTime,
                        nextCollisionTime, nextDistance,
                        AccNoise, slowNoise)
    elif state[j].visibility:
      prevNextBackPosition = prevBackPosition
      prevNextVelocity = prevVelocity
      prevBackPosition = state[j].position - state[j].kind.length
      prevVelocity = state[j].velocity
    if state[j].kind.name =='Off Ramp' and state[j].avLatency is not None:
      LaneThroughput = state[j].throughput
      LaneLatency = state[j].avLatency
  return (LaneThroughput, LaneLatency)

def initial_lane(kind_array, position_array, velocity_array,
                 stress_array = None):
  # initializer of the array of vehicles in one lane
  state=[]
  if (len(kind_array) == len(position_array) and
      len(velocity_array) == len(position_array)):
    length = len(position_array)
    if stress_array is not None:
      for j in range(length):
        car=cars(position_array[j],velocity_array[j],kind_array[j],
                 stress_array[j])
        state.append(car)
      return(state)
    else:
      for j in range(length):
        car = cars(position_array[j],velocity_array[j],kind_array[j])
```

```
          state.append(car)
        return(state)
    else:
      print 'Incompatible arrays: different lengths'



"""functions related to the street and the interaction with it"""

def averageStreetVelocity(leftMostLane):
  lane = leftMostLane
  sumVelocity = 0
  sumNumber = 0
  while lane is not None:
    v, num = lane.avVelocity()
    sumVelocity += num*v
    sumNumber += num
    lane = lane.getRight()
  if sumNumber is 0:
    return 0
  else:
    return sumVelocity/sumNumber

def averageStreetDistance(leftMostLane):
  # it calculates the average distance between vehicles
  lane = leftMostLane
  sumDistance = 0
  sumNumber = 0
  while lane is not None:
    d, num = lane.avDistance()
    sumDistance += num*d
    sumNumber += num
    lane = lane.getRight()
  if sumNumber is 0:
    return 0
  else:
    return (sumDistance/sumNumber, sumNumber)

def updateStreet(leftMostLane):
  # updating of the multilane model, the update is done
  # from left to right (the leftmost has the precedence)
  Throughput = 0
  SumLatency = 0
  numLanes = 0
  lane = leftMostLane
```

```
  while lane is not None:
    lane.eval()
    Throughput += lane.LaneThroughput
    SumLatency += lane.LaneLatency
    numLanes += 1
    lane = lane.getRight()
  #print 'Average Latency: ', SumLatency/numLanes
  #print 'Throughput: ', Throughput
  #print 'Average Velocity: ', averageStreetVelocity(leftMostLane)
  #print ('Average Distance, Num of Vehicles:',
  #        averageStreetDistance(leftMostLane))
  #print '\n'
  AvDist, num = averageStreetDistance(leftMostLane)
  return(Throughput, SumLatency/numLanes,
          averageStreetVelocity(leftMostLane),
          AvDist, num)

def createStreet(rightMostLane, numLanes):
  rightLane = lane(rightMostLane, None, None)
  leftLane = rightLane
  for j in range(numLanes - 1):
    leftLane=lane(left = None, right = rightLane)
    rightLane.setLeft(leftLane)
    rightLane=leftLane
  return(leftLane)

def createOnToll(leftMostLane,position,emissionRate,kindDistribution):
  lane = leftMostLane
  while lane is not None:
    lane.createOnRamp(position, emissionRate, kindDistribution)
    lane = lane.getRight()

def createOffToll(leftMostLane, position, absorptionProb,
                  influenceRadius, bufferCapacity, sampRate):
  lane = leftMostLane
  while lane is not None:
    lane.createOffRamp(position, absorptionProb, influenceRadius,
                       bufferCapacity, sampRate)
    lane = lane.getRight()

def createRandHighway(initRightLane, length, numLanes, emissRate,
                      kindDistribution, numObstacles, numRamps):
  # a simple highway random generator
  initialPosition = 2
  leftMostLane = createStreet(initRightLane, numLanes)
```

```
  rightMostLane = leftMostLane.returnLane(numLanes - 1)
  createOnToll(leftMostLane,initialPosition,emissRate,kindDistribution)
  createOffToll(leftMostLane, length + initialPosition, 1, 25, 100, 10)
  position = initialPosition
  maxSpaceOffOnRamp = 200
  maxInflunceRadius = 30
  if numRamps is 0:
    interval = length
  else:
    interval = length/numRamps
  for i in range(numRamps):
    SpaceOffOnRamp = maxSpaceOffOnRamp*random.random()
    position += random.random()*(interval/2) + (interval/2)
    rightMostLane.createOffRamp(position, random.random(),
                                30 + maxInflunceRadius*random.random())
    rightMostLane.createOnRamp(position + 200 + random.random()*
                               maxSpaceOffOnRamp, random.random(),
                               kindDistribution)
  maxDimObstacle = 70
  position = initialPosition
  if numObstacles is 0:
    interval = length
  else:
    interval = length/numObstacles
  for i in range(numObstacles):
    dimObstacle = 30 + maxDimObstacle*random.random()
    position += (random.random()*(interval/2) + (interval/2)
                 + dimObstacle)
    choosenIndx = random.choice(range(numLanes))
    lane = leftMostLane.returnLane(choosenIndx)
    lane.createObstacle(position, dimObstacle)
  return leftMostLane

def slowing_perturbation(leftmost_lane):
  # it slows down the first vehicle on each lane
  lane = leftmost_lane
  while lane is not None:
    l = len(lane.contents)
    indexVehicles = range(l)
    indexVehicles.reverse()
    for j in indexVehicles:
      if not lane.contents[j].extObj:
        velocity = lane.contents[j].velocity
        lane.contents[j].velocity = velocity/5
        break
```

```
    lane = lane.getRight()

def randObstacle(leftmost_lane):
  # it creates a random obstacle in front of the first vehicle
  lane = leftmost_lane
  numLanes = 0
  while lane is not None:
    numLanes += 1
    lane = lane.getRight()
  obstrLane = None
  while obstrLane is None:
    randIndx = random.choice(range(numLanes))
    obstrLane = leftmost_lane.returnLane(randIndx)
    if obstrLane is not None:
      obstrLane.createObstacle()
      break


"""fuzzy agent"""

def lin(input, function):
  # this function returns the value of the scattered function
  length = len(function)
  for i in range(length):
    if input < function[0][0]:
      return 0
    elif input > function[length -1][0]:
      return 0
    elif function[i][0] <= input <= function[i+1][0]:
      # it finds the position of the input
      position = i
      break
  # linear approximation
  x = function[position][0]
  y = function[position + 1][0]
  f_x = function[position][1]
  f_y = function[position + 1][1]
  if x == y:
    if f_x != f_y:
      print('it is not a function')
    else:
      return(f_x)
  slope=(f_x - f_y)/(x-y)
  return(f_y + slope*(input-y))

```

```
def lin_preimage(input,function):
  # this function computes the preimages of an input of a given
  # function which is represented by a linear interpolation
  length = len(function)
  out=[]
  if input == 0:
    out=[0]
  else:
    for i in range(length-1):
      if function[i+1][1] == function[i][1] == input:
        # plateu case
        out.append(function[i][0])
      else:
        if (function[i+1][1] < input <= function[i][1] or
            function[i][1] <= input<function[i+1][1]):
          slope = ((function[i+1][1] - function[i][1])
                   /(function[i+1][0] - function[i][0]))
          out.append(function[i][0]+((input-function[i][1])/slope))
        if function[length-1][1] == input:
          out.append(function[length-1][0])
  return(out)

def fuzzy_agent(car, tau_plus, tau_minus, delta_plus,
                delta_minus, tauNext, deltaNext):
  # it returns the accelaration that the agent decides
  kindVehicle = car.kind
  collision_PVS = lin(tau_plus, kindVehicle.PVS)
  collision_PS = lin(tau_plus, kindVehicle.PS)
  collision_PN = lin(tau_plus, kindVehicle.PN)
  collision_PB = lin(tau_plus, kindVehicle.PB)
  collision_NVS = lin(tau_minus, kindVehicle.NVS)
  front_distance_VS = lin(delta_plus, kindVehicle.VS)
  front_distance_S = lin(delta_plus, kindVehicle.S)
  front_distance_N = lin(delta_plus, kindVehicle.N)
  front_distance_B = lin(delta_plus, kindVehicle.B)
  back_distance_S = lin(delta_minus, kindVehicle.backS)
  jam_factor = lin(car.velocity, kindVehicle.jamVelS)
  collisionNext_PVS = lin(tauNext, kindVehicle.PVS)
  collisionNext_PS = lin(tauNext, kindVehicle.PS)
  collisionNext_PN = lin(tauNext, kindVehicle.PN)
  collisionNext_PB = lin(tauNext, kindVehicle.PB)
  distanceNext_VS = lin(deltaNext, kindVehicle.VS)
  distanceNext_S = lin(deltaNext, kindVehicle.S)
  distanceNext_N = lin(deltaNext, kindVehicle.N)
  distanceNext_B = lin(deltaNext, kindVehicle.B)
```

```
rules = []
rules.append([min(collision_PB, front_distance_B, 1 - jam_factor),
                   lin_preimage(min(collision_PB, front_distance_B,
                                    1 - jam_factor), kindVehicle.accP)])
rules.append([min(collision_PB, front_distance_N, 1 - jam_factor),
                   lin_preimage(min(collision_PB, front_distance_N,
                                    1 - jam_factor), kindVehicle.accPS)])
rules.append([min(collision_PB, front_distance_S),
                   lin_preimage(min(collision_PB, front_distance_S),
                                kindVehicle.accVS)])
rules.append([min(collision_PB, front_distance_VS),
                   lin_preimage(min(collision_PB, front_distance_VS),
                                kindVehicle.accVS)])
rules.append([min(collision_PN, front_distance_B),
                   lin_preimage(min(collision_PN, front_distance_B),
                                kindVehicle.accVS)])
rules.append([min(collision_PN, front_distance_N),
                   lin_preimage(min(collision_PN, front_distance_N),
                                kindVehicle.accVS)])
rules.append([min(collision_PN, front_distance_S),
                   lin_preimage(min(collision_PN, front_distance_S),
                                kindVehicle.accNS)])
rules.append([min(collision_PN, front_distance_VS),
                   lin_preimage(min(collision_PN, front_distance_VS),
                                kindVehicle.accNS)])
rules.append([min(collision_PS, front_distance_B),
                   lin_preimage(min(collision_PS, front_distance_B),
                                kindVehicle.accN)])
rules.append([min(collision_PS, front_distance_N),
                   lin_preimage(min(collision_PS, front_distance_N),
                                kindVehicle.accN)])
rules.append([min(collision_PS, front_distance_S),
                   lin_preimage(min(collision_PS, front_distance_S),
                                kindVehicle.accN)])
rules.append([min(collision_PS, front_distance_VS),
                   lin_preimage(min(collision_PS, front_distance_VS),
                                kindVehicle.accN)])
rules.append([min(collision_PVS, front_distance_B),
                   lin_preimage(min(collision_PVS, front_distance_B),
                                kindVehicle.accNB)])
rules.append([min(collision_PVS, front_distance_N),
                   lin_preimage(min(collision_PVS, front_distance_N),
                                kindVehicle.accNB)])
rules.append([min(collision_PVS, front_distance_S),
                   lin_preimage(min(collision_PVS, front_distance_S),
```

```
                                   kindVehicle.accNB)])
    rules.append([min(collision_PVS, front_distance_VS),
                      lin_preimage(min(collision_PVS, front_distance_VS),
                                   kindVehicle.accNB)])
    rules.append([min(collision_NVS, collision_PB, back_distance_S,
                      front_distance_B), lin_preimage(min(collision_NVS,
                      collision_PB, back_distance_S, front_distance_B),
                      kindVehicle.accPS)])
    rules.append([min(collision_NVS, collision_PB, back_distance_S,
                      front_distance_N), lin_preimage(min(collision_NVS,
                      collision_PB, back_distance_S, front_distance_N),
                      kindVehicle.accPS)])
    rules.append([min(collision_NVS, collision_PN, back_distance_S,
                      front_distance_B), lin_preimage(min(collision_NVS,
                      collision_PN, back_distance_S, front_distance_B),
                      kindVehicle.accPS)])
    rules.append([min(collision_NVS, collision_PN, back_distance_S,
                      front_distance_N), lin_preimage(min(collision_NVS,
                      collision_PN, back_distance_S, front_distance_N),
                      kindVehicle.accPS)])
    # if the car is in a jam situation
    # and the time of collision is big then
    # strongly accelerate to be more reactive
    rules.append([min(collision_PB, jam_factor),
                  lin_preimage(min(collision_PB, jam_factor),
                               kindVehicle.accPB)])
    # try to keep a safety distance between the vehicles
    # worstCollisionTime is the collision time in the case
    # the front vehicle stops suddenly
    worstCollisionTime = delta_plus/(0.1 + car.velocity)
    worstCollisionTime_PVS = lin(worstCollisionTime, kindVehicle.PVS)
    rules.append([min(worstCollisionTime_PVS, front_distance_VS),
                 lin_preimage(min(worstCollisionTime_PVS,front_distance_VS),
                              kindVehicle.accN)])
    rules.append([min(worstCollisionTime_PVS, front_distance_S),
                 lin_preimage(min(worstCollisionTime_PVS,front_distance_S),
                              kindVehicle.accN)])
    rules.append([min(worstCollisionTime_PVS, front_distance_N),
                 lin_preimage(min(worstCollisionTime_PVS,front_distance_N),
                              kindVehicle.accNS)])
    # for the next front vehicle we have another set of rules
    rulesNext = []
    rulesNext.append([min(collisionNext_PVS, distanceNext_VS),
                       lin_preimage(min(collisionNext_PVS, distanceNext_VS),
                                    kindVehicle.accNB)])
```

```
  rulesNext.append([min(collisionNext_PVS, distanceNext_S),
                    lin_preimage(min(collisionNext_PVS, distanceNext_S),
                                 kindVehicle.accNB)])
  rulesNext.append([min(collisionNext_PVS, distanceNext_N),
                    lin_preimage(min(collisionNext_PVS, distanceNext_N),
                                 kindVehicle.accNB)])
  rulesNext.append([min(collisionNext_PVS, distanceNext_B),
                    lin_preimage(min(collisionNext_PVS, distanceNext_B),
                                 kindVehicle.accN)])
  rulesNext.append([min(collisionNext_PS, distanceNext_VS),
                    lin_preimage(min(collisionNext_PS, distanceNext_VS),
                                 kindVehicle.accN)])
  rulesNext.append([min(collisionNext_PS, distanceNext_S),
                    lin_preimage(min(collisionNext_PS, distanceNext_S),
                                 kindVehicle.accN)])
  rulesNext.append([min(collisionNext_PS, distanceNext_N),
                    lin_preimage(min(collisionNext_PS, distanceNext_N),
                                 kindVehicle.accNS)])
  rulesNext.append([min(collisionNext_PS, distanceNext_B),
                    lin_preimage(min(collisionNext_PS, distanceNext_B),
                                 kindVehicle.accNS)])
  rulesNext.append([min(collisionNext_PN, distanceNext_VS),
                    lin_preimage(min(collisionNext_PN, distanceNext_VS),
                                 kindVehicle.accNS)])
  rulesNext.append([min(collisionNext_PB, distanceNext_VS),
                    lin_preimage(min(collisionNext_PB, distanceNext_VS),
                                 kindVehicle.accNS)])
  # we make a deffuzification for the first set of rules
  num = 0
  den = 0
  for rule in rules:
    # defuzzifier: weighted sum of the preimages
    #       (in the symmetric case is WAF)
    acceleration = rule[1]
    l=len(acceleration)
    sum_acceleration = 0
    for j in range(l):
      sum_acceleration += acceleration[j]
    num += (rule[0] * sum_acceleration)
    den += (rule[0] * l)
  if den == 0:
    Acceleration = 0
  else:
    Acceleration = num/den
  # we make a deffuzification for the second set of rules
```

```
  num=0
  den=0
  for rule in rulesNext:
    # defuzzifier: weighted sum of the preimages
    #       (in the symmetric case is WAF)
    acceleration = rule[1]
    l=len(acceleration)
    sum_acceleration = 0
    for j in range(l):
      sum_acceleration += acceleration[j]
    num += (rule[0] * sum_acceleration)
    den += (rule[0] * l)
  if den == 0:
    AccelerationNext = 0
  else:
    AccelerationNext = num/den
  if Acceleration <= 0:
    return min(Acceleration, AccelerationNext)
  else:
    if AccelerationNext < -0.25:
      return (Acceleration + AccelerationNext)/2
    else:
      return Acceleration


"""visualization functions """

def draw_car(car, index_of_lane, numRoadPiece, height, width, matrix,
             dimension_of_road, separation_width, visual_separation,
             one_lane_width):
  # the index of the leftmost lane is 0
  # it draws a car inside a matrix
  position = car.position
  kind_of_car = car.kind
  back_position = position - kind_of_car.length
  color = kind_of_car.color
  vehicle_width = one_lane_width - 2*visual_separation
  wrap_factor = (back_position//width)%numRoadPiece
  y = (wrap_factor * dimension_of_road + separation_width
       + index_of_lane * one_lane_width + visual_separation)
  # it calculates the position of the vehicle module
  # the border of the screen
  x = round(back_position%width)
  if wrap_factor == numRoadPiece - 1:
    # the last row case
```

```
    if (x + 2 * kind_of_car.length) <= width:
      # it does not go outside the screen
      for i in range(int(2 * kind_of_car.length)):
        for j in range(int(vehicle_width)):
          matrix[int(y) + j][int(x) + i] = color
    else:
      # otherwise, draw the car one piece on this row and
      # we wrap the right down corner with the left up corner,
      # note that the simulation has not closed boundaries,
      # so this is done as a matter of visualization
      for i in range(int(width - x)):
        for j in range(int(vehicle_width)):
          matrix[int(y) + j][int(x) + i] = color
      for i in range(int(2*kind_of_car.length - (width - x))):
        for j in range(int(vehicle_width)):
          matrix[separation_width + index_of_lane * one_lane_width
                 + visual_separation + j][i] = color
  else:
    if (x + 2*kind_of_car.length) <= width:
      # it does not go outside the screen
      for i in range(int(2*kind_of_car.length)):
        for j in range(int(vehicle_width)):
          matrix[int(y) + j][int(x) + i] = color
    else:
      # otherwise, draw the car one piece on this row
      for i in range(int(width - x)):
        for j in range(int(vehicle_width)):
          matrix[int(y) + j][int(x) + i] = color
      for i in range(int(2*kind_of_car.length - (width - x))):
        # the other piece in the next row
        for j in range(int(vehicle_width)):
          matrix[int(y+dimension_of_road) + j][i] = color
  return(matrix)

def visual_position(leftmost_lane, numLanes, numRoadPiece,
                    width, height):
  # it returns a matrix which is the representation of
  # the configuration of each lane (road configuration)
  dimension_of_road = height//numRoadPiece
  separation_width = dimension_of_road//4
  one_lane_width = (dimension_of_road - separation_width)//(numLanes)
  visual_separation = one_lane_width//8
  street_matrix = numpy.ones((height,width)).astype(numpy.float32)
  # it initializes the matrix corresponding to the representation
  # of the street, it draws the separation (in black)
```

```
  for i in range(numRoadPiece):
    for j in range(int(separation_width)):
      for l in range(int(width)):
        street_matrix[i * int(dimension_of_road) + j][l]=0
  if numRoadPiece*int(dimension_of_road)+int(separation_width) <= height:
    for j in range(int(separation_width)):
      for l in range(int(width)):
        street_matrix[numRoadPiece * int(dimension_of_road) + j][l] = 0
  for i in range(numRoadPiece):
    for j in range(1,numLanes):
      for k in range(int(visual_separation)):
        for l in range(int(width)):
          street_matrix[int(dimension_of_road) * i
                        + int(separation_width) + int(one_lane_width) * j
                        - int(visual_separation//2) + k][l] = 0
  lane = leftmost_lane
  index_of_lane = 0
  visual_matrix = street_matrix
  while lane != None:
    for c in lane.contents:
      if c != lane.first_dummy and c != lane.last_dummy:
        # don't draw the dummies
        vehicle_kind = c.kind
        visual_matrix = draw_car(c,index_of_lane,numRoadPiece,height,
                                 width,visual_matrix,dimension_of_road,
                                 separation_width,visual_separation,
                                 one_lane_width)
    lane=lane.getRight()
    index_of_lane += 1
  return(visual_matrix)

# glumpy 1.1
def Real_Time_Visualizator(leftmost_lane, num_of_lanes, numRoadPiece,
                           width, height):
  global state, time, initial_time, frames
  time, initial_time, frames = 0,0,0
  state = leftmost_lane

  window = glumpy.Window(width, height)

  @window.event
  def on_mouse_press(x, y, LEFT):
    global state
    slowing_perturbation(state)
    #randObstacle(state)
```

```

  @window.event
  def on_idle(*args):
    global state, time, initial_time, frames, fuel
    fuel = 0
    window.clear()
    V = visual_position(state,num_of_lanes,numRoadPiece,width,height)
    I = glumpy.Image(V,cmap=glumpy.colormap.Hot, vmin=0, vmax=1)
    I.blit(0,0,window.width,window.height)
    window.draw()
    updateStreet(state)
    time += args[0]
    frames += 1
    if time-initial_time > 5.0:
      fps = float(frames)/(time-initial_time)
      print 'FPS: %.2f (%d frames in %.2f seconds)' % (fps, frames,
                                                        time-initial_time)
      frames,initial_time=0, time
  window.mainloop()
#
## for glumpy 2.1
#def Real_Time_Visualizator(leftmost_lane, num_of_lanes, numRoadPiece,
#                            width, height):
#  global state, time, initial_time, frames, previous_state
#  time, initial_time, frames = 0,0,0
#  state = leftmost_lane
#  window = glumpy.figure((width, height))
#
#  @window.event
#  def on_mouse_press(x, y, LEFT):
#    global state
#    # randObstacle(state)
#    slowing_perturbation(state)
#
#  @window.event
#  def on_idle(*dt):
#    global state, time, initial_time, frames
#    window.clear()
#    V = visual_position(state,num_of_lanes,numRoadPiece,width,height)
#    I = glumpy.image.Image(V,colormap=glumpy.colormap.Hot,vmin=0,vmax=1)
#    I.draw(0,0,0, window.width, window.height)
#    updateStreet(leftmost_lane)
#    I.update()
#    window.redraw()
#    time += dt[0]
```

```
#     frames += 1
#     if time-initial_time > 5.0:
#       fps = float(frames)/(time-initial_time)
#       print ('FPS: %.2f (%d frames in %.2f seconds)'
#              % (fps, frames, time-initial_time))
#     frames,initial_time=0, time
#   glumpy.show()
#


"""random functions to randomly initialize the cellular automata"""

def random_kind_array(kinds_distribution, numVeh):
    # from a set of kinds of cars(in vehicle_class)
    # it creates a random array of kinds of cars
    # distributed uniformly depending on the second input
    # of set_of_kinds (ex. set_of_kinds=[(test1,2),(test2,4)]
    # 1/3 is distributed as test1 and the other is distrubuted with density 2/3
    # the slowest vehicle is the leading one
    (first_kind, first_percentage) = kinds_distribution[0]
    slowest = first_kind
    for (kind, perc) in kinds_distribution:
        if kind.maxV < slowest.maxV and perc is not 0:
            slowest = kind
    kind_array = [slowest]*numVeh
    Sum=0
    for (kind, percentage) in kinds_distribution:
        Sum += percentage
    for i in range(numVeh-1):
        rand = random.choice(range(Sum))
        rand += 1
        scan=0
        for (kind, percentage) in kinds_distribution:
            scan += percentage
            if rand <= scan:
                kind_array[i] = kind
                break
    return(kind_array)

def random_position_array(kind_array, maxDistance, minDistance, numVeh):
  # gives an array of random positions of cars (uniformly distributed)
  # where the space between cars is randomly distrib. between minDistance
  # and maxDistance
  position_distribution = numpy.zeros(numVeh).astype(numpy.float32)
```

```
  position_distribution[0]=kind_array[0].length
  for i in range(1,numVeh):
    position_distribution[i] = (position_distribution[i-1] +
                                kind_array[i-1].length +
                                minDistance + random.random() *
                                (maxDistance - minDistance) +
                                kind_array[i].length)
  return(position_distribution)

def random_velocity_array(kind_array, numVeh, v_min, v_max):
  # it gives an array of random velocities (uniformly distributed)
  # between minimum velocity v_min and maximum velocity v_max
  velocity_distribution = numpy.zeros(numVeh).astype(numpy.float32)
  for i in range(numVeh):
    maximumVelocity = min(v_max, kind_array[i].maxV)
    minimumVelocity = min(v_min, kind_array[i].maxV)
    velocity_distribution[i] = (minimumVelocity +
                                random.random()*(maximumVelocity -
                                                 minimumVelocity))
  return(velocity_distribution)

def random_stress(kind_array, numVeh, min_stress, max_stress):
  stressArray = numpy.zeros(numVeh).astype(numpy.float32)
  for i in range(numVeh):
    maximum_stress = max(max_stress, kind_array[i].maxstress)
    minimum_stress = min(min_stress, kind_array[i].maxstress)
    stressArray[i] = min_stress + random.random() * (maximum_stress -
                                                     minimum_stress)
  return(stress)



"""I/O function for file storage """

def saveData(data, nameFile):
  # it store the data into a file named nameFile
  file = open(nameFile, 'wb')
  pickle.dump(data, file)
  file.close()

def loadData(nameFile):
  # it loads the file and returns the set of files saved on the file
  file = open(nameFile, 'rb')
  data = pickle.load(file)
  file.close()
```

```
    return(data)

### Plain cars
plain = vehicles('plain car',                                   # name
                              0,                                # color
                              28,                               # optimal velocity
                              36,                               # max speed
                              2.0,                              # length
                              [[0,1],[3,1],[5,0]],              # PVS
                              [[3,0],[5,1],[7,0]],              # PS
                              [[5,0],[7,1],[9,0]],              # PN
                              [[-10000,1],[-0.001,1],[0,0],[7,0],[9,1],[10000,1]],
                                   # PB
                              [[0,1],[3,1],[5,0]],              # NVS
                              [[0,1],[10,1],[15,0]],             # VS front
                              [[10,0],[25,1],[40,0]],           # S front
                              [[25,0],[50,1],[80,0]],           # N
                              [[50,0],[90,1],[100000,1]],       # B
                              [[0,1],[5,1],[10,0]],             # S back
                              [[0,1],[10,1],[14,0]],            # S jam velocity
                              [[-0.30,0],[0,1],[0.20,0]],       # VS acc
                              [[0,0],[1.6,1],[3.1,0]],          # PS acc
                              [[1.6,0],[3.1,1],[4.6,0]],        # P acc
                              [[3.1,0],[4.6,1],[6.7,0]],        # PB acc
                              [[-5.0,0],[-3.3,1],[0,0]],        # NS acc
                              [[-6.7,0],[-5.0,1],[-3.3,0]],     # N acc
                              [[-8.4,0],[-6.7,1],[-5.0,0]],     # NB acc
                              0.2,                              # noise
                              '3*V',                            # consumption
                              500,                              # max stress
                              -450,                             # min stress,
                              lambda x:x,                       # LCRP
                              lambda x:x,                       # LCLP
                              # not used but settable for instance (45-x)/45)
                              lambda x: 1)                      # engine
                                   efficience


### Plain trucks
long = vehicles('long vehicle',                                 # name
                              0.15,                             # color
                              20,                               # optimal velocity
                              25,                               # max speed
                              4.5,                              # length
                              [[0,1],[5,1],[7,0]],              # PVS
```

```
                        [[5,0],[7,1],[9,0]],                   # PS
                        [[7,0],[9,1],[11,0]],                  # PN
                        [[-10000,1],[-0.001,1],[0,0],[9,0],[11,1],[10000,1]],
                            # PB
                        [[0,1],[1,1],[2,0]],                   # NVS
                        [[0,1],[20,1],[30,0]],                 # VS front
                        [[20,0],[40,1],[60,0]],                # S front
                        [[40,0],[70,1],[100,0]],               # N
                        [[70,0],[110,1],[100000,1]],           # B
                        [[0,1],[5,1],[10,0]],                  # S back
                        [[0,1],[8,1],[12,0]],                  # S jam velocity
                        [[-0.40,0],[0,1],[0.10,0]],            # VS acc
                        [[0,0],[0.9,1],[1.8,0]],               # PS acc
                        [[0.9,0],[1.8,1],[2.7,0]],             # P acc
                        [[1.8,0],[2.7,1],[3.6,0]],             # PB acc
                        [[-2.9,0],[-1.9,1],[0,0]],             # NS acc
                        [[-3.9,0],[-2.9,1],[-1.9,0]],          # N acc
                        [[-4.9,0],[-3.9,1],[-2.9,0]],          # NB acc
                        0.1,                                   # noise
                        '5*V',                                 # consumption
                        300,                                   # max stress
                        -700,                                  # min stress ,
                        lambda x:x,                            # LCRP
                        lambda x:x**1.25,                      # LCLP
                        lambda x: 1)                           # engine
                            efficience


### a possible modelization of a nervous-sportive driver
anxi = vehicles('anxiuos vehicle',                             # name
                        0.20,                                  # color
                        30,                                    # optimal velocity
                        36,                                    # max speed
                        1.7,                                   # length
                        [[0,1],[3,1],[4,0]],                   # PVS
                        [[3,0],[4,1],[6,0]],                   # PS
                        [[4,0],[6,1],[8,0]],                   # PN
                        [[-10000,1],[-0.001,1],[0,0],[8,0],[10,1],[10000,1]],
                           # PB
                        [[0,1],[2,1],[3,0]],                   # NVS
                        [[0,1],[8,1],[12,0]],                  # VS front
                        [[8,0],[20,1],[30,0]],                 # S front
                        [[20,0],[40,1],[70,0]],                # N
                        [[40,0],[80,1],[100000,1]],            # B
                        [[0,1],[5,1],[10,0]],                  # S back
```

```
                            [[0,1],[12,1],[16,0]],             # S jam velocity
                            [[-0.20,0],[0,1],[0.30,0]],        # VS acc
                            [[0,0],[1.8,1],[3.5,0]],           # PS acc
                            [[1.8,0],[3.5,1],[5.5,0]],         # P acc
                            [[3.5,0],[5.5,1],[7,0]],           # PB acc
                            [[-6,0],[-4.0,1],[0,0]],           # NS acc
                            [[-8.0,0],[-6,0],[-4,0]],          # N acc
                            [[-9.0,0],[-8.0,1],[-6,0]],        # NB acc
                            0.30,                              # noise
                            '3*V',                             # consumption
                            800,                               # max stress
                            -350,                              # min stress ,
                            lambda x:x,                        # LCRP
                            lambda x:x,                        # LCLP
                            # not used but settable for instance (45-x)/45)
                            lambda x: 1)                       # engine efficience


## Example of the real time simulator
#kindDistribution = [(plain, 70),(long,10),(anxi,20)]
#numVeh = 300
#maxAvDistance = 60
#minAvDistance = 60
#v_min = 35
#v_max = 35
#numRoadPiece = 15 # number of pieces you want to divide the screen
#width = 1500       # width of the window
#height = 750       # height of the window
#numLanes = 3
#
#randKind=random_kind_array(kindDistribution, numVeh)
#initPosition=random_position_array(randKind,
#                                    maxAvDistance, minAvDistance, numVeh)
#initVelocity=random_velocity_array(randKind, numVeh, v_min, v_max)
#l=initial_lane(randKind, initPosition, initVelocity)
##leftMostLane = createStreet(l, numLanes)
#leftMostLane = createRandHighway(None, 22000, numLanes, 0.5,
#                                   kindDistribution, 25, 2, 2)
#Real_Time_Visualizator(leftMostLane, numLanes, numRoadPiece, width, height)


#  The main function for running some experiments
#  Scenarios:  Length of the road
```

```
#                 Number of lanes
#                 Number of Iterations
#                 Number of repetions of the same experiment
#                 Emission rate (average vehicles per seconds entering)
#                 Influence radius (if it is negative the Tollbooth is not visible
#                                     hence the throughput is unlimited, the number
#                                     of vehicles processed by the Tollbooth is
#                                     infinite)
#                 Obstacle -1 it puts an obstacle at L/2 on the leftmost lane
#                           1 it puts an obstacle at L/2 on the rightmost lane
if __name__=='__main__':
  Length = float(sys.argv[1])
  numLanes = int(sys.argv[2])
  Iterations = int(sys.argv[3])
  NumExp = int(sys.argv[4])
  EmissionRate = float(sys.argv[5])/numLanes
  MaxDensityLongVeh = 4
  if len(sys.argv) is 6:
    InfluenceRadius = 25
    titleRadius = ''
    Obstacle = 0
    obstacleTitle = ''
  if len(sys.argv) is 7:
    InfluenceRadius = float(sys.argv[6])
    titleRadius = '-' + sys.argv[6]
    Obstacle = 0
    obstacleTitle = ''
  if len(sys.argv) is 8:
     #-1 put an obstacle at Length/2 on the most left lane
     #+1 put an obstacle at Length/2 on the most right lane
    Obstacle = int(sys.argv[7])
    InfluenceRadius = float(sys.argv[6])
    titleRadius = '-' + sys.argv[6]
    obstacleTitle = '-' + sys.argv[7]
  T = numpy.zeros((MaxDensityLongVeh, NumExp, Iterations))
  L = numpy.zeros((MaxDensityLongVeh, NumExp, Iterations))
  AV = numpy.zeros((MaxDensityLongVeh, NumExp, Iterations))
  AD = numpy.zeros((MaxDensityLongVeh, NumExp, Iterations))
  N = numpy.zeros((MaxDensityLongVeh, NumExp, Iterations))
  F = numpy.zeros((MaxDensityLongVeh, NumExp, Iterations))
  for k in range(MaxDensityLongVeh):
    for j in range(NumExp):
```

```
      # distribution of long vehicles with step 10: 0, 10, 20, 30
      distr = 10 * k
      kindDistribution = [(plain, 100 - distr),(long, distr)]
      leftMostLane = createStreet(None, numLanes)
      createOnToll(leftMostLane, 0, EmissionRate, kindDistribution)
      createOffToll(leftMostLane, Length, 1, InfluenceRadius, 100, 10)
      if Obstacle is -1:
        # it creates an obstacle in position Length/2
        leftMostLane.createObstacle(Length/2, Length/5, 0.45)
      if Obstacle is 1:
        rightMostLane = leftMostLane.returnLane(numLanes - 1)
        rightMostLane.createObstacle(Length/2, Length/5, 0.45)
      for i in range(Iterations):
        (through, lat, avVel, avDist, num) =  uptdateStreet(leftMostLane)
        T[k][j][i] = through
        L[k][j][i] = lat
        AV[k][j][i] = avVel
        AD[k][j][i] = avDist
        N[k][j][i] = num
  saveData(T, 'Throughput' + sys.argv[1] + '-' + sys.argv[2] +
            '-' + sys.argv[3] + '-' + sys.argv[4] + '-' + sys.argv[5]
            + titleRadius + obstacleTitle)
  saveData(L, 'Latency'+sys.argv[1] + '-'+sys.argv[2] +
            '-' + sys.argv[3] + '-' + sys.argv[4] + '-' + sys.argv[5]
            + titleRadius + obstacleTitle)
  saveData(AV, 'AvVelocity' + sys.argv[1] + '-' + sys.argv[2] +
            '-' + sys.argv[3] + '-' + sys.argv[4] + '-' + sys.argv[5]
            + titleRadius + obstacleTitle)
  saveData(AD, 'AvDistance'+ sys.argv[1] + '-' + sys.argv[2] +
            '-' + sys.argv[3] + '-' + sys.argv[4] + '-' + sys.argv[5]
            + titleRadius + obstacleTitle)
  saveData(N/Length, 'Density' + sys.argv[1] + '-' + sys.argv[2] +
            '-' + sys.argv[3] + '-' + sys.argv[4] + '-' + sys.argv[5]
            + titleRadius + obstacleTitle)
```

ozsim.py

Appendix B

The Implementation with PyCuda

B.1 Cuda and PyCuda: An Overview

Cuda (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. Cuda is the computing engine in NVIDIA Graphics Processing Units (GPUs) that is accessible to software developers through variants of industry standard programming languages. It enables dramatic increases in computing performance by controlling the power of the GPU.

PyCuda is an open-source toolkit that supports GPU run-time code generation for high performance computing and was developed to access NVIDIA's Cuda parallel computation application programming interface (API) from Python.

B.2 PyCuda Code of the Simulator

```
from __future__ import division
import sys, os
sys.path.append(os.getcwd())
import numpy
import random
import string
import math
import time
import copy
import pickle
```

```
# glumpy is requested only by Real_Time_Visualizer
import glumpy
# pycuda
import pycuda.autoinit
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray
from pycuda.curandom import rand as curand
from pycuda.compiler import SourceModule
import time

global Normalized, ThreadsPerBlock, unitTime

unitTime = 1
# number of threads for each block
ThreadsPerBlock = 128
# 1 second of simulation every 1 seconds
Normalized = False





"""classes definitions"""

class vehicles():
  def __init__(self, name = None, color = None, optV = None,
               maxV = None, length = None, FctVS = None,
               FctS = None, FctM = None, FctB = None,
               BctVS = None, FdVS = None, FdS = None, FdM = None,
               FdB = None, BdVS = None, VelS = None,
               accZ = None, accPS = None,
               accPM = None, accPB = None,
               accNS = None, accNM = None,
               accNB = None, accNoise = None, maxstress = None,
               minstress = None, LCRP = None, LCLP = None, engEff = None):
    self.name = name
    self.color = color
    self.optV = optV
    self.maxV = maxV
    self.length = length
    self.FctVS = FctVS
    self.FctS = FctS
    self.FctM = FctM
    self.FctB = FctB
    self.BctVS = BctVS
```

```
    self.FdVS = FdVS
    self.FdS = FdS
    self.FdM = FdM
    self.FdB = FdB
    self.BdVS = BdVS
    self.VelS = VelS
    self.accZ = accZ
    self.accPS = accPS
    self.accPM = accPM
    self.accPB = accPB
    self.accNS = accNS
    self.accNM = accNM
    self.accNB = accNB
    self.accNoise = accNoise
    self.maxstress = maxstress
    self.minstress = minstress
    self.LCRP = LCRP
    self.LCLP = LCLP
    self.engEff = engEff

class SetOfVehicles():
  def __init__(self, list = None):
    self.list = list

  def toCuda(self):
    # transform the membership function of the set vehicles in a suitable
    # matrix to pass to the cuda interface
    maxlength = 1
    for (kind, perc) in self.list:
      if perc is not 0:
        for (Att, Value) in kind.__dict__.iteritems():
          if Att in MembershipFunctions:
            maxlength = max(maxlength, len(Value))
    CudaMembFunctions = numpy.zeros(2 * maxlength).astype(numpy.float32)
    CudaMembFunctions[0] = maxlength
    CudaMembFunctions[1] = 2 * maxlength * len(MembershipFunctions)
    CudaProp =  numpy.array((len(Properties))).astype(numpy.float32)
    for (kind, perc) in self.list:
      if perc is not 0:
        for att in MembershipFunctions:
          for (Att, Value) in kind.__dict__.iteritems():
            if Att is att:
              length = len(Value)
              Dom = numpy.zeros(maxlength).astype(numpy.float32)
              Cod = numpy.zeros(maxlength).astype(numpy.float32)
```

```
              for i in range(maxlength):
                if i < length:
                  Dom[i] = Value[i][0]
                  Cod[i] = Value[i][1]
                else:
                  Dom[i] = Dom[i - 1] + 1
                  Cod[i] = 0
              CudaMembFunctions = numpy.hstack((CudaMembFunctions, Dom,
                  Cod))
        for att in Properties:
          for (Att, Value) in kind.__dict__.iteritems():
            if Att is att:
              CudaProp = numpy.hstack((CudaProp, numpy.array((Value)).
                  astype(numpy.float32)))
    self.CudaMembFunct = CudaMembFunctions
    self.CudaProperties = CudaProp

  def posOfVeh(self, kindVeh):
    i = 0
    for (kind, perc) in self.list:
      if perc is not 0:
        if kind is kindVeh:
          return i
        else:
          i += 1

class cars(object):
  def __init__(self, position = None, velocity = None,
                kind = None, stress = 0, color = None):
    self.position = position
    self.velocity = velocity
    self.kind = kind
    self.stress = stress
    # if checked, it has just changed lane
    self.alreadyDone = False
    # external object or "not a vehicle mode" False
    self.extObj = False
    # the car is visible to the others
    self.visibility = True
    # internal timer set to zero
    self.timer = 0
    self.color = self.kind.color

  def evalFeelings(self, lane):
    # nstress is always positive
```

```
    if self.stress >= 0:
      if self.kind.maxstress is 0:
        nstress = 0
      else:
        nstress = self.stress/self.kind.maxstress
      if random.random() < self.kind.LCRP(nstress):
        return lane.right
    else:
      if self.kind.minstress is 0:
        nstress = 0
      else:
        nstress = (self.stress/self.kind.minstress)
      if random.random() < self.kind.LCLP(nstress):
        # if you are in a jam situation try to change lane
        # to get out from the jam
        if random.random() < lin(self.velocity, self.kind.VelS):
          if lane.right is None:
            return lane.left
          if lane.left is None:
            return lane.right
          if random.random() < 0.7:
            return lane.left
          else:
            return lane.right
          # uncomment to reduce to decrease the ping-pong effect
          # if it is commented then the system seems more reactive
          # in a jam situation
          #self.stress = 0
        else:
          return lane.left
    return None

class external(cars):
  def __init__(self, position = None, kind = None, visibility = None,
               emissionRate = None, kindDistribution = None,
               initialVelocity = None, absorptionProb = None,
               influenceRadius = None, probe = False, bufferCapacity = None):
    self.velocity = 0
    self.position = position
    self.kind = kind
    # external object or ”not a vehicle mode” True
    self.extObj = True
    # the car is visible to the others
    self.visibility = False
```

```
    self.emissionRate = emissionRate
    self.kindDistribution = kindDistribution
    self.initialVelocity = 12
    self.absorptionProb = absorptionProb
    self.influenceRadius = influenceRadius
    # loop detector modality off
    self.probe = probe
    self.counter = 0
    self.buffer = []
    self.bufferCapacity =  bufferCapacity

  def obstacle(self, dimension, color):
    obstacle = vehicles()
    obstacle.length = dimension
    obstacle.color = color
    obstacle.name = 'Obstacle'
    self.kind = obstacle
    self.visibility = True

  def onRamp(self, emissionRate, kindDistribution, bufferCapacity, color):
    ramp = vehicles()
    ramp.length = 3
    ramp.color = color
    ramp.name = 'On Ramp'
    self.kind = ramp
    self.visibility = False
    self.emissionRate = emissionRate
    self.kindDistribution = kindDistribution
    self.bufferCapacity = bufferCapacity

  def offRamp(self, absorptionProb, influenceRadius, bufferCapacity,
              sampRate, color):
    ramp = vehicles()
    ramp.length = 3
    ramp.color = color
    ramp.name = 'Off Ramp'
    self.kind = ramp
    self.absorptionProb = absorptionProb
    self.influenceRadius = influenceRadius
    self.bufferCapacity = bufferCapacity
    self.sampRate = sampRate
    self.avLatency = None
    self.throughput = None
    # if influence radius is negative then make it invisible
    # (the vehicles do not slow down in the proximy)
```

```
    if influenceRadius < 0:
      self.visibility = False
    else:
      self.visibility = True

  def loopDetector(self, influenceRadius, color):
    ramp = vehicles()
    ramp.length = 1
    ramp.color = color
    ramp.name = 'Loop Detector'
    self.kind = ramp
    self.visibility = False
    self.influenceRadius = influenceRadius
    self.probe = True

class lane(object):
  def __init__(self, ilist = None, left = None, right = None, SetOfVeh = None):
    self.left = left
    self.right = right
    self.SetOfVeh = SetOfVeh
    self.initContents(ilist)

  def initContents(self, ilist = None):
    if ilist is not None:
      List = ilist
    else:
      List = []
    dummy=vehicles()
    dummy.name = 'Dummy'
    dummy.maxV = 40
    dummy.length = 0
    first_dummy = cars(-10, 0, dummy)
    first_dummy.extObj = True
    List.insert(0, first_dummy)
    self.first_dummy = first_dummy
    if ilist is not None:
      length = len(ilist)
      last_vehicle = ilist[length - 1]
      last_vehicle_position = last_vehicle.position
    else:
      last_vehicle_position = 0
    position_dummy = last_vehicle_position + 1000
    last_first_dummy = cars(position_dummy, dummy.maxV, dummy)
    last_second_dummy = cars(position_dummy + 10, dummy.maxV, dummy)
```

```
    last_first_dummy.extObj = True
    last_second_dummy.extObj = True
    List.append(last_first_dummy)
    List.append(last_second_dummy)
    self.last_first_dummy = last_first_dummy
    self.last_second_dummy = last_second_dummy
    self.contents = List
    CudaInt = numpy.zeros(4 * len(self.contents)).astype(numpy.float32)
    for i, car in enumerate(List):
      CudaInt[4 * i] = car.position
      CudaInt[4 * i + 1] = car.velocity
      CudaInt[4 * i + 2] = car.stress
      if car.kind is not dummy:
        CudaInt[4 * i + 3] = self.SetOfVeh.posOfVeh(car.kind)
      else:
        # a negative value for the kind is reserved for external objects
        CudaInt[4 * i + 3] = -1
    self.CudaInterface = CudaInt
    self.CudaBuffer = numpy.zeros_like(CudaInt).astype(numpy.float32)
    self.CudaNumThreads = len(List)

  def setLeft(self, left):
    self.left = left

  def getLeft(self):
    return(self.left)

  def setRight(self, right):
    self.right = right

  def getRight(self):
    return(self.right)

  def delete(self, car):
    Index = self.contents.index(car)
    del self.contents[Index]
    CudaIndx = int(4 * Index)
    self.CudaInterface = numpy.delete(self.CudaInterface, [CudaIndx,
        CudaIndx + 1,
                                              CudaIndx + 2, CudaIndx + 3])
    self.CudaBuffer = numpy.delete(self.CudaBuffer, [CudaIndx, CudaIndx +
        1,
                                              CudaIndx + 2, CudaIndx + 3])
    self.CudaNumThreads -= 1
```

```
  def evalChanges(self):
    for c in self.contents:
      if not c.extObj:
        if not c.alreadyDone:
          newLane = c.evalFeelings(self)
          if newLane is not None:
            if self.transfer(c, newLane):
              self.delete(c)
      elif (c.kind is not self.first_dummy.kind):
        self.evalExternal(c)

  def TransferFromCuda(self):
    # used to update the lane using the information
    # obtained by the CUDA TransFunction
    for i, car in enumerate(self.contents):
      # update only the objects which are not external
      if (not car.extObj or car.kind is self.first_dummy.kind):
        car.position = self.CudaInterface[4 * i]
        car.velocity = self.CudaInterface[4 * i + 1]
        car.stress = self.CudaInterface[4 * i + 2]
        car.alreadyDone = False
        #car.timer += 1

  def GlobalTransition(self):
    # the random array for the evaluation of the stress
    # of each vehicles
    Number = self.CudaNumThreads
    NumOfBlocks = Number//ThreadsPerBlock + 1
    Rand = numpy.random.random(Number).astype(numpy.float32)
    N = numpy.array((Number)).astype(numpy.int32)
    TransFunction(drv.In(Rand), drv.In(self.CudaInterface),
                  drv.Out(self.CudaBuffer), drv.In(N),
                  block=(ThreadsPerBlock,1,1), grid=(NumOfBlocks,1))
    B = self.CudaInterface
    self.CudaInterface = self.CudaBuffer
    self.CudaBuffer = B
    self.TransferFromCuda()

  def transfer(self, car, tolane):
    if tolane is not None:
      i = tolane._possibleCar(car)
      if i is not None:
        car.alreadyDone = True
        # sense of satisfaction in changing lane
        car.stress /= 5
```

```
        # cuda interface as to be changed
        CudaIndx = int(4 * i)
        tolane.CudaInterface = numpy.insert(tolane.CudaInterface,
                     [CudaIndx, CudaIndx, CudaIndx, CudaIndx],
                     [car.position, car.velocity, car.stress,
                      tolane.SetOfVeh.posOfVeh(car.kind)])
        tolane.CudaBuffer = numpy.insert(tolane.CudaBuffer,
                     [CudaIndx, CudaIndx, CudaIndx, CudaIndx],
                     [0, 0, 0, 0])
        tolane.CudaNumThreads += 1
        tolane.contents.insert(i,car)
        return True
    return False

  def _possibleCar(self, car):
    precIndx, indx = self._index(car)
    if indx is not None:
        x = (car.position - self.contents[precIndx].position - car.kind.
            length -
        self.contents[precIndx].kind.length)
        y = (self.contents[indx].position - car.position - car.kind.length
            -
        self.contents[indx].kind.length)
        if car.velocity < 0:
          print "Alert!", car.velocity, car.position, car.kind.name
        if ((x > (self.contents[precIndx].velocity)**1.2 - car.velocity +
              abs(self.contents[precIndx].velocity - car.velocity) + 3)
                  and
            (3 + (car.velocity)**1.25 - self.contents[indx].velocity < y )
                ):
          return indx
    else:
      return None

  def _index(self, car):
    # it returns the indices of the closest
    # front and back visible cars
    Length = len(self.contents)
    Interval = [0, Length - 1]
    while True:
      MedIndx = (Interval[0] + Interval[1])//2
      leftcar = self.contents[Interval[0]]
      midcar = self.contents[MedIndx]
      rightcar = self.contents[Interval[1]]
      if (leftcar.position <= car.position <= midcar.position):
```

```
        Interval = [Interval[0], MedIndx]
      else:
        Interval = [MedIndx, Interval[1]]
      if (Interval[1] == Interval[0] + 1):
        break
    if not car.extObj:
      front_position = car.position + car.kind.length
      back_position = car.position - car.kind.length
      # check for the closest back and front visible vehicles
      frontcar = self.contents[Interval[1]]
      backcar = self.contents[Interval[0]]
      while (not frontcar.visibility and Interval[1] < Length):
        Interval[1] += 1
        frontcar = self.contents[Interval[1]]
      while (not backcar.visibility and Interval[0] >= 0):
        Interval[0] -= 1
        backtcar = self.contents[Interval[0]]
      if (front_position <= frontcar.position - frontcar.kind.length and
          back_position >= backcar.position - backcar.kind.length):
        return Interval
      else:
        return (None, None)
    else:
      return Interval

  def insertExternal(self, ext):
    # it insets an external object in a lane
    Index = self._index(ext)[1]
    if Index is not None:
      self.contents.insert(Index, ext)
      # cuda interface as to be changed
      CudaIndx = int(4 * Index)
      if ext.visibility:
        self.CudaInterface = numpy.insert(self.CudaInterface,
                                          [CudaIndx, CudaIndx, CudaIndx,
                                              CudaIndx],
                                          [ext.position, 0, ext.kind.
                                              length, -1])
        self.CudaBuffer = numpy.insert(self.CudaBuffer,
                                       [CudaIndx, CudaIndx, CudaIndx,
                                           CudaIndx],
                                       [ext.position, 0, ext.kind.length,
                                           -1])
      else:
        self.CudaInterface = numpy.insert(self.CudaInterface,
```

```
                                       [CudaIndx, CudaIndx, CudaIndx,
                                         CudaIndx],
                                       [-1, 0, 0, -1])
        self.CudaBuffer = numpy.insert(self.CudaBuffer,
                                       [CudaIndx, CudaIndx, CudaIndx,
                                         CudaIndx],
                                       [-1, 0, 0, -1])
      self.CudaNumThreads += 1

  def evalExternal(self, ext):
    # update the position (as index) in the lane
    self.delete(ext)
    self.insertExternal(ext)
    # if it is an emitter
    if ext.emissionRate is not None and ext.kindDistribution is not None:
      if len(ext.buffer) < ext.bufferCapacity:
        pos = ext.position
        rate = ext.emissionRate
        # using a poisson distribution we calculate the probability
        # of having at least one occurrence of a vehicle in the interval
        # [0,ext.counter + 1]
        prob = 1 - (math.exp(-(rate * (ext.counter + 1) * unitTime)))
        # if the random test succeed, randomly choose a kind of vehicle
        # distributed as kindDistribution
        if random.random() <= prob:
          Sum=0
          for (kind, percentage) in ext.kindDistribution.list:
            Sum += percentage
          rand = random.choice(range(Sum))
          rand += 1
          scan=0
          for (kind, percentage) in ext.kindDistribution.list:
            scan += percentage
            if rand <= scan:
              chosenVehicle = kind
              break
          newCar = cars(pos, ext.initialVelocity, chosenVehicle)
          ext.buffer.append(newCar)
          ext.counter = 0
        else:
          ext.counter += 1
        if ext.buffer != []:
          i = self._possibleCar(ext.buffer[0])
          if i is not None:
            ext.buffer[0].alreadyDone = True
```

```
            self.transfer(ext.buffer[0], self)
            del ext.buffer[0]
      # if the buffer is full do not create any other vehicles
      # but instead just put the vehicles already in the buffer
      else:
        if ext.buffer != []:
          i = self._possibleCar(ext.buffer[0])
          if i is not None:
            ext.buffer[0].alreadyDone = True
            self.transfer(ext.buffer[0], self)
            del ext.buffer[0]
    # if it is a sink
    if ext.absorptionProb is not None and ext.influenceRadius is not None:
      if len(ext.buffer) <= ext.bufferCapacity:
        # this case is used to simulate open road tolling system
        # in the off-toll plaza
        if ext.influenceRadius < 0:
          influencePosition = (ext.position - ext.kind.length - 50)
          for c in self.contents:
            if ((influencePosition <= c.position + c.kind.length)
              and not c.extObj):
              # if you grab it then store it in the buffer
              if random.random() <= ext.absorptionProb:
                if ext.bufferCapacity != 0:
                  ext.buffer.append(c)
                  self.delete(c)
                else:
                  # if bufferCapacity is 0 then the capacity
                  # of the buffer is infinite
                  self.delete(c)
              else:
                carPosition = c.position
                c.position = carPosition + ext.influenceRadius + ext.kind.
                    length
                # otherwise teleport it beyond the ramp
                if self.transfer(c, self):
                  self.delete(c)
                # if you cannot do it leave it there
                else:
                  c.position = carPosition
        else:
          influencePosition = (ext.position - ext.kind.length
                               - ext.influenceRadius)
          for c in self.contents:
```

```
            if ((influencePosition <= c.position + c.kind.length <= ext.
                position
                - ext.kind.length) and not c.extObj):
              # if you grab it then store it in the buffer
              if random.random() <= ext.absorptionProb:
                if ext.bufferCapacity != 0:
                  ext.buffer.append(c)
                  self.delete(c)
                else:
                  # if bufferCapacity is 0 then the capacity
                  # of the buffer is infinite
                  self.delete(c)
              else:
                carPosition = c.position
                c.position = carPosition + ext.influenceRadius + ext.kind.
                    length
                # otherwise teleport it beyond the ramp
                if self.transfer(c, self):
                  self.delete(c)
                # if you cannot do it leave it there
                else:
                  c.position = carPosition
      else:
        # not able to store in the buffer
        return False
      if ext.sampRate is not None:
        if ext.counter is ext.sampRate:
          T = 0
          for c in ext.buffer:
            T += c.timer
          if ext.buffer != []:
            ext.avLatency = T/len(ext.buffer)
            ext.throughput = len(ext.buffer)
          ext.buffer = []
          ext.counter = 0
        else:
          ext.counter += 1
    # if it is a loop detector
    if ext.probe:
      influencePosition = ext.position - ext.kind.length - ext.
          influenceRadius
      # saving the vehicles within the influence radius
      j = newIndex - 1
      newList = []
      while 0 < j < newIndex:
```

```
        if not self.contents[j].extObj:
          if (influencePosition <= self.contents[j].position +
              self.contents[j].kind.length <= ext.position - ext.kind.length):
            newList.append(self.contents[j])
            j -= 1
          else:
            break
        else:
          j -= 1
      missingVehicles = len([x for x in ext.buffer if x not in newList])
      ext.counter += missingVehicles
      ext.buffer = newList

  def returnLane(self, num):
    # given a number and lane it returns
    # the lane of distance num from self
    lane = self
    if num >= 0:
      for i in range(num):
        lane = lane.getRight()
      return lane
    if num < 0:
      for i in range(num):
        lane = lane.getLeft()
      return lane

  def RandomInit(self, kinds_distribution, numVeh, max_distance, min_distance,
                 v_min, v_max):
    # it randomly initializes a lane
    for (first_kind, first_percentage) in kinds_distribution.list:
      if first_percentage is not 0:
        slowest = first_kind
    for (kind, perc) in kinds_distribution.list:
      if kind.maxV < slowest.maxV and perc is not 0:
        slowest = kind
    kind_array = [slowest]*numVeh
    Sum=0
    for (kind, percentage) in kinds_distribution.list:
        Sum += percentage
    for i in range(numVeh-1):
        rand = random.choice(range(Sum))
        rand += 1
        scan=0
```

```
        for (kind, percentage) in kinds_distribution.list:
            scan += percentage
            if rand <= scan:
                kind_array[i] = kind
                break
    position_distribution = numpy.zeros(numVeh).astype(numpy.float32)
    position_distribution[0] = kind_array[0].length
    for i in range(1,numVeh):
      position_distribution[i] = (position_distribution[i-1] +
                                  kind_array[i-1].length +
                                  min_distance + random.random() *
                                  (max_distance - min_distance) +
                                  kind_array[i].length)
    velocity_distribution = numpy.zeros(numVeh).astype(numpy.float32)
    for i in range(numVeh):
      maximumVelocity = min(v_max, kind_array[i].maxV)
      minimumVelocity = min(v_min, kind_array[i].maxV)
      velocity_distribution[i] = (minimumVelocity +
                                  random.random()*(maximumVelocity -
                                                   minimumVelocity))
    state = []
    for j in range(numVeh):
      car=cars(position_distribution[j], velocity_distribution[j],
               kind_array[j], 0)
      state.append(car)
    self.initContents(state)

class Street():
  def __init__(self, SetOfVeh = None, SetOfLanes = None, NumLanes = None):
    self.SetOfVeh = SetOfVeh
    self.NumLanes = NumLanes
    if SetOfLanes is None:
      rightLane = lane(None, None, None, SetOfVeh)
    else:
      # the first element of the list is the rightmost lane
      rightLane = SetOfLanes[0]
    self.RightMostLane = rightLane
    leftLane = rightLane
    for j in range(NumLanes - 1):
      if ( (SetOfLanes is not None) and (j + 1) < len(SetOfLanes) ):
        leftLane = SetOfLanes[j + 1]
        leftLane.setRight(rightLane)
        rightLane.setLeft(leftLane)
      else:
        leftLane = lane(None, None, rightLane, SetOfVeh)
```

```
        rightLane.setLeft(leftLane)
      rightLane=leftLane
    self.LeftMostLane = leftLane
    self.UpdatePosDummies()

  def UpdatePosDummies(self):
    # used to have the dummies always as the last elements even when we add
    # something
    MaxDistance = 0
    Lane = self.RightMostLane
    while Lane is not None:
      MaxDistance = max(MaxDistance, Lane.last_first_dummy.position)
      Lane = Lane.getLeft()
    Lane = self.RightMostLane
    while Lane is not None:
      Lane.last_first_dummy.position = MaxDistance
      Lane.last_second_dummy.position = MaxDistance + 10
      # update the cuda interface
      I = Lane.CudaNumThreads
      Lane.CudaInterface[4 * (I - 1)] = Lane.last_second_dummy.position
      Lane.CudaInterface[4 * (I - 2)] = Lane.last_first_dummy.position
      Lane = Lane.getLeft()

  def GlobalTransitionStreet(self):
    # updating of the multilane model, the update
    # is done from left to right (the leftmost has the precedence)
    lane = self.LeftMostLane
    while lane is not None:
      lane.evalChanges()
      lane = lane.getRight()
    lane = self.LeftMostLane
    # it is possible to pararelize in CUDA this part
    while lane is not None:
      lane.GlobalTransition()
      lane = lane.getRight()



  def slowing_perturbation(self):
    # it slows down the first car in each lane
    lane = self.LeftMostLane
    while lane is not None:
      l = len(lane.contents)
      indexVehicles = range(l)
      indexVehicles.reverse()
```

```
      for j in indexVehicles:
        if not lane.contents[j].extObj:
          lane.CudaInterface[4 * j + 1] /= 5
          break
      lane = lane.getRight()

  def createObstacle(self, lane = None, pos = None,
                     dimension = 200, color = 0.45):
    if lane is None:
      lane = self.RightMostLane
    if pos is None:
      l = len(lane.contents)
      indexVehicles = range(l)
      indexVehicles.reverse()
      for j in indexVehicles:
        if not lane.contents[j].extObj:
          pos = lane.contents[j].position + 200
          break
    if (pos + 2 * dimension) >= lane.last_first_dummy.position:
      lane.last_first_dummy.position = pos + 2 * dimension + 10
      lane.last_second_dummy.position = pos + 2 * dimension + 20
      # update the cuda interface
      self.UpdatePosDummies()
    obst = external(pos + dimension)
    obst.obstacle(dimension, color)
    lane.insertExternal(obst)

  def randObstacle(self):
    # it creates a random obstacle in front of the first vehicle
    randIndx = random.choice(range(self.NumLanes))
    obstLane = self.LeftMostLane.returnLane(randIndx)
    self.createObstacle(obstLane)

  def createOnRamp(self, lane = None, pos = None, emissionRate = None,
                   kindDistribution = None, bufferCapacity = 100, color =
                      0.33):
    if lane is None:
      lane = self.RightMostLane
    l = lane.CudaNumThreads
    if pos is None:
      pos = cont[l - 3].position + 200
    if pos >= lane.last_first_dummy.position:
      lane.last_first_dummy.position = pos + 1000
      lane.last_second_dummy.position = pos + 1010
      self.UpdatePosDummies()
```

```
    OnRamp = external(pos)
    OnRamp.onRamp(emissionRate, kindDistribution, bufferCapacity, color)
    lane.insertExternal(OnRamp)

  def createOffRamp(self, lane = None, pos = None, absorptionProb = None,
                    influenceRadius = None, bufferCapacity = 0, sampRate = None,
                    color = 0.5):
    if lane is None:
      lane = self.RightMostLane
    l = lane.CudaNumThreads
    if pos is None:
      pos = cont[l - 3].position + 200
    if pos >= lane.last_first_dummy.position:
      lane.last_first_dummy.position = pos + 1000
      lane.last_second_dummy.position = pos + 1010
      self.UpdatePosDummies()
    if (sampRate is not None and bufferCapacity is not None
        and bufferCapacity < sampRate):
      bufferCapacity = sampRate
    OffRamp = external(pos)
    OffRamp.offRamp(absorptionProb, influenceRadius, bufferCapacity,
                    sampRate, color)
    lane.insertExternal(OffRamp)


  def createLoopDetector(self, lane = None, pos = None,
                         influenceRadius = 36, color = 0):
    if lane is None:
      lane = self.RightMostLane
    l = lane.CudaNumThreads
    if pos is None:
      pos = cont[l - 3].position + 200
    if pos >= lane.last_first_dummy.position:
      lane.last_first_dummy.position = pos + 1000
      lane.last_second_dummy.position = pos + 1010
      self.UpdatePosDummies()
    loopDet = external(pos)
    loopDet.loopDetector(influenceRadius, color)
    lane.insertExternal(OffRamp)

  def createOnToll(self, position, emissionRate, kindDistribution):
    lane = self.LeftMostLane
    while lane is not None:
      self.createOnRamp(lane, position, emissionRate, kindDistribution)
```

```
      lane = lane.getRight()

  def createOffToll(self, position, absorptionProb, influenceRadius,
                    bufferCapacity, sampRate):
    lane = self.LeftMostLane
    while lane is not None:
      self.createOffRamp(lane, position, absorptionProb, influenceRadius,
                          bufferCapacity, sampRate)
      lane = lane.getRight()




class FuzzyModule():
  def __init__(self, LinguisticTerms = None, Inputs = None):
    self.LinguisticTerms = LinguisticTerms
    self.Inputs = Inputs
    self.SetOfRules = []

  def AddRule(self, antecendent, consequent):
    # an antecedent of the form [["x", "A"], ["y", "B"]] means x is A and
        y is B
    # ["x", "notA"] means x is not A
    # a consequent is of the form ["C"] since acceleration is implicit
    self.SetOfRules.append([antecendent, consequent])

  def toCuda(self):
    numOfRules = len(self.SetOfRules)
    numOfAntColumns = 0
    AntRules = numpy.empty(0).astype(numpy.int32)
    ConsRule =  numpy.empty(0).astype(numpy.int32)
    PhiFactor = numpy.empty(0).astype(numpy.int32)
    for rule in self.SetOfRules:
      numOfAntColumns = max(numOfAntColumns, len(rule[0]))
    for rule in self.SetOfRules:
      ConsIndx = self.LinguisticTerms.index(rule[1][0]) + 1
      ConsRule = numpy.hstack( (ConsRule, numpy.array((ConsIndx)).astype(
          numpy.int32)) )
      for ant in rule[0]:
        if (ant[1].startswith("not")):
          AntIndx = self.LinguisticTerms.index(ant[1][3:]) + 1
          VarIndx = self.Inputs.index(ant[0]) + 1
          AntRules = numpy.hstack((AntRules, numpy.array((AntIndx)).astype
              (numpy.int32)))
```

```
          AntRules = numpy.hstack((AntRules, numpy.array((-VarIndx)).astype(numpy.int32)))
        else:
          AntIndx = self.LinguisticTerms.index(ant[1]) + 1
          VarIndx = self.Inputs.index(ant[0]) + 1
          AntRules = numpy.hstack((AntRules, numpy.array((AntIndx)).astype(numpy.int32)))
          AntRules = numpy.hstack((AntRules, numpy.array((VarIndx)).astype(numpy.int32)))
      for i in range(numOfAntColumns - len(rule[0])):
        AntRules = numpy.hstack((AntRules, numpy.array((0)).astype(numpy.int32)))
        AntRules = numpy.hstack((AntRules, numpy.array((0)).astype(numpy.int32)))
    for ling in PhiParmeter:
      PhiIndx = self.LinguisticTerms.index(ling) + 1
      PhiFactor = numpy.hstack( (PhiFactor, numpy.array((PhiIndx)).astype(numpy.int32)) )
    self.CudaAntRules = AntRules
    self.CudaConsRules = ConsRule
    self.PhiFactor = PhiFactor
    self.CudaNAntColumns = numpy.array((numOfAntColumns)).astype(numpy.int32)
    self.CudaNRules = numpy.array((numOfRules)).astype(numpy.int32)




"""Some utility functions"""

def lin(input, function):
  # this function returns the value of the scattered function
  length = len(function)
  for i in range(length):
    if input < function[0][0]:
      return 0
    elif input > function[length-1][0]:
      return 0
    elif function[i][0] <= input <= function[i+1][0]:
      # it finds the position of the input
      position = i
      break
  # linear approximation
  x = function[position][0]
  y = function[position + 1][0]
```

```
  f_x = function[position][1]
  f_y = function[position + 1][1]
  if x == y:
    if f_x != f_y:
      print('it is not a function')
    else:
      return(f_x)
  slope=(f_x - f_y)/(x-y)
  return(f_y + slope*(input-y))


def createRandHighway(length, numLanes, emissRate,
                      kindDistribution, infRadius, numObstacles, numRamps)
                         :
  # a simple highway random generator
  initialPosition = 2
  Str = Street(kindDistribution, None, numLanes)
  Str.createOnToll(initialPosition, emissRate, kindDistribution)
  # 13 meters of influence radius 40 cars of buffer capacity 40 seconds
  # of sample rate
  Str.createOffToll(length + initialPosition, 1, infRadius, 100, 10)
  position = initialPosition
  maxSpaceOffOnRamp = 200
  maxInflunceRadius = 30
  if numRamps is 0:
    interval = length
  else:
    interval = length/numRamps
  for i in range(numRamps):
    SpaceOffOnRamp = maxSpaceOffOnRamp * random.random()
    position += random.random() * (interval/2) + (interval/2)
    Str.createOffRamp(Str.RightMostLane, position, random.random(),
                              30 + maxInflunceRadius * random.random())
    Str.createOnRamp(Str.RightMostLane, position + 200 + random.random() *
                             maxSpaceOffOnRamp, random.random(),
                             kindDistribution)
  maxDimObstacle = 70
  position = initialPosition
  if numObstacles is 0:
    interval = length
  else:
    interval = length/numObstacles
  for i in range(numObstacles):
    dimObstacle = 30 + maxDimObstacle*random.random()
```

```
    position += ( random.random() * (interval/2) + (interval/2) +
        dimObstacle )
    choosenIndx = random.choice(range(numLanes))
    lane = Str.LeftMostLane.returnLane(choosenIndx)
    Str.createObstacle(lane, position, dimObstacle)
  return Str

""" visualization functions """

def draw_car(car, index_of_lane, numRoadPiece, height, width, matrix,
             dimension_of_road, separation_width, visual_separation,
             one_lane_width):
  # the index of the lane the leftmost is 0
  # it draws a car inside a matrix
  position = car.position
  kind_of_car = car.kind
  back_position = position - kind_of_car.length
  if car.extObj:
    color = kind_of_car.color
  else:
    color = car.color
  vehicle_width = one_lane_width-2*visual_separation
  wrap_factor = (back_position//width)%numRoadPiece
  y = (wrap_factor * dimension_of_road + separation_width + index_of_lane
     *
        one_lane_width + visual_separation)
  # it calculate the position of the car module the border of the screen
  x = round(back_position%width)
  if wrap_factor == numRoadPiece-1:
    # the last raw case
    if (x + 2 * kind_of_car.length) <= width:
      # it does not go outside the screen
      for i in range(int(2*kind_of_car.length)):
        for j in range(int(vehicle_width)):
          matrix[int(y)+j][int(x)+i]=color
    else:
      # otherwise draw the car one piece on this raw and we wrapp the
          right
      # down corner with the left up corner note that the simulation has
          not
      # closed boundaries, however for this is done for a matter of
      # visualization
      for i in range(int(width-x)):
        for j in range(int(vehicle_width)):
          matrix[int(y)+j][int(x)+i]=color
```

```
      for i in range(int(2*kind_of_car.length-(width-x))):
        for j in range(int(vehicle_width)):
          matrix[separation_width + index_of_lane * one_lane_width +
                 visual_separation + j][i] = color
  else:
    if (x+2*kind_of_car.length)<=width:
      # it does not go outside the screen
      for i in range(int(2*kind_of_car.length)):
        for j in range(int(vehicle_width)):
          matrix[int(y)+j][int(x) + i]=color
    else:
      # otherwise draw the car one piece on this raw
      for i in range(int(width-x)):
        for j in range(int(vehicle_width)):
          matrix[int(y)+j][int(x)+i]=color
      for i in range(int(2*kind_of_car.length - (width-x))):
        # the other piece in the next raw
        for j in range(int(vehicle_width)):
          matrix[int(y+dimension_of_road)+j][i]=color
  return(matrix)

def visual_position(leftmost_lane, numLanes, numRoadPiece,
                    width, height):
  # given a state of the CA it returns an array of the positions of the
      cars
  #  to be represented
  dimension_of_road = height//numRoadPiece
  separation_width = dimension_of_road//4
  one_lane_width = (dimension_of_road - separation_width)//(numLanes)
  visual_separation = one_lane_width//8
  street_matrix = numpy.ones((height,width)).astype(numpy.float32)
  # it initializes the matrix corresponding to the representation of the
  # street it draws the separation (in black)
  for i in range(numRoadPiece):
    for j in range(int(separation_width)):
      for l in range(int(width)):
        street_matrix[i*int(dimension_of_road)+j][l]=0
  if numRoadPiece*int(dimension_of_road)+int(separation_width)<=height:
    for j in range(int(separation_width)):
      for l in range(int(width)):
        street_matrix[numRoadPiece*int(dimension_of_road)+j][l] = 0
  for i in range(numRoadPiece):
    for j in range(1,numLanes):
      for k in range(int(visual_separation)):
        for l in range(int(width)):
```

```
          street_matrix[int(dimension_of_road)*i
                        + int(separation_width) + int(one_lane_width)*j
                        - int(visual_separation//2)+k][l] = 0
  lane = leftmost_lane
  index_of_lane = 0
  visual_matrix = street_matrix
  while lane != None:
    for c in lane.contents:
      if c.kind is not lane.first_dummy.kind:
        # don't draw the dummies
        vehicle_kind = c.kind
        visual_matrix = draw_car(c, index_of_lane, numRoadPiece, height,
                                 width, visual_matrix, dimension_of_road,
                                 separation_width, visual_separation,
                                 one_lane_width)
    lane=lane.getRight()
    index_of_lane += 1
  return(visual_matrix)

# glumpy 1.1
def Real_Time_Visualizator(street, num_of_lanes, numRoadPiece,
                           width, height):
  global time, initial_time, frames
  time, initial_time, frames = 0,0,0

  window = glumpy.Window(width, height)

  @window.event
  def on_mouse_press(x, y, LEFT):
    street.slowing_perturbation()
    #street.randObstacle()


  @window.event
  def on_idle(*args):
    global time, initial_time, frames, clock
    clock = 0
    window.clear()
    V = visual_position(street.LeftMostLane, num_of_lanes, numRoadPiece,
        width, height)
    I = glumpy.Image(V, cmap=glumpy.colormap.Hot, vmin=0, vmax=1)
    I.blit(0, 0, window.width,window.height)
    window.draw()
    time += args[0]
    clock += args[0]
```

```
    if Normalized:
      if clock == 1:
        street.GlobalTransitionStreet()
        clock = 0
        frames += 1
    else:
      street.GlobalTransitionStreet()
      frames += 1
    if time-initial_time > 5.0:
      fps = float(frames)/(time-initial_time)
      print 'FPS: %.2f (%d frames in %.2f seconds)' % (fps, frames,
                                                          time-initial_time)
      frames, initial_time = 0, time
  window.mainloop()



code = """

#define DIM_MEMB_FUNC $DIM_MEMB_FUNC
#define DIM_PROPERTIES $DIM_PROPERTIES
#define DIM_FIRST_ANT_RULE $DIM_FIRST_ANT_RULE
#define DIM_FIRST_CONS_RULE $DIM_FIRST_CONS_RULE
#define DIM_SECOND_ANT_RULE $DIM_SECOND_ANT_RULE
#define DIM_SECOND_CONS_RULE $DIM_SECOND_CONS_RULE
#define HALFNUM_COLUMNS_FIRST $HALFNUM_COLUMNS_FIRST
#define HALFNUM_COLUMNS_SECOND $HALFNUM_COLUMNS_SECOND
#define NUM_RULES_FIRST $NUM_RULES_FIRST
#define NUM_RULES_SECOND $NUM_RULES_SECOND
#define DIM_PHI_FACTORS $DIM_PHI_FACTORS


__device__ __constant__ float MembFunctions[DIM_MEMB_FUNC];
__device__ __constant__ float Properties[DIM_PROPERTIES];
__device__ __constant__ int FirstAntRules[DIM_FIRST_ANT_RULE];
__device__ __constant__ int FirstConsRules[DIM_FIRST_CONS_RULE];
__device__ __constant__ int SecondAntRules[DIM_SECOND_ANT_RULE];
__device__ __constant__ int SecondConsRules[DIM_SECOND_CONS_RULE];
__device__ __constant__ int PhiFactor[DIM_PHI_FACTORS];


__device__ void SumPre(float Inp, float *function, int dimension, float *
    Out) {
   if ( Inp == 0 ) {
      Out[0] = 0;
```

```
                      Out[1] = 1;
    }
    else {
        for(int i=0; i < (dimension - 1); ++i) {
            float slopeInv;
            if ( (function[i + dimension] == function[i + 1 + dimension]) && (Inp == function[i + dimension]) ) {
                Out[0] += function[i + dimension];
                                           Out[1] += 1;
            }
            if ( ((function[i + 1 + dimension] < Inp) && (Inp <= function[i + dimension])) || ((function[i + dimension] <= Inp) && (Inp < function[i + 1 + dimension])) ) {
                slopeInv = (function[i + 1] - function[i])/(function[i + 1 + dimension] - function[i + dimension]);
                Out[0] += ( (function[i] + (Inp - function[i + dimension]) * slopeInv) );
                                           Out[1] += 1;
            }
        }
        if ( function[dimension - 1 + dimension] == Inp ) {
            Out[0] +=  function[dimension - 1 + dimension];
                                Out[1] += 1;
        }
    }
};

__device__ float LinEval(float Inp, float *function , int dimension) {
           float Out;
           if ( (Inp < function[0]) || (Inp > function[dimension - 1]) ) {
           Out = 0;
           return Out;
    }
           else if ( Inp == function[dimension - 1] ) {
                      Out = function[dimension - 1 + dimension];
                      return Out;
    }
    else {
        int x;
        int y;
        for(int i=0; i< (dimension - 1); ++i) {
            if ( (function[i] <= Inp) && (Inp <= function[i + 1]) ) {
                x = i;
                y = i+1;
                break;
```

```
            }
        };
        float f_x = function[x + dimension];
        float f_y = function[y + dimension];
        float slope;
        slope = (f_y - f_x)/(function[y] - function[x]);
        Out = (f_x + slope * (Inp - function[x]));
                      return Out;
        }
};

__device__ float FuzzyEval( float *Inp, float *MembershipF, int DomDim,
     int *AntRules, int *ConsRule, int numOfColumnsRules, int numberOfRules
      ) {
           float Num = 0;
           float Den = 0;
           int FunctionLength = 2 * (DomDim);
           for(int i=0; i < numberOfRules; ++i) {
                      int numOfCons = ConsRule[i];
                      float weight = 1;
                      for(int j=0; j < numOfColumnsRules; ++j) {
                                 if (AntRules[2 * j + i * (2 * numOfColumnsRules)]
                                     > 0) {
                                            int numOfFunction = AntRules[2 * j + i *
                                                 (2 * numOfColumnsRules)];
                                            int numOfVariable = AntRules[2 * j + i *
                                                 (2 * numOfColumnsRules) + 1];
                                            if ( numOfVariable < 0 ) {
                                                       weight = min( weight, abs(1 -
                                                            LinEval( Inp[-numOfVariable -
                                                            1], (MembershipF +
                                                            numOfFunction * FunctionLength
                                                            ), DomDim) ) );
                                            }
                                            else {
                                                       weight = min( weight, LinEval(Inp[
                                                            numOfVariable - 1], (
                                                            MembershipF + numOfFunction *
                                                            FunctionLength), DomDim) ) ;
                                            }
                                 }
                      };
                      float Out[] = {0,0};
                      SumPre(weight, (MembershipF + numOfCons * FunctionLength),
                            DomDim, Out);
```

```
                Num += (weight * Out[0]);
                Den += (weight * Out[1]);
        };
   if (Den != 0) {
      return Num/Den;
   }
   else {
      return Den;
   }
};

__global__ void TransFunction(
float *Rand,
float *source,
float *output,
int *NumOfThreads
) {
  int DomDim = (int) MembFunctions[0];
  int KindMembOffset = (int) MembFunctions[1];
  int KindPropOffset = (int) Properties[0];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  float delta_plus;
  float tau_plus;
  float PerceivedFct;
  float delta_minus;
  float tau_minus;
  float delta_next;
  float tau_next;
  float worstCollTime;
  if (tid == 0) {
    output[4 * (tid)] = source[4 * (tid)];
    output[4 * (tid) + 1] = source[4 * (tid) + 1];
    output[4 * (tid) + 2] = source[4 * (tid) + 2];
    output[4 * (tid) + 3] = source[4 * (tid) + 3];
  }
  else if ( (tid >= *NumOfThreads - 2) && (tid < *NumOfThreads) ) {
    output[4 * (tid)] = source[4 * (tid)] + source[4 * (tid) + 1];
    output[4 * (tid) + 1] = source[4 * (tid) + 1];
    output[4 * (tid) + 2] = source[4 * (tid) + 2];
    output[4 * (tid) + 3] = source[4 * (tid) + 3];
  }
  else if (tid < *NumOfThreads) {
    int myKindIndx = (int) source[4 * tid + 3];
    if (myKindIndx >= 0) {
      int VisibleFrontIndex = 1;
```

```
    while (source[4 * (tid + VisibleFrontIndex)] < 0 ) {
      VisibleFrontIndex += 1;
    }
    int VisibleNextIndex = VisibleFrontIndex + 1;
    while (source[4 * (tid + VisibleNextIndex)] < 0 ) {
      VisibleNextIndex += 1;
    }
    int VisibleBackIndex = -1;
    while (source[4 * (tid + VisibleBackIndex)] < 0 ) {
      VisibleBackIndex -= 1;
    }
    float myHalfLength =  Properties[myKindIndx * KindPropOffset + 3];
    int frontKindIndx = (int) source[4 * (tid + VisibleFrontIndex) + 3];
    int nextKindIndx = (int) source[4 * (tid + VisibleNextIndex) + 3];
    int backKindIndx = (int) source[4 * (tid + VisibleBackIndex) + 3];
    float frontHalfLength;
    if (frontKindIndx < 0) {
      frontHalfLength = source[4 * (tid + VisibleFrontIndex) + 2];
    }
    else {
      frontHalfLength = Properties[frontKindIndx * KindPropOffset + 3];
    }
    float nextHalfLength;
    if (nextKindIndx < 0) {
      nextHalfLength = source[4 * (tid + VisibleNextIndex) + 2];
    }
    else {
      nextHalfLength = Properties[nextKindIndx * KindPropOffset + 3];
    }
    float backHalfLength;
    if (backKindIndx < 0) {
      backHalfLength = source[4 * (tid + VisibleBackIndex) + 2];
    }
    else {
      backHalfLength = Properties[backKindIndx * KindPropOffset + 3];
    }
    float V_max = Properties[myKindIndx * KindPropOffset + 2];
    float V_opt = Properties[myKindIndx * KindPropOffset + 1];
    float MaxStress = Properties[myKindIndx * KindPropOffset + 4];
    float MinStress = Properties[myKindIndx * KindPropOffset + 5];
    delta_plus = source[4 * (tid + VisibleFrontIndex)] - source[4 * (tid
        )] - myHalfLength - frontHalfLength;
    delta_minus = source[4 * (tid)] - source[4 * (tid + VisibleBackIndex
        )] - myHalfLength - backHalfLength;
```

```
    delta_next = source[4 * (tid + VisibleNextIndex)] - source[4 * (tid)] - myHalfLength - nextHalfLength;
    if (source[4 * (tid) + 1] == source[4 * (tid + VisibleFrontIndex) + 1]) {
      tau_plus = 999;
    }
    else {
      tau_plus = delta_plus/(source[4 * (tid) + 1] - source[4 * (tid + VisibleFrontIndex) + 1]);
    }
    if (source[4 * (tid) + 2] > 0) {
      float slowParameter = (MaxStress - source[4 * (tid) + 2])/(0.1 + source[4 * (tid) + 1]);
      if ( (tau_plus < 0) && (source[4 * (tid) + 2] >= (0.5 * MaxStress) ) ) {
        PerceivedFct = slowParameter;
      }
      else {
        PerceivedFct = min(tau_plus, slowParameter);
      }
    }
    else {
      PerceivedFct = tau_plus;
    }
    if (source[4 * (tid) + 1] == 0) {
      worstCollTime = 999;
    }
    else {
      worstCollTime = delta_plus/source[4 * (tid) + 1];
    }
    if (source[4 * (tid + VisibleBackIndex) + 1] == source[4 * (tid) + 1]) {
      tau_minus = 999;
    }
    else {
      tau_minus = delta_minus/(source[4 * (tid + VisibleBackIndex) + 1] - source[4 * (tid) + 1]);
    }
    if (source[4 * (tid) + 1] == source[4 * (tid + VisibleNextIndex) + 1]) {
      tau_next = 999;
    }
    else {
      tau_next = delta_next/(source[4 * (tid) + 1] - source[4 * (tid + VisibleNextIndex) + 1] );
```

```
    }
    float FirstInp[] = {delta_minus, tau_minus, delta_plus, PerceivedFct
        , worstCollTime, source[4 * (tid) + 1]};
    float FirstAcc = FuzzyEval( FirstInp, (MembFunctions +
        KindMembOffset * myKindIndx), DomDim, FirstAntRules,
        FirstConsRules, HALFNUM_COLUMNS_FIRST, NUM_RULES_FIRST );
    float SecondInp[] = {delta_next, tau_next};
    float SecondAcc = FuzzyEval( SecondInp, (MembFunctions +
        KindMembOffset * myKindIndx), DomDim, SecondAntRules,
        SecondConsRules, HALFNUM_COLUMNS_SECOND, NUM_RULES_SECOND );
    float DecidedAcc;
    if (FirstAcc <= 0) {
      DecidedAcc = min(FirstAcc, SecondAcc);
    }
    else if(SecondAcc < -0.25) {
      DecidedAcc = (FirstAcc + SecondAcc)/2;
    }
    else {
      DecidedAcc = FirstAcc;
    }
    float newVel;
    newVel = min( V_max, min(max(0.0, delta_plus), max(0.0, source[4 * (
        tid) + 1] + DecidedAcc)) );
    output[4 * (tid) + 1] = newVel;
    output[4 * (tid)] = source[4 * (tid)] + newVel;
    output[4 * (tid) + 2] = source[4 * (tid) + 2] + (source[4 * (tid) +
        1] - V_opt) * Rand[tid];
    output[4 * (tid) + 3] = source[4 * (tid) + 3];
    if (output[4 * (tid) + 2] > MaxStress) {
      output[4 * (tid) + 2] = MaxStress;
    }
    if (output[4 * (tid) + 2] < MinStress) {
      output[4 * (tid) + 2] = MinStress;
    }
    if ( (MinStress/2 < output[4 * (tid) + 2]) && (output[4 * (tid) + 2]
        < 0) ) {
      if (tau_plus < 0) {
        output[4 * (tid) + 2] /= 2;
      }
      else {
        float Coll[2];
        float Dist[2];
        Coll[0] = LinEval(tau_plus, (MembFunctions + KindMembOffset *
            myKindIndx + PhiFactor[0]), DomDim);
```

```
            Coll[1] = LinEval(tau_plus, (MembFunctions + KindMembOffset *
                myKindIndx + PhiFactor[1]), DomDim);
            Dist[0] = LinEval(delta_plus, (MembFunctions + KindMembOffset *
                myKindIndx + PhiFactor[2]), DomDim);
            Dist[1] = LinEval(delta_plus, (MembFunctions + KindMembOffset *
                myKindIndx + PhiFactor[3]), DomDim);
            float factor = 0;
            for(int i=0; i<2; ++i){
              for(int j=0; j<2; ++j){
                factor = max( factor, min(Coll[i], Dist[j]) );
              }
            }
            output[4 * (tid) + 2] *= (1 + factor);
          }
        }
      }
      else {
        output[4 * (tid)] = source[4 * (tid)];
        output[4 * (tid) + 1] = source[4 * (tid) + 1];
        output[4 * (tid) + 2] = source[4 * (tid) + 2];
        output[4 * (tid) + 3] = source[4 * (tid) + 3];
      }
    }
};

"""



### passenger vehicles
passenger = vehicles('passenger vehicle',                 # name
                    0.0,                                  # color
                    28,                                   # optimal velocity
                    36,                                   # max speed
                    2.0,                                  # length
                    [[0,1],[3,1],[5,0]],                  # FctVS
                    [[3,0],[5,1],[7,0]],                  # FctS
                    [[5,0],[7,1],[9,0]],                  # FctM
                    [[-10000,1],[-0.001,1],[0,0],[7,0],[9,1],[10000,1]],
                        # FctB
                    [[0,1],[3,1],[5,0]],                  # BctVS
                    [[0,1],[10,1],[15,0]],                # FdVS
                    [[10,0],[25,1],[40,0]],               # FdS
                    [[25,0],[50,1],[80,0]],               # FdM
                    [[50,0],[90,1],[100000,1]],           # FdB
```

```
                      [[0,1],[5,1],[10,0]],                  # S back
                      [[0,1],[10,1],[14,0]],                 # S jam velocity
                      [[-0.30,0],[0,1],[0.20,0]],            # VS acc
                      [[0,0],[1.6,1],[3.1,0]],               # PS acc
                      [[1.6,0],[3.1,1],[4.6,0]],             # PM acc
                      [[3.1,0],[4.6,1],[6.7,0]],             # PB acc
                      [[-5.0,0],[-3.3,1],[0,0]],             # NS acc
                      [[-6.7,0],[-5.0,1],[-3.3,0]],          # NM acc
                      [[-8.4,0],[-6.7,1],[-5.0,0]],          # NB acc
                      0.20,                                  # noise
                      500,                                   # max stress
                      -450,                                  # min stress ,
                      lambda x:x,                            # LCRP
                      lambda x:x)                            # LCLP

### plain trucks
long = vehicles('long vehicle',                              # name
                      0.15,                                  # color
                      20,                                    # optimal velocity
                      25,                                    # max speed
                      4.5,                                   # length
                      [[0,1],[5,1],[7,0]],                   # FctVS
                      [[5,0],[7,1],[9,0]],                   # FctS
                      [[7,0],[9,1],[11,0]],                  # FctM
                      [[-10000,1],[-0.001,1],[0,0],[9,0],[11,1],[10000,1]],
                          # FctB
                      [[0,1],[1,1],[2,0]],                   # BctVS
                      [[0,1],[20,1],[30,0]],                 # FdVS front
                      [[20,0],[40,1],[60,0]],                # FdS front
                      [[40,0],[70,1],[100,0]],               # FdM
                      [[70,0],[110,1],[100000,1]],           # FdB
                      [[0,1],[5,1],[10,0]],                  # S back
                      [[0,1],[8,1],[12,0]],                  # S jam velocity
                      [[-0.40,0],[0,1],[0.10,0]],            # VS acc
                      [[0,0],[0.9,1],[1.8,0]],               # PS acc
                      [[0.9,0],[1.8,1],[2.7,0]],             # P acc
                      [[1.8,0],[2.7,1],[3.6,0]],             # PB acc
                      [[-2.9,0],[-1.9,1],[0,0]],             # NS acc
                      [[-3.9,0],[-2.9,1],[-1.9,0]],          # N acc
                      [[-4.9,0],[-3.9,1],[-2.9,0]],          # NB acc
                      0.10,                                  # noise
                      300,                                   # max stress
                      -700,                                  # min stress ,
                      lambda x:x,                            # LCRP
                      lambda x:x**1.25)                      # LCLP
```

```
PhiParmeter = ["FctVS","FctS", "FdS", "FdM"]
MembershipFunctions = ["FctVS","FctS", "FctM", "FctB", "BctVS", "FdVS", "
    FdS", "FdM", "FdB",
                           "BdVS", "VelS", "accZ", "accPS", "accPM", "accPB",
                           "accNS", "accNM", "accNB"]
Properties = ["optV", "maxV", "length", "maxstress", "minstress"]
FirstInputs =["backDistance","backCollTime",
          "frontDistance","frontCollTime","worstCollTime","velocity"]
FirstFuzzy = FuzzyModule(MembershipFunctions, FirstInputs)
FirstFuzzy.AddRule([["frontCollTime", "FctB"], ["frontDistance", "FdB"],
  ["velocity", "notVelS"]],["accPM"])
FirstFuzzy.AddRule([["frontCollTime", "FctB"], ["frontDistance", "FdM"],
  ["velocity", "notVelS"]],["accPS"])
FirstFuzzy.AddRule([["frontCollTime", "FctB"], ["frontDistance", "FdS"]],[
    "accZ"])
FirstFuzzy.AddRule([["frontCollTime", "FctB"], ["frontDistance", "FdVS"
    ]],["accZ"])
FirstFuzzy.AddRule([["frontCollTime", "FctM"], ["frontDistance", "FdB"]],[
    "accZ"])
FirstFuzzy.AddRule([["frontCollTime", "FctM"], ["frontDistance", "FdM"]],[
    "accZ"])
FirstFuzzy.AddRule([["frontCollTime", "FctM"], ["frontDistance", "FdS"]],[
    "accNS"])
FirstFuzzy.AddRule([["frontCollTime", "FctM"], ["frontDistance", "FdVS"
    ]],["accNS"])
FirstFuzzy.AddRule([["frontCollTime", "FctS"], ["frontDistance", "FdB"]],[
    "accNM"])
FirstFuzzy.AddRule([["frontCollTime", "FctS"], ["frontDistance", "FdM"]],[
    "accNM"])
FirstFuzzy.AddRule([["frontCollTime", "FctS"], ["frontDistance", "FdS"]],[
    "accNM"])
FirstFuzzy.AddRule([["frontCollTime", "FctS"], ["frontDistance", "FdVS"
    ]],["accNM"])
FirstFuzzy.AddRule([["frontCollTime", "FctVS"], ["frontDistance", "FdB"
    ]],["accNB"])
FirstFuzzy.AddRule([["frontCollTime", "FctVS"], ["frontDistance", "FdM"
    ]],["accNB"])
FirstFuzzy.AddRule([["frontCollTime", "FctVS"], ["frontDistance", "FdS"
    ]],["accNB"])
FirstFuzzy.AddRule([["frontCollTime", "FctVS"], ["frontDistance", "FdVS"
    ]],["accNB"])
```

```
FirstFuzzy.AddRule([["backCollTime", "BctVS"], ["backDistance", "BdVS"], ["frontCollTime", "FctB"], ["frontDistance", "FdB"]],["accPS"])
FirstFuzzy.AddRule([["backCollTime", "BctVS"], ["backDistance", "BdVS"], ["frontCollTime", "FctB"], ["frontDistance", "FdM"]],["accPS"])
FirstFuzzy.AddRule([["backCollTime", "BctVS"], ["backDistance", "BdVS"], ["frontCollTime", "FctM"], ["frontDistance", "FdB"]],["accPS"])
FirstFuzzy.AddRule([["backCollTime", "BctVS"], ["backDistance", "BdVS"], ["frontCollTime", "FctM"], ["frontDistance", "FdM"]],["accPS"])
FirstFuzzy.AddRule([["frontCollTime", "FctB"], ["velocity", "VelS"]],["accPB"])
FirstFuzzy.AddRule([["worstCollTime", "FctVS"], ["frontDistance", "FdVS"]],["accNM"])
FirstFuzzy.AddRule([["worstCollTime", "FctVS"], ["frontDistance", "FdS"]],["accNM"])
FirstFuzzy.AddRule([["worstCollTime", "FctVS"], ["frontDistance", "FdM"]],["accNS"])

SecondInputs =["NextDistance","NextCollTime"]
SecondFuzzy = FuzzyModule(MembershipFunctions, SecondInputs)
SecondFuzzy.AddRule([["NextCollTime", "FctVS"], ["NextDistance", "FdVS"]],["accNB"])
SecondFuzzy.AddRule([["NextCollTime", "FctVS"], ["NextDistance", "FdS"]],["accNB"])
SecondFuzzy.AddRule([["NextCollTime", "FctVS"], ["NextDistance", "FdM"]],["accNB"])
SecondFuzzy.AddRule([["NextCollTime", "FctVS"], ["NextDistance", "FdB"]],["accNM"])
SecondFuzzy.AddRule([["NextCollTime", "FctS"], ["NextDistance", "FdVS"]],["accNM"])
SecondFuzzy.AddRule([["NextCollTime", "FctS"], ["NextDistance", "FdS"]],["accNM"])
SecondFuzzy.AddRule([["NextCollTime", "FctS"], ["NextDistance", "FdM"]],["accNS"])
SecondFuzzy.AddRule([["NextCollTime", "FctS"], ["NextDistance", "FdB"]],["accNS"])
SecondFuzzy.AddRule([["NextCollTime", "FctM"], ["NextDistance", "FdVS"]],["accNS"])
SecondFuzzy.AddRule([["NextCollTime", "FctB"], ["NextDistance", "FdVS"]],["accNS"])



kindDistribution = SetOfVehicles([(long, 20),(passenger, 80)])

def CudaInit(code, kindDistr, FirstFuModule, SecondFuModule):
    kindDistr.toCuda()
```

```
    FirstFuModule.toCuda()
    SecondFuModule.toCuda()
    code = string.Template(code)
    DIM_MEMB_FUNC = len(kindDistr.CudaMembFunct)
    DIM_PROPERTIES = len(kindDistr.CudaProperties)
    DIM_FIRST_ANT_RULE = len(FirstFuModule.CudaAntRules)
    DIM_FIRST_CONS_RULE = len(FirstFuModule.CudaConsRules)
    DIM_SECOND_ANT_RULE = len(SecondFuModule.CudaAntRules)
    DIM_SECOND_CONS_RULE = len(SecondFuModule.CudaConsRules)
    DIM_PHI_FACTORS = len(SecondFuModule.PhiFactor)
    HALFNUM_COLUMNS_FIRST = FirstFuModule.CudaNAntColumns
    HALFNUM_COLUMNS_SECOND = SecondFuModule.CudaNAntColumns
    NUM_RULES_FIRST = FirstFuModule.CudaNRules
    NUM_RULES_SECOND = SecondFuModule.CudaNRules
    Code = code.substitute(DIM_MEMB_FUNC = DIM_MEMB_FUNC,
                           DIM_PROPERTIES = DIM_PROPERTIES,
                           DIM_FIRST_ANT_RULE = DIM_FIRST_ANT_RULE,
                           DIM_FIRST_CONS_RULE = DIM_FIRST_CONS_RULE,
                           DIM_SECOND_ANT_RULE = DIM_SECOND_ANT_RULE,
                           DIM_SECOND_CONS_RULE = DIM_SECOND_CONS_RULE,
                           HALFNUM_COLUMNS_FIRST = HALFNUM_COLUMNS_FIRST,
                           HALFNUM_COLUMNS_SECOND = HALFNUM_COLUMNS_SECOND,
                           NUM_RULES_FIRST = NUM_RULES_FIRST,
                           NUM_RULES_SECOND = NUM_RULES_SECOND,
                           DIM_PHI_FACTORS = DIM_PHI_FACTORS)
    mod = SourceModule(Code)
    MembFunctions = mod.get_global("MembFunctions")[0]
    Properties = mod.get_global("Properties")[0]
    FirstAntRules = mod.get_global("FirstAntRules")[0]
    FirstConsRules = mod.get_global("FirstConsRules")[0]
    SecondAntRules = mod.get_global("SecondAntRules")[0]
    SecondConsRules = mod.get_global("SecondConsRules")[0]
    PhiFactor = mod.get_global("PhiFactor")[0]
    drv.memcpy_htod(MembFunctions,  kindDistribution.CudaMembFunct)
    drv.memcpy_htod(Properties,  kindDistribution.CudaProperties)
    drv.memcpy_htod(FirstAntRules,  FirstFuModule.CudaAntRules)
    drv.memcpy_htod(FirstConsRules,  FirstFuModule.CudaConsRules)
    drv.memcpy_htod(SecondAntRules,  SecondFuModule.CudaAntRules)
    drv.memcpy_htod(SecondConsRules,  SecondFuModule.CudaConsRules)
    drv.memcpy_htod(PhiFactor,  SecondFuModule.PhiFactor)
    return mod

Module = CudaInit(code, kindDistribution, FirstFuzzy, SecondFuzzy)
TransFunction = Module.get_function("TransFunction")
```

```


numVeh = 5000
maxAvDistance = 20
minAvDistance = 10
v_min = 10
v_max = 15
numRoadPiece = 20 # number of pieces you want to divide the screen
width = 1350       # width of the window
height = 650       # height of the window
numLanes = 3


Right = lane(None, None, None, kindDistribution)
Left = lane(None, None, None, kindDistribution)
LeftMost = lane(None, None, None, kindDistribution)

Right.RandomInit(kindDistribution, numVeh, maxAvDistance, minAvDistance, v_min, v_max)
Left.RandomInit(kindDistribution, numVeh, maxAvDistance, minAvDistance, v_min, v_max)
LeftMost.RandomInit(kindDistribution, numVeh, maxAvDistance, minAvDistance, v_min, v_max)

Str = Street(kindDistribution, [Right, Left, LeftMost], numLanes)
#Str.createOnRamp(None, 1000, 1, kindDistribution)
#Str.createOffRamp(None, 6000, 1, 25, 100, 10)
#Str = createRandHighway(20000, numLanes, 1.5, kindDistribution, 25, 4, 1)
#Real_Time_Visualizator(Str, numLanes, numRoadPiece, width, height)


t0 = time.clock()
for i in range(1000):
  Str.GlobalTransitionStreet()

print time.clock() - t0
```

cozsim.py

References

[1] A.J.R. Amaya, O. Lengerke, C.A. Cosenza, M.S. Dutra, and M.J.M. Tavera. **Comparison of defuzzification methods: Automatic control of temperature and flow inheat exchanger**. *Automation Control-Theory and Practice, InTech*, December 2009. 21

[2] A. Aw and M. Rascle. **Resurrection of second order models of traffic flow**. *SIAM Journal of Applied Mathematics*, **60(3)**:916–938, 2000. 7

[3] M. Bando, K. Hasebe, K. Nakanishi, A. Nakayama, A. Shibata, and Y. Sugiyama. **Phenomenological study of dynamical model of traffic flow**. *J. Phys. I France*, **5**:1389–1399, 1995. 6

[4] S.C. Benjamin, N.F. Johnson, and P.M. Hui. **Cellular automata models of traffic flow along a highway containing a junction**. *Journal of Physics A: Mathematical and General*, **29(12)**:3119–3127, 1996. 12

[5] E.R. Berlekamp, J.H. Conway, and R.K. Guy. **Winning Ways for Your Mathematical Plays II**. *Academic Press, NewYork*, 1982. 9

[6] V. Blue, F. Bonetto, and M. Embrechts. **A Cellular Automata of Vehicular Self Organization and Nonlinear Speed Transitions**. *Transportation Research Board Annual Meeting, Washington, DC,*,), 1996. 10

[7] M. Brackstone and M. McDonald. **Car-following: A historical review**. *Transportation Research Part F*, **2**:181–196, 1999. 5, 40, 98

[8] E. Van Broekhoven and B. De Baets. **Fast and accurate center of gravity defuzzification of fuzzy system outputs defined on trapezoidal fuzzy partitions**. *Fuzzy Sets and Systems*, **157**:904–918, 2006. 21

[9] E.G. Campari and G. Levi. **A realistic simulation for highway traffic by the use of cellular automata**. *Lecture Notes in Computer Science*, **2329**:763–772, 2002. 10, 32

[10] R.E. Chandler, R. Herman, and E.W. Montroll. **Traffic dynamics: Studies in car-following**. *Operations Research,INFORMS*, **6(2)**:165–184, 1958. 5

[11] G. Chen and T.T. Pham. **Introduction to fuzzy sets, fuzzy logic, and fuzzy systems**. *Boca Raton, FL: CRC Press*, 2001. 18

[12] D. Chowdhury, L. Santen, and A. Schadschneider. **Vehicular traffic: A system of interacting particles driven far from equilibrium**. *Current Science*, **77(411)**, 1999. 31

[13] A. Clarridge and K. Salomaa. **Analysis of a cellular automaton model for car traffic with a slow-to-stop rule**. *Theoretical Computer Science*, **411(38–39)**:3507–3515, 2010. 13

[14] O. Cordon, F. Herrera, F. Hoffmann, and L. Magdalena. **Genetic Fuzzy Systems: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases**. *In: Advances in Fuzzy Systems: Applications and Theory, World Scientific, Singapore*, 2001. 20

[15] J.A.M. Felippe de Souza, L. Schnitman, and T. Yoneyama. **A new Mamdani-like fuzzy structure**. *Proc. of the WSES Int. Conf. on Fuzzy Sets and Fuzzy Systems, Tenerife, Canary Islands, Spain*, pages 120–126, February 2001. 20

[16] S. Wolfram (Ed.). **Theory and Applications of Cellular Automata**. *World Scientific Press, Singapore*, 1986. 9, 25

[17] H. Emmerich and E. Rank. **An improved cellular automaton model for traffic flow simulation**. *Physica A: Statistical and Theoretical Physics*, **234(3–4)**:676–686, 1997. 10

[18] M. Errampalli, M. Okushima, and T. Akiyama. **Fuzzy logic based lane change model for microscopic traffic flow simulation**. *Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII)*, **12(2)**:172–181, 2008. 5, 40, 98

[19] M. Fukui and Y. Ishibashi. **Traffic flow in 1D cellular automaton model including cars moving with high speed**. *J. Phys. Soc. Japan*, **65(6)**:1868–1870, 1996. 10, 12

[20] M. Gardner. **Mathematical games**. *Sci. Amer.*, pages 223–225, 1970–71. 9

[21] Nikolas Geroliminis and Jie Sun. **Hysteresis phenomena of a macroscopic fundamental diagram in freeway networks**. *Transportation Research Part A*, **45**:966–979, 2011. 92

[22] B.D. Greenshields, D.Shapiro, and E.L Ericksen. **Traffic Performance at Urban Street Intersections**. *Technical Report, Bureau of Highway Traffic, Yale University, New Haven, CT*, **1**, 1947. 5

[23] I.D. Greenwood. **A New Approach to Estimate Congestion Impacts for Highway Evaluation-Effects on Fuel Consumption and Vehicle Emissions**. *PhD thesis, University of Auckland, New Zealand*, 2003. 35, 70, 71

[24] S.F. Hafstein, R. Chrobok, A. Pottmeier, M. Schreckenberg, and F.C. Mazur. **A high-resolution cellular automata traffic simulation model with application in a freeway traffic information system**. *Computer-Aided Civil and Infrastructure Engineering*, **19(5)**:338–350, 2004. 54

[25] S.F. Hafstein, R. Chrobok, A. Pottmeier, J. Wahle, and M. Schreckenberg. **Cellular automaton modeling of the autobahn traffic in North Rhine-Westphalia**. *Proceedings of the 4-th MATHMOD Vienna, 4-th IMCAS Symposium on Mathematical Modelling, ARGESIM Report no. 24, ed. I. Troch and F. Breitenecker, Vienna, Austria*, pages 1322–1331, 2003. 10, 32

[26] Dirk Helbing. **Traffic and related self-driven many-particle systems**. *Reviews of Modern Physics*, **73**:1067–1141, October 2001. 31

[27] R. Herman and R.B. Potts. **Single lane traffic theory and experiment**. *Proceedings Symposium on Theory of Traffic Flow. Ed. R. Herman, Elsevier Publications Co.*, pages 120–146, 1959. 5

[28] Serge P. Hoogendoorn and Piet H.L. Bovy. **State-of-the-art of vehicular traffic flow modelling**. *Proceedings of the Institution of Mechanical Engineers*, **215(1)**:283–303, 2001. 5

[29] J. Kari. **Theory of cellular automata: A survey**. *Theoretical Computer Science*, **334**:3–33, 2005. 23

[30] B.S. Kerner and H. Rehborn. **Experimental properties of phase transitions in traffic flow**. *Phys. Rev. Lett.*, **79**:4030–4033, 1997. 82, 92

[31] R. Klir and B. Yuan. **Fuzzy sets and fuzzy logic: Theory and applications**. *Prentice-Hall, PTR, Upper Saddle River, NJ*, 1995. 15

[32] W. Knospe, L. Santen, A. Schadschneider, and M. Schreckenberg. **Disorder effects in cellular automata for two-lane traffic**. *Physica A: Statistical and Theoretical Physics*, **265(3–4)**:614–633, 1999. 9, 55

[33] W. Knospe, L. Santen, A. Schadschneider, and M. Schreckenberg. **Towards a realistic microscopic description of highway traffic**. *Journal of Physics A*, **33**:477–485, 2000. 32, 85

[34] Myer Kutz. **Handbook of Transportation Engineering**. *Myer Kutz Assoc. Inc.*, pages 8.1–8.17, 2004. 77

[35] M.H. Lighthill and G.B. Whitham. **On kinematic waves II: A theory of traffic flow on long crowded roads**. *Proceedings of The Royal Society of London Ser.*, **A 229**:317–345, 1955. 6

[36] S. Maerivoet and B. De Moor. **Transportation Planning and Traffic Flow Models**. *Katholieke Universiteit Leuven, Department of Electrical Engineering ESAT-SCD (SISTA), Technical Report*, **05–155**, July 2005. 8

[37] S. Maerivoet and B. De Moor. **Cellular automata models of road traffic**. *Physics Reports*, **419(1)**:1–64, November 2005. 6, 9

[38] D. Makowiec and W. Miklaszewski. **Nagel-Schreckenberg model of traffic-Study of diversity of car rules**. *International Conference on Computational Science*, **3993**:256–263, 2006. 10

[39] E.H. Mamdani. **Applications of fuzzy algorithm for control a simple dynamic plant**. *Proceedings of the IEEE*, **121(12)**:1585–1588, 1974. 20

[40] E.H. Mamdani and S. Assilian. **An experiment in linguistic synthesis with a fuzzy logic controller**. *International Journal of ManMachine Studies*, **7(1)**:1–13, 1975. 20

[41] T. Nagatani. **Self-organization and phase transition in the traffic-flow model of a two-lane roadway**. *Journal of Physics A:Mathematical and General*, **26**:781–787, 1993. 9, 55

[42] K. Nagel. **Particle hopping models and traffc flow theory**. *Physical Review E*, **53(5)**:4655–4672, 1996. 9, 10

[43] K. Nagel and M. Schreckenberg. **A cellular automaton model for freeway traffic**. *Journal de Physique I*, **2(12)**:2221–2229, 1992. 10, 32, 34, 36

[44] K. Nagel, D. E. Wolf, P. Wagner, and P. Simon. **Two-lane traffic rules for cellular automata: A systematic approach**. *Phys. Rev. E*, **58(2)**:1425–1437, 1998. 9

[45] L. Neubert, L. Santen, A. Schadschneider, and M. Schreckenberg. **Single-vehicle data of highway trac: A statistical analysis**. *Phys. Rev. E*, **60**:6480–6490, 1999. 82, 85

[46] M.A. Pollatsche, A.Polus, and M. Livneh. **A decision model for gap acceptance and capacity at intersections**. *Transportation Research Part B*, **36**:649–663, 2002. 5

[47] P.Wagner. **Traffic simulations using cellular automata: Comparison with reality**. *Traffic and Granular Flow, World Scientific*, 1996. 10

[48] Kamini Rawat, Vinod Kumar Katiyar, and Pratibha Gupta. **Two-lane traffic flow simulation model via cellular automaton**. *International Journal of Vehicular Technology*, **2012**, 2012. 10, 32

[49] P.I. Richards. **Shock waves on the highway**. *Operation Research*, **4(1)**:42–51, 1956. 6

[50] M. Rickert, K. Nagel, M. Schreckenberg, and A. Latour. **Two lane traffic simulations using cellular automata**. *Physica A: Statistical and Theoretical Physics*, **231(4)**:534–550, 1996. 9, 55

[51] M. Sasvri and J. Kertsz. **Cellular automata models of single-lane traffic**. *Physical Review E*, **56(4)**:4104–4110, 1997. 10

[52] A. Schadschneider. **The Nagel-Schreckenberg model revisited**. *Eur. Phys. J. B*, **10**:573–582, 1999. 10

[53] A. Schadschneider and M. Schreckenberg. **Cellular automaton models and traffic flow**. *Journal of Physics A: Mathematical and General*, **26**:679–683, 1993. 10

[54] T. Toledo. **Driving behaviour: Models and challenges**. *Transport Reviews*, **27(1)**:65–84, January 2007. 5

[55] T. Toledo, C. Choudhury, and M. Ben-Akiva. **A lane-changing model with explicit target lane choice**. *Transportation Research Record*, **1934**:157–165, 2005. 5

[56] J. von Neumann. **The general and logical theory of automata**. *In: Cerebral Mechanisms in Behavior: The Hixon Symposium, L. A. Jeffress (Ed.), New York*, pages 1–41, 1948. 8

[57] P. Wagner, K. Nagel, and D.E. Wolf. **Realistic multi-lane traffic rules for cellular automata**. *Physica A: Statistical and Theoretical Physics*, **234(3–4)**:687–698, 1997. 9

[58] L. Wang, B.H. Wang, and B. Hu. **Cellular automaton traffic flow model between the Fukui-Ishibashi and Nagel-Schreckenberg models**. *Physical Review E*, **63(5)**:056117, 2001. 12

[59] Dietrich E. Wolf. **Cellular automata for traffic simulations**. *Physica A*, **263**:438–451, 1999. 6, 31, 85

[60] S. Wolfram. **Statistical mechanics of cellular automata**. *Rev. Mod. Phys.*, **55**:601–644, 1983. 9, 25

[61] S. Wolfram. **A New Kind of Science**. *Wolfram Media, Inc.*, 2002. 9, 10

[62] L.A. Zadeh. **Fuzzy sets**. *Informations and Control*, **8**:338–353, 1965. 15

[63] L.A. Zadeh. **Fuzzy logic**. *IEEE Computing Magazine*, **21(4)**:83–93, 1988. 21

[64] M. Zamith, R.C.P. Leal-Toledo, M. Kischinhevsky, E. Clua, D. Brando, A. Montenegro, and E.B. Lima. **A probabilistic cellular automata model for highway traffic simulation**. *Procedia Computer Science*, **1**:337–345, 2010. 10

[65] H.M. Zhang. **A non-equilibrium traffic model devoid of gas-like behavior**. *Transportation Research Part B*, **36**:275–290, 2002. 7