BACHELOR

Transient behavior of queues at signalized traffic intersections

van Riel, Jeroen

Award date:
2021

Transient Behavior of Queues at Signalized Traffic Intersections

Jeroen van Riel

July 2021

Bachelor Thesis
Applied Mathematics
Eindhoven University of Technology
Supervised by Dr. ir. M.A.A. Boon

Contents

Chapter 1

Introduction

Traffic congestion is one of the major problems of modern urban infrastructure, because it causes waste of energy, driver delays and pollution. Important first steps towards improvement are understanding and measuring the performance of existing road network, for which different kinds of models have been developed since the 1930s. Applications of these models may be roughly divided into two categories: design of infrastructure and dynamic control of traffic.

When building new infrastructure, traffic models are used to assess and compare different designs based on measures such as capacity and delay. We note that expanding road capacity does not always solve the problem of congestion and can even cause couterproductive results as shown by Ding & Song (2012). Models may also be used to understand the causes for congestion in existing road networks, which is important for maintenance.

The development of intelligent traffic control solutions, which often goes under the name of *smart mobility*, provides interesting alternatives to building roads. Existing traffic control devices include signs, traffic lights, ramp meters for highways and real-time traffic information on digital signs or via radio. This paper focuses on traffic lights, which provide a way of distributing the capacatiy of an intersection over multiple connected road segments in a safe and efficient way. This means that they have a direct impact on the performance of road networks, which makes it an interesting field of study. Assessing the performance of traffic lights can be done by studying the delays for drivers or the dynamics of the vehicle queues by using queueing models or simulations. Traffic engineers may use these models to calibrate green and red times for traffic lights with fixed settings. In combination with sensors, these models may form the basis of dynamic control algorithms that can, for example, enable green waves that respond to the current demand.

The rapid integration of autonomous driving and sensing technology in vehicles opens up many new possible approaches to traffic control. We give a short overview of some developments in the study of traffic control devices, which may be seen as two converging branches of research. On one hand there are innovations that make the infrastructure more responsive to the traffic. On the other hand, while modern vehicles already assist the driver heavily, this trend is heading towards the adoption of fully autonomous vehicles. The two branches meet in the field of infrastructure-to-vehicle (I2V) communication. There are simulation software packages that can be used to design wireless protocols for this kind of communication. One such simulation tool is Veins (Sommer *et al.* (2011)), which is in fact a combination of OMNeT++ (Varga (2010)), an event-based network simulator, and SUMO (Lopez *et al.* (2018)), a road traffic simulator. In very general terms, the network simulator models the temporal behavior of the communication and the road traffic simulation models the dynamic environment in which this communication must take place. Some of the issues that arise when implementing I2V are the placement of access points, signal blocking caused by buildings, and transmission delays. One may imagine an intersection that is only available for autonomous vehicles that are able to connect with the infrastructure. In such environment, Timmerman & Boon (2021) have shown that efficient scheduling algorithms can be constructed that let vehicles cross the intersection in platoons without requiring them to stop.

Dynamic traffic control is a popular field of study with exciting new challenges that is getting

more attention lately[1]. In order to study and optimize the performance of dynamic control systems, it is necessary to have accurate traffic models first. When we want to apply this model-based optimization, models that allow for fast numerical computation are preferred. Complex models that try to capture every detail of the behavior of traffic require a lot of computation time, which is often not available when real-time decisions need to be made. On the other hand, models that are too simplistic are not able to capture phenomena found in real situations, which may result in a loss of predictive power.

In this report, we will discuss some models of varying complexity for the queueing process of vehicles at a traffic lights with fixed control, so the length of the red and green phases are constant. We study the behavior of the length of the queue, with a particular focus on transient behavior. Many research papers only analyze the steady-state queues, but it may be argued that transient queue length is of particular importance for dynamic control. Being able to predict the queue length for each individual lane connected to a traffic light allows us to balance these lenghts by adjusting the settings of the traffic light program. When intersections are close to each other, it may happen that a long queue starts causing congestion or even blockage at the other intersection. This phenomenon is called *spillback* and may possibly be prevented by smart traffic lights.

The remainder of this thesis is organized as follows. Some history of the subject of traffic modeling in general is given in Chapter 2, along with some background on the development of microscopic simulation techniques and the role of car-following models. Section 2.4 gives a very brief introduction of mathematical queueing models and their basic assumptions. Chapter 3 presents the probabilistic model that has been proposed by Viti & van Zuylen (2010) for modeling the transient behavior of queues at signalized intersections while supporting general arrival and departure distributions. We will show that this model generalizes the classical bulk service queueing model in Section 3.3. A continuous extension of this model is discussed in Section 3.4. We discuss the classical FCTL model in Chapter 4 and present some existing results along with a numerical comparison with the bulk service queue. We turn to microscopic simulation models in Chapter 5, where we explain how we interact with the simulation software package SUMO to perform experiments in order to compare a SUMO model with the FCTL queue. We conclude our work in Chapter 6 and mention some pointers to possible further work. A selection of the scripts and notebooks that we used in this study have been included in the appendices.

[1] See for example this news article (in Dutch) about a recent implementation of a queue prediction system near Schiphol https://www.verkeersnet.nl/mobiliteitsmanagement/34860/noord-holland-voorspelt-wachtrijen-voor-verkeerslicht/

Chapter 2

Background and history of traffic modeling

This chapter aims to give an overview of different aspects of traffic modeling. First, we discuss some models from the theory of traffic flow modeling, which tries to explain the collective behavior of vehicles by modeling individual vehicle movement or by considering a large amount of vehicles as a continuum. After that we turn to queueing models for modeling queues near signalized intersections.

2.1 Traffic flow modeling

Traffic flow modeling is an important aspect in traffic modeling, since the behavior of vehicles on links determines the arrival process at (signalized) intersections. Therefore, we include a short overview of the different types of traffic models that exist. For a high-level overview of the development of traffic flow models in historical perspective, we refer to van Wageningen-Kessels *et al.* (2015).

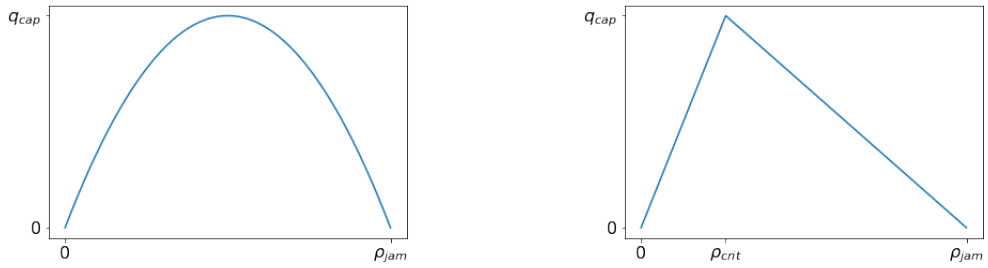
The earliest work on traffic flow modeling stems from the 1930s, when Greenshields (1934) studied how the head-to-head distance between vehicles is related to their velocity, which led to the concept of the *fundamental diagram*. During the 1950s and 1960s, three categories of flow models arised that aim to describe the temporal dynamics of traffic: Studying aggregated quantities over a large number of vehicles is the aim of (1) *macroscopic models*. These models consider traffic as being a continuum, and often focus on measures of traffic at a large scale such as flow (vehicles per hour) or density (vehicles per kilometer) at a specific road cross-section. They are sometimes derived from and explained as fluid models and many of them are described in terms of differential equations. In contrast, (2) *microscopic models* try to model the dynamics of each individual vehicle by specifying how its speed changes based on a set of rules. We will discuss some variants of these so-called car-following models later in this chapter, because they form the basis of many microscopic simulation software packages that are used in practice. The third category is that of *mesoscopic models* (3), whose level of detail is in between that of micro- and macroscopic models. In Chapter 5 we will use a microscopic model in order to study the predictive capability of an equivalent queueing model.

2.2 Fundamental diagram and macroscopic models

Macroscopic traffic flow models are used to get insight into the relationships between measures of traffic flow such as velocity, vehicle density, delay and congestion. Traffic flow is often characterized by the following equation

$$q = v\rho \,,$$

where q is the flow in vehicles per hour, v is the speed in kilometers per hour and ρ is the vehicle density in vehicles per kilometer.



(a) Greenshields (b) Daganzo

Figure 2.1: Schematic illustration of the two most well-known classic fundamental diagrams, showing the flow (vehicles per second) as a function of the vehicle density (vehicles per meter).

The classical fundamental relation (later called fundamental diagram) introduced by Greenshields *et al.* (1935) relates the vehicle density ρ (in vehicles per kilometer) to the velocity of vehicles v (kilometers per hour) as a linear relationship $v(\rho) = -\alpha\rho$ for some $\alpha \geq 0$. Therefore, the relationship between density and flow is parabolic, see Figure 2.1. This agrees with the intuition that increased vehicle density will increase the probability of traffic jams, which will eventually decrease the total traffic flow. Interesting phenomena related to traffic congestion can be observed from fundamental diagrams obtained from real traffic counts. Based on these *emperical fundamental diagrams*, other models have been proposed that better capture certain phase change phenomena observed in real traffic, of which the model of Daganzo (1994) is the most well-known. An interesting phenomenon is traffic hysteresis in the congested phase (corresponding to the part of the fundamental diagram to the right of ρ_{crit} in Figure 2.1). This hysteresis can be observed as different trajectories in the emperical fundamental diagram for the period in which congestion increases and the period when congestion decreases. Zhang (1999) argued that this effect occurs due to differences in how fast vehicles accelerate or decelerate and it has also been observed in larger traffic networks by Geroliminis & Sun (2011).

Whereas fundamental diagrams capture the steady-state behavior of traffic systems very well, macroscopic models can be viewed as a tool for explaining the temporal dynamics of the fundamental diagram. The first macroscopic traffic flow model was developed independently by Lighthill & Whitham (1955) and Richards (1956). Their model was later called the LWR model, which is essentially a law of conservation of vehicles. Let $N(x,t)$ denote the cumulative number of vehicles (this specification of traffic flow is called the *Eulerian coordinate system* see Knoop (2017)) that has passed a certain point x on the road at time t. The flow at that point may then be expressed as

$$q(x,t) = \frac{\partial N(x,t)}{\partial t} \,,$$

and the density may be expressed as

$$\rho(x,t) = -\frac{\partial N(x,t)}{\partial x} \,.$$

The negative sign is necessary because the N is monotonically decreasing in x for a fixed t (cumulative counts are higher upstream at any point in time). If N is twice differentiable, we get

$$\frac{\partial^2 N}{\partial x \partial t} = \frac{\partial^2 N}{\partial t \partial x} \,,$$

from which it follows that

$$\frac{\partial q}{\partial x} + \frac{\partial \rho}{\partial t} = 0 \,,$$

which models the conservation of vehicles on a road segment. The main assumption of the LWR model is that the flow q is a function of the density ρ alone in the fundamental relationship, so $q(\rho) = v(\rho)\rho$.

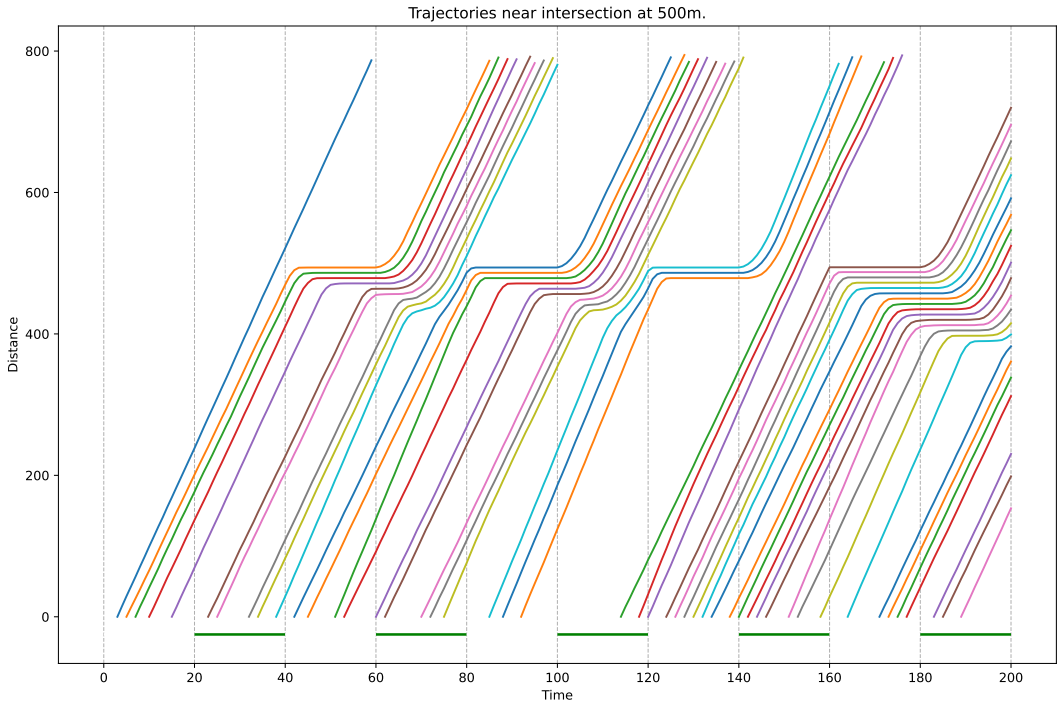


Figure 2.2: Trajectory plot of vehicles approaching an intersection at 500m. Both the green and the red phase take 20 time steps and the green phases are indicated by horizontal green lines. The very sharp halt at $t = 160$ was due to an emergency stop, which often happens in the SUMO simulation when no amber phase is specified. A similar plot of trajectories has been used for the cover of this report.

2.3 Microscopic models

Instead of considering collective behavior, microscopic models provide a desciption of the individual behavior of vehicles or drivers. The two most important microscopic model categories are car-following models and lane changing models. Car-following models try to model the change of speed of an individual vehicle based on the vehicle ahead and the general environment (road condition, speed limits, near an intersection). Furthermore, lane changing models can be used to model when a particular vehicle decides to switch lanes, which also enables us to model the process of overtaking. This study only considers single-lane traffic, so lane changing behavior is not relevant.

There are numerous traffic simulation software packages that are based on car-following models to simulate urban traffic. Such software provides tools to construct a microscopic model of the network infrastructure, which includes traffic lights. We can insert vehicles of different types into this network and assign each vehicle a route to follow. Determining these individual route choices based on the current state of the network is an interesting field of study on its own. Traffic demand can be defined using *origin-destination* (O-D) matrices, which specify the amount of trips between two locations in the network. These O-D matrices can be obtained by measuring real traffic in a specific area. Another model category that is often used are user equilibrium traffic assignment models. The cost of a certain route depends on the current traffic conditions, so drivers will try to find the least congested route (an increasing number of drivers is able to do this using navigation software that uses real-time information). Using mathematical optimization techniques a set of routes with the least total cost may be found. Once routes have been calculated, the microscopic simulation software computes the position a vehicle in the network over the course of a certain period. We can then choose to measure certain values that we are interested in. Figure 2.2 shows an example of simulated trajectories of vehicles near a traffic light. We will discuss the model that was used to construct this plot in more detail in Chapter 5.

The challenge of constructing a realistic car-following model is to maintain the collective phenomena found in real traffic. The accuracy of car-following models can be assessed by considering for example the fundamental diagram (flow-density relationship), the average gap between vehicles, or the passing time, the time which a vehicle needs to travel through a particular road segment. The latter measure is particularly interesting when studying the queueing process at intersections. Even the car-following model in the widely used traffic simulation software SUMO (see Chapter 5) has still issues in capturing the dynamic behavior of real traffic: for example, Bieker-Walz *et al.* (2017) found that the passing time of cars bending left on a traffic intersection with green light shows much less variance than what was found from real recordings of trajectories.

In the following we will give a short overview of the development of car-following models and the current status in microscopic simulation software. We refer to Brackstone & McDonald (1999) and Krauss (1998) for a more elaborate overview.

2.3.1 Model categories

Most car-following models are based on the idea that drivers only change their speed at the moment when their current speeds is not equal to some desired speed, which may be based on other traffic or road and safety conditions such as speed limitations. Therefore, most car-following models can be thought of as a differential equation of the form

$$\frac{dv_i(t)}{dt} = \frac{V_{des} - v_i}{\tau} ,$$

where v_i is the current velocity, V_{des} the desired velocity, and τ a time scaling factor that models the speed of adaptation. Initially, traffic flow was modelled using results from fluid dynamics. Later, it was recognized that when the density of cars gets higher, a phase transition happens from laminar flow to the occurance of start-stop waves. The ability of a car-following model to reproduce this behavior has been an important requirement. We now shortly present some of the different categories of car-following models, as a summary of the overview that Krauss (1998) provides.

Classical car-following models. In most classical models only the motion of the vehicle ahead is considered. The most simple model in this category is

$$\frac{dv_i(t)}{dt} = \frac{v_{i+1}(t) - v_i(t)}{\tau} ,$$

which has a stable steady state solution, so it does not model congestion. Therefore, extensions have been proposed to model instability at higher traffic densities.

Optimal velocity model. Instead of adjusting the velocity based on the velocity of the car ahead, it is also possible to use the distance to the vehicle ahead (gap) in the formula. These models are called optimal velocity models.

Discrete models. Instead of solving a differential equation, it may be more efficient computationally to implement a model in discrete time, or even in discrete space. This category is illustrated by the model of Nagel and Schreckenberg, which we will discuss shortly in Section 2.3.2.

Modeling based on human behavior. An example of this category is the model of Wiedemann (1974), which is also available in SUMO. These models are focused on the perception and reaction of human drivers. It may be reasonable, for example, to assume that human drivers are not able to perceive very small changes of velocity, and thus are also not able to react to them. Keeping perfect constant velocity is also not reasonably possible for most human drivers.

2.3.2 Cellular automaton

Among the first car-following models was a class of discrete time and space models, or *cellular automatons*. For single-lane traffic, Nagel & Schreckenberg (1992) proposed a cellular automaton based on some simple rules, in which the phase transition from laminar flow to start-stop waves could be observed. They model the lane as a finite array of L positions, each of which may contain

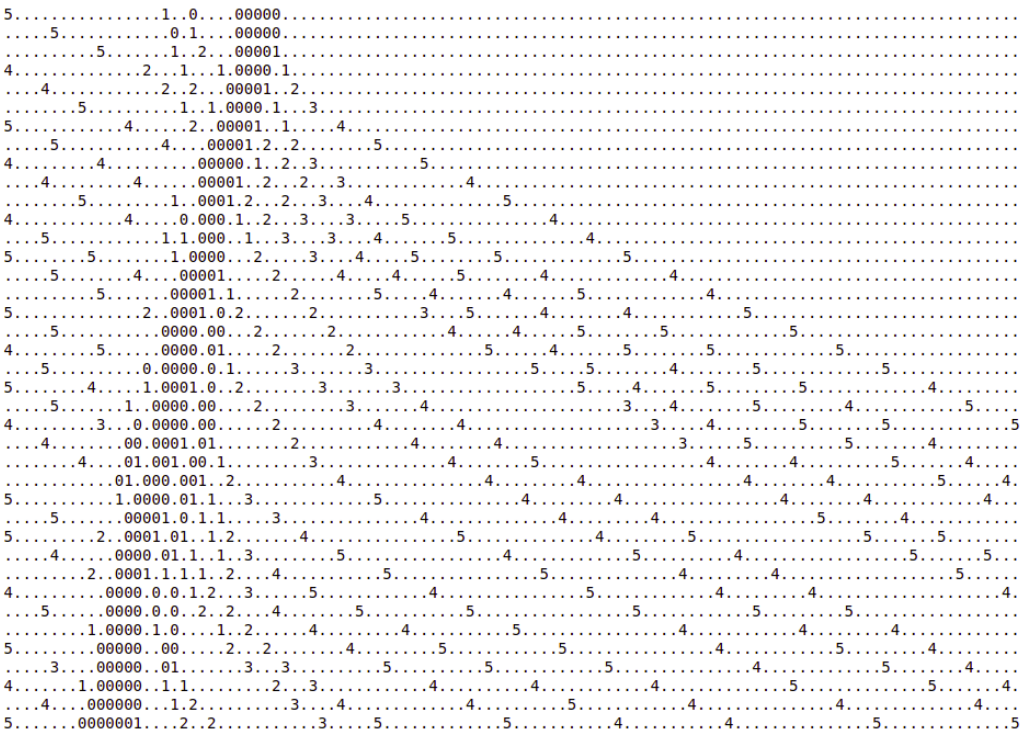


Figure 2.3: Behavior of an artificial traffic jam in the cellular automaton of Nagel and Schreckenberg with a lane of length $L = 100$, insertion probability of $p_i = 0.8$ and a probability of drivers slowing down of $p = 0.45$. The traffic jam is artificial because the initial state was constructed for the purpose of this illustration.

exactly one vehicle with an integer velocity between zero and v_{max}. In each time step, the velocity v of a vehicle at position i is updated based on the distance j to the next car in front of it. If $j > v + 1$, the car will accelerate ($v \to v + 1$). If $j \leq v$, then it will decelerate ($v \to j - 1$) to avoid direct collision. To model the random behavior of human driving, after these updates the velocity may further decreases by one with some probability p.

They consider both traffic in a closed system (periodic boundaries) and in a bottleneck situation (open boundaries). The closed system may be viewed as a circle on which a fixed number of cars are driving (race circuit). In the open system, a new car with velocity zero is put on the first site once it becomes free. By measuring the density and flow of cars in the system, averaged over 10^6 time steps, they constructed a fundamental diagram for the closed system that shows the same shape as diagrams constructed from real traffic data.

For the sake of illustration, we implemented this simple model in Python, see Appendix A. Vehicles are being inserted at the first position of the lane at a cruise speed of v_{max}. Whenever there is enough head-space (so the first vehicle in the lane is at position $j \geq v_{max} + 1$), a new vehicle will be inserted with probability p_i. Even this simple model is accurate enough to produce real traffic phenomena such as backpropagation of traffic jams, as can be seen in Figure 2.3. Using a discrete model may be advantageous in terms of computational speed. Therefore, it is useful to know that this fundamental phase transition can already be reproduced by a discrete model.

2.3.3 Car-following model in SUMO

SUMO is an example of a microscopic simulation software package that is used a lot by both researchers and practitioners in the field of traffic engineering. It supports a lot of different car-following models, but for our discussion in Chapter 5, we will stick to the default model, which was developed by Krauss (1998). In his thesis, he argues that the fundamental diagram alone is not

sufficient when assessing the accuracy of a car-following model, because there are other dynamical phenomena that are not captured in this diagram. The main point he makes is that traffic flow can exist in three different states: free flow, jammed traffic and synchronized traffic. The model family that Krauss proposes aims to be able to model these three traffic states using only three parameters: acceleration, deceleration and human driving perfection.

2.4 Queueing models

We will now give a very brief introduction to the mathematical analysis of queueing models, which appear in a wide range of fields. Queueing models are used to model, for example, communication systems, production systems, call centers and insurance claims. Earliest research on queueing models was done by Danish mathematician Erlang in the 1910s when he was studying the queueing behavior of phone calls at the Copenhagen Telephone Company (we refer to Brockmeyer *et al.* (1948) for an overview of the extensive body of work that Erlang left behind). A queue generally consists of one or more *servers* to which *customers* arrive for service. An analogy that is often used is that of a cashier at a shop, to which customers arrive at random. The cashier can handle one customer at a time, so customers that arrive while the cashier is busy should join the back of the queue. In very general terms, queueing models are characterized by the following three components:

(A) rate of arrivals, described by the distribution of inter-arrival times,

(S) distribution of service times,

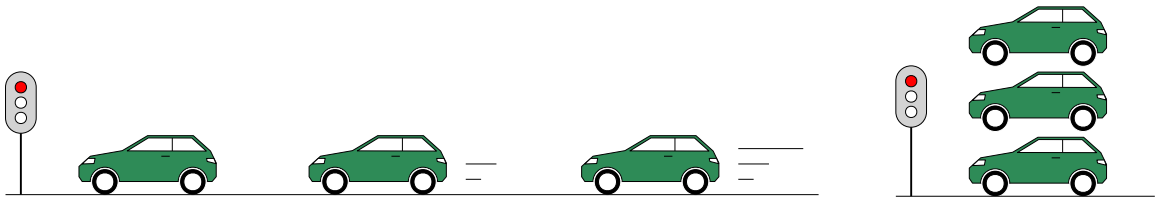
(c) number of servers.

Kendall (1953) introduced the now widely used notation for specifying each of these components as A/S/c. Poisson arrivals, so with exponentially distributed inter-arrival times, are denoted by the letter M (from memoryless or Markovian). For example, the elementary queue with Poisson arrivals (so exponential inter-arrival times) and exponential service times with one server is denoted as M/M/1.

When analyzing the behavior of queues, a number of different performance measures may be considered. First of all, the delay expresses the total time that a customer is in the system. Furthermore, the queue length (number of customers) can be considered, which is of particular interest in this study. When analyzing the performance of a queue, researchers often consider the steady-state behavior. This means that a situation is considered in which the distribution of the queue length does not change over time. This stable state can only exist if the queue length does not grow indefinitely over time, for which it is required that, loosely speaking, the amount of work that arrives is less than the amount of work that can be processed per time unit. More formally, let λ be the rate of arrival and $\mathbb{E}B$ the mean service time, then the amount of work arriving per time unit is given by $\rho := \lambda \mathbb{E}B$. One server can handle 1 unit of work per time unit, so when there are c servers available, the condition for stability is $\rho < c$. In the case of a single single server, ρ is called the *occupation rate*, because it represents the fraction of time during which the server is actually serving customers. The situation is a little bit different for queues of vehicles at traffic lights, because the server is not active during the period when the traffic light is red, which affects the stability condition as we will see in subsequent chapters.

There are also applications for which the non-steady, or *transient*, behavior is more interesting. For example, in dynamical control of traffic lights, it may be the case that the arrival rate is higher than the departure rate ($\rho \geq 1$), so that a steady-state does not even exist. In that case, it is only possible to use transient measures. When the system is stable ($\rho < 1$) we may make statements about how fast the system converges to the stationary state.

The expected value of the queue length gives already a lot of insight in the behavior of the queue. However, when we know the full distribution of the queue length at a particular time, we can calculate percentiles. In the setting of traffic light queues, these may be interpreted as the probability that the queue lenght ever grows beyond a given length. The lanes before a traffic light are of finite length, so when the queue grows beyond this length, it may start to cause congestion



(a) Horizontal queue

(b) Vertical queue

Figure 2.4: Illustration of the two different models categories for traffic light queues. The horizontal model does take into account the speed and position of each approaching vehicle on the lane and the acceleration that is needed for each vehicle to depart once the traffic light turns green. The vertical queue only considers the number of vehicles in the queue and assumes that each vehicle needs a constant time to depart from the queue during the green phase.

at an upstream intersection, which is known as spillback. The full distribution of the queue length enables us to calculate the probability of spillback.

Analyzing queueing systems can be done in a very elegant way using *probability generating functions (pgf)*. However, these methods have often been found to be prohibitive for practical application Abate *et al.* (2000), because a pgf needs to be inverted to obtain the actual probability distribution. The process of inversion depends on the actual generating function at hand. However, for the queueing models that we consider in this report, formulas are readily available for most distributions that are used in practise, as we will see in Section 4.4.

The rest of this report will focus on the analysis of a single lane for traffic on which a single traffic light is situated. The traffic light is assumed to have a fixed program, which means that the durations of the *green phase* and the *red phase* do never change. We will first consider two mathematical queueing models and then compare one of them to a microscopic model. For the queueing models, we use existing methods for computing the stationary queue length. The microscopic model can only provide us with approximations of the stationary queue length. The mathematical queueing models do not describe the exact position (in meters) of a vehicle in the queue. When an arrival happens, it is assumed that the vehicle stops immediately and joins the back of the queue. For this reason, these models are often called *vertical queueing models* because one may imagine that the vehicles form a stack just before the traffic light. When the light turns green, vehicles leave one by one at cruise speed, so acceleration is neglected. In contrast, models that do describe the exact speed and position of vehicles in the queue, such as the microscopic model that we constructed in SUMO, are called *horizontal queueing models*. We tried to make the distinction clear using the illustration in Figure 2.4. Comparing results between both model categories depends on the exact definition of queue length in vertical models, which is an interesting issue to which we will return in Chapter 5.

We will now provide some general notation that we will use throughout the rest of this report. Let X be a random variable. We denote the expected value of X as μ_X and the variance as σ_X^2. The variables T_g and T_r are used to denote the duration of the green phase and the red phase (in seconds) of a traffic light program, respectively. The total duration of one *cycle* of the traffic light program is denoted by $T_c = T_g + T_r$. We use Q to denote the random variable of queue lenght in number of vehicles.

Chapter 3

Probabilistic model for queue length

We are interested in the transient behavior of a queue of vehicles at a signalized intersection. Because of the dynamic nature of modern traffic, it is often not realistic to assume a stationary arrival rate over a long period of time. Therefore, a description of the transient behavior with varying arrivals is more appropriate for practical applications. Furthermore, a stationary analysis is only possible if the system is undersaturated ($\rho < 1$). However, in real traffic, it is no exception that the system becomes temporarily unstable because of a high arrival rate.

This chapter is mainly devoted to a discussion of the work done by Viti & van Zuylen (2010), who proposed a probabilistic method for calculating the full distribution of a queue at a signalized intersection for non-stationary arrivals and departures. First, some existing results and classical formulas are presented in Section 3.1. In Section 3.2, we introduce the *cycle-to-cycle* model of Viti and van Zuylen that describes the queue length just after the green phase has ended, called the *overflow queue*. We first discuss this model under the assumption of stationary arrivals and departures, in which case it may be described as a Markov chain. In this situation, we show in Section 3.3 that it is equivalent to the classical *bulk service queue*. Section 3.4 explains the continuous *within-cycle* extension that Viti and van Zuylen propose to describe the queue length at any moment during a cycle. There, we also discuss how to handle non-stationary arrivals and departures. We note that the work of Viti and van Zuylen also appeared as Viti & van Zuylen (2009), where they also consider the case of vehicle actuated traffic lights, which we will not discuss in this study.

3.1 Classical models

We will shortly discuss some classical formulas for the overflow queue that are also mentioned by Viti & van Zuylen (2010). Let us first define some more notation. Let T_r and T_g denote the red and green times, respectively. Following Viti and van Zuylen, we will assume in this chapter that a cycle of length $T_c = T_r + T_g$ starts with the red phase. Let n be the cycle number, starting from 1, and let $\tau_n = n \cdot T_c \geq 0$ denote the time epoch of the end of this cycle. Let the overflow queue of cycle n, which we denote as $Q_O(n)$, be defined as the number of vehicles in the queue just after the green phase of this cycle has ended, so at τ_n. Let μ_a be the average arrival rate in vehicles per second and let μ_d be the saturation flow rate, which is the average number of vehicles that can pass the intersection during the green phase ('d' for departures). In traffic engineering, the average *signal capacity* is defined as $c = \mu_d \cdot T_g/T_c$. The *degree of saturation* measures how much demand a traffic intersection experiences compared to its capacity and is defined as

$$x = \mu_a/c = \frac{\mu_a \cdot T_c}{\mu_d \cdot T_g}. \tag{3.1}$$

This value may be used to formulate a condition for stability of the traffic light as $x < 1$. This condition is equivalent to $\rho < 1$ that we discussed in Section 2.4. Under this stability condition,

Miller (1968) described the expected overflow queue in equilibrium as

$$\mathbb{E}[Q_O] = \frac{\exp\Big[-1.33 \cdot \sqrt{\mu_d \cdot T_g \cdot (1-x)/x}\Big]}{2 \cdot (1-x)}, \tag{3.2}$$

which is one of the most popular expressions used by practitioners. This formula was later simplified by Akçelik (1980) to

$$\mathbb{E}[Q_O] = \frac{1.5 \cdot (x - x_0)}{1-x}, \tag{3.3}$$

where $x_0 = 0.67 + \mu_d \cdot T_g/600$, which represents a threshold for when the overflow queue becomes non-negligible. Expression (3.3) is valid for $x_0 < x < 1$, and zero for $x \leq x_0$. Using the idea of a coordinate transformation of Kimber & Hollis (1979), Akçelik (1980) generalized his expression to provide a time-dependent description of the expected overflow queue

$$\mathbb{E}[Q_O(t)] = \frac{c \cdot t}{4}\left(x - 1 + \sqrt{(x-1)^2 + \frac{12 \cdot (x - x_0)}{c \cdot t}}\right), \tag{3.4}$$

which simplifies to (3.3) for $t \to \infty$ and $x < 1$. Wu & Brilon (1990) have shown that this expression does not yield good approximations when the arrival rate is non-stationary. Moreover, one may argue that this expression is the result of a heuristic approach of fitting formulas to real traffic measurements without a valid theoretical underpinning to construct a general model.

The goal of Viti and van Zuylen is to provide a methodological framework for computing the transient queue length at a signalized intersection with fixed control (constant red and green times). Calculating the transient dynamics also allows one to consider systems that are temporarily unstable. Moreover, instead of only focusing on the expected value, Viti and van Zuylen consider the full distribution of the queue length. For reasons of presentation, most figures in this report will only show the expected value. However, we note that knowing the full distribution of X allows us to compute measures such as percentiles $\mathbb{P}(X \leq x)$ or the probability of spillback $\mathbb{P}(X > x)$ for a given maximum queueing space x. The latter may be particularly interesting if one is interested in congestion effects in a network with several intersections close to each other.

3.2 Cycle-to-cycle model

Viti and van Zuylen first introduce a model for the evolution of the overflow queue from cycle to cycle. Later, they extend this model to also describe the queue length in continuous time. Let us now consider this discrete cycle-to-cycle model. Note that the formulation here may use slightly different notation, because we think that the orignal formulas contained some small unintentional errors. We will first consider the model as a Markov chain. The states $Q_O(n) \geq 0$ represent the overflow queue length at the end of cycle n, so at time epoch τ_n. The main principle of the model is that the overflow queue $Q_O(n)$ can be completely described in terms of the overflow queue of the previous cycle $Q_O(n-1)$ and the total number of arrivals A_n and departures D_n in the current cycle $(\tau_n - T_c, \tau_n]$ by the following relationship

$$Q_O(n) = \max\{0, Q_O(n-1) + A_n - D_n\}. \tag{3.5}$$

It is assumed that the arrivals A_n and departures D_n in cycle n are *independent and identically distributed* (i.i.d.) according to random variables A and D, respectively. This assumption guarantees that the model is a Markov chain, which enables us to compute the queue length in a cycle from the previous cycle by matrix multiplication with a transition matrix. We now formulate this transition matrix P_{ij}, which gives the probability of moving from state $Q_O(n-1) = j$ to state $Q_O(n) = i$ during any cycle n. The probability of transitioning to a non-empty queue ($i > 0$) is given by

$$P_{ij} = \sum_{j+a-d=i} \mathbb{P}(A = a) \cdot \mathbb{P}(D = d), \tag{3.6}$$

the probability of transitioning into an empty queue is given by

$$P_{0j} = \sum_{j+a-d\leq 0} \mathbb{P}(A = a) \cdot \mathbb{P}(D = d). \tag{3.7}$$

Let us now slightly abuse notation to denote the distribution of $Q_O(n)$ as a column vector $(Q_O(n))_j = \mathbb{P}(Q_O(n) = j)$, then the evolution follows the simple recursive relation

$$Q_O(n) = PQ_O(n-1), \tag{3.8}$$

which allows us to express the overflow queue after any number of cycles in terms of the initial queue $Q(0)$ as

$$Q_O(n) = P^n Q_O(0). \tag{3.9}$$

Note that the above distribution and transition matrix both have infinite dimensions. Therefore, in order to compute the distribution of numerically, some further assumptions are needed. The queue is assumed to have a maximum length Q_{max}, so there are finitely many states to compute. Relationship (3.5) may be changed accordingly by clamping the state between 0 and Q_{max} as follows

$$Q_O(n) = \max\{0, \min\{Q_O(n-1) + A_n - D_n, Q_{max}\}\}. \tag{3.10}$$

Probability (3.6) is still valid for $0 < i < Q_{max}$, but the probability for $i = Q_{max}$ now needs to account for the absorbing behavior of the new upper boundary, and is given by

$$P_{Q_{max},j} = \sum_{j+a-d\geq Q_{max}} \mathbb{P}(A = a) \cdot \mathbb{P}(D = d). \tag{3.11}$$

We note that (3.6), (3.7) and (3.11) still represent infinite summations if the support of either arrivals A or departures D is infinite. Therefore, we truncate the summations in the following way. Let F_A be the cumulative distribution function of random variable A, defined as $F_A(a) := \mathbb{P}(A \leq a)$, then the quantile function of A is defined as

$$Q_A(p) := \inf\{x \in \mathbb{R} : p \leq F(x)\}. \tag{3.12}$$

Now choose ν small, then we define $A_{max} = Q_A(1-\nu)$ and $D_{max} = Q_D(1-\nu)$. Now we are able to compute a finite transition matrix in the following way

$$P_{ij} = \sum_{a=i-j}^{A_{max}} \mathbb{P}(A = a)\mathbb{P}(D = j + a - i) \qquad \text{for } 0 < i < Q_{max}, \tag{3.13}$$

$$P_{0j} = \sum_{a=0}^{A_{max}} \sum_{d=j+a}^{D_{max}} \mathbb{P}(A = a)\mathbb{P}(D = d) \qquad \text{(underflow)}, \tag{3.14}$$

$$P_{Q_{max},j} = \sum_{Q_{max}-j}^{A_{max}} \sum_{d=0}^{j+a-Q_{max}} \mathbb{P}(A = a)\mathbb{P}(D = d) \qquad \text{(overflow)}. \tag{3.15}$$

See Appendix B for details on the calculations in Python.

We may now approximate the stationary queue length by computing (3.9) for large n. See Figure 3.1 for an example of the evolution of the expected overflow queue length with Poisson distributed arrivals and deterministic departures. The stationary overflow queue (3.3) and the transient expected overflow queue (3.4) of Akcelik are also plotted for comparison. We see that his transient formula indeed approaches the stationary limit.

Note that the model provides the full distribution of the overflow queue, not only the expected value like some of the classical formulas. This enables us to compute, for example, the probability that the overflow queue is larger than some given value, which could be interpreted as the probability of the occurance of spillback effects.

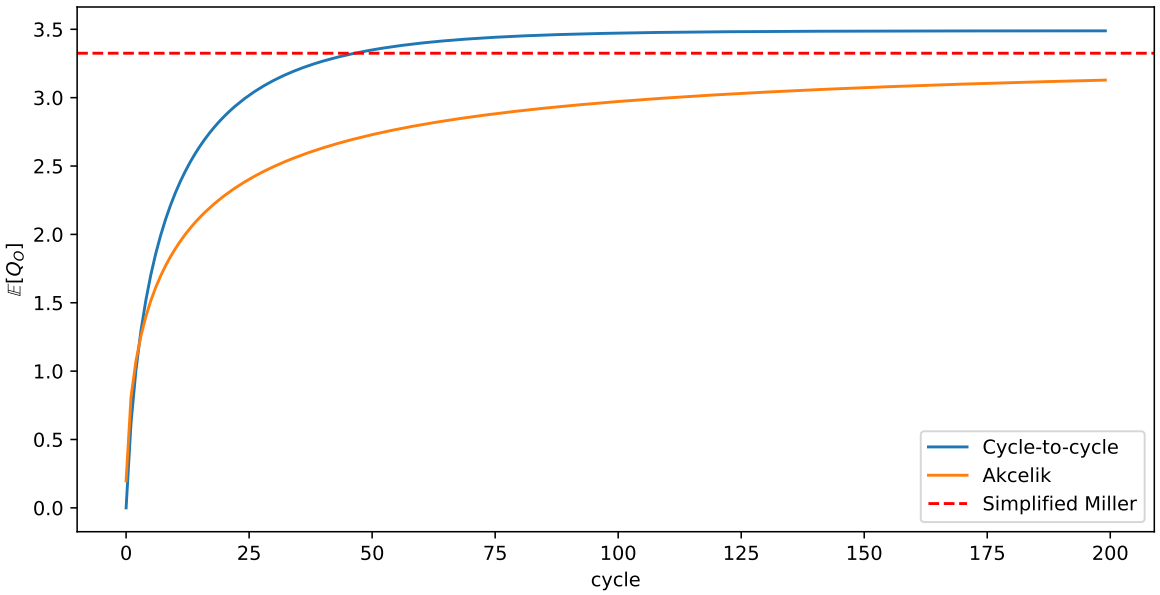


Figure 3.1: Expected overflow queue for 200 cycles, with $T_r = T_g = 5$ and arrivals $A_n \overset{iid}{\sim} \text{Poisson}(4.5)$, constant deterministic departures $D_n \overset{iid}{\sim} D \equiv T_g$. The cycle-to-cycle model has a maximum queue length of $Q_{max} = 40$ and we used $\nu = 10^{-7}$ to truncate the sums to a maximum number of arrivals per cycle.

We note that the above description allows A and D to have any distribution. This enables us to use real traffic data to define demand and capacity. Suppose that a traffic light is located at a single lane at some location x. We could use the difference between the cumulative counts of vehicles at some predetermined distance Δx from the traffic light to define the arrival distribution by considering the arrivals in cycle n, given by $A_n = N(x - \Delta x, \tau_n) - N(x - \Delta x, \tau_{n-1})$. Similarly, we may choose D to fit the number of departures that we can measure from an induction loop after the stop line.

The assumption of i.i.d. arrivals and departures caused the model to have the Markov property, which enables very efficient calculations using matrix multiplication. This assumption means that arrivals and departures have a constant rate over all cycles. However, in real traffic, these rates are constantly changing due to the dynamic nature of traffic in urban networks (influence of nearby signalized intersection) and possibly due to periods of increased traffic intensity, e.g., morning rush hour. In order to have a more realistic model, we could drop the assumption of identically distributed arrivals and departures. This would entail that we have to calculate a different transition matrix for every cycle, so equation (3.9) is no longer valid, which will affect the computation time. We will come back to this topic when we discuss the continuous-time extension in Section 3.4.

3.3 Bulk service queue

The cycle-to-cycle model may be viewed as a generalization of the classical bulk service queue, which we will discuss in this section. The bulk service queue is introduced from a historical perspective by van Leeuwaarden (2005), showing some of the development of queueing theory along the way. We will briefly give an overview of some existing results.

Bulk service queueing models exist in some different forms. The M/D/s queue was first studied by Erlang in the 1920's, see Brockmeyer *et al.* (1948), to model the process of handling phone calls at a telephone exchange. Incoming calls arrive according to a Poisson process and can be handled by s available lines. Each call lasts for a fixed 'holding time' v. When we consider the number of calls in the system X_n (each either waiting in the queue or occupying a single channel) at the fixed

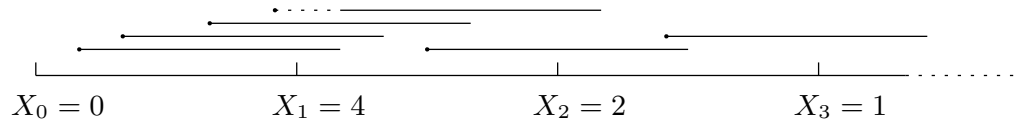


Figure 3.2: Evolution of a bulk service queue with $s = 3$, illustrating that the exact arrival epochs do not really matter for the queue size at the fixed points τ_n as long as they happen in the same interval $(\tau_{n-1}, \tau_n]$. We see that the 4th arriving call had to wait (dashed line) until the first call finished.

epochs $\tau_n = n \cdot v$, we get a Markov chain satisfying the relation

$$X_n = (X_{n-1} - s)^+ + A_n \ , \tag{3.16}$$

where $x^+ = \max\{x, 0\}$, and A_n is the number of calls that arrived during the time interval $(\tau_{n-1}, \tau_n]$, see Figure 3.2 for an illustration. It is assumed that the arrivals per cycle A_n are i.i.d. according to a random variable A. The exact moment of an arrival within a time slot does not matter for the analysis of the bulk service queue.

In the cycle-to-cycle model, the exact moment of arrivals or departures within a cycle do not matter for the analysis or computations. Let us assume that the traffic light has fixed green and red times, so that the number of vehicles s that can leave during the green phase is fixed and arrivals are i.i.d according to some random variable A. In that case, we can show that the cycle-to-cycle model reduces to a bulk service queue by using a formula similar to (3.16) to express the recursion (3.8) for the overflow queue length as

$$X_n^{'} = (X_{n-1}^{'} + A_n - s)^+ \ . \tag{3.17}$$

The processes that equations (3.16) and (3.17) describe are equivalent, the only difference is the moment of evaluating the queue lenght. Formula (3.16) may be seen as an expression for the queue length at the start of the green phase (end of red phase), whereas formula (3.17) expresses the queue length after the green phase (overflow queue). When we substitute $X_n = X_{n-1}^{'} + A_n$ into (3.17), we obtain the relationship

$$X_n^{'} = (X_n - s)^+ = X_{n+1} - A_{n+1} \ . \tag{3.18}$$

Therefore, the stationary queue lengths are related as

$$E[X^{'}] = E[X] - \mu_A, \tag{3.19}$$

which we need for numerical comparison, because the formulas that we use for the bulk service calculations apply to (3.16).

Using a generating function technique it is possible to derive the full distribution of the expected stationary queue length in the bulk service queue, which was done first by Bailey (1954) for some specific classes of arrival distributions. Using the results from Secion 2.2 in van Leeuwaarden (2005), we are able to calculate the queue length for general arrival distributions. Let the pgf of A be denoted by $A(z)$, and let the s roots of equation $z^s = A(z)$ in $|z| < 1$ be denoted by $z_0 = 1, z_1, \ldots, z_{s-1}$. If the expected number of arrivals per cycle is less than the capacity of the server, $\mu_A < s$ (Theorem 2.2.1 in van Leeuwaarden (2005)), then these roots lie on or within the unit circle, and the stationary queue has expected value and variance

$$\mu_X = \frac{\sigma_A^2}{2(s-\mu_A)} + \frac{1}{2}\mu_A - \frac{1}{2}(s-1) + \sum_{k=1}^{s-1} \frac{1}{1-z_k}, \tag{3.20}$$

$$\begin{aligned} \sigma_X^2 = \sigma_A^2 &+ \frac{A'''(1) - s(s-1)(s-2)}{3(s-\mu_A)} + \frac{A''(1) - s(s-1)}{2(s-\mu_A)} \\ &+ \left(\frac{A''(1) - s(s-1)}{2(s-\mu_A)} \right)^2 - \sum_{k=1}^{s-1} \frac{z_k}{(1-z_k)^2}. \end{aligned} \tag{3.21}$$

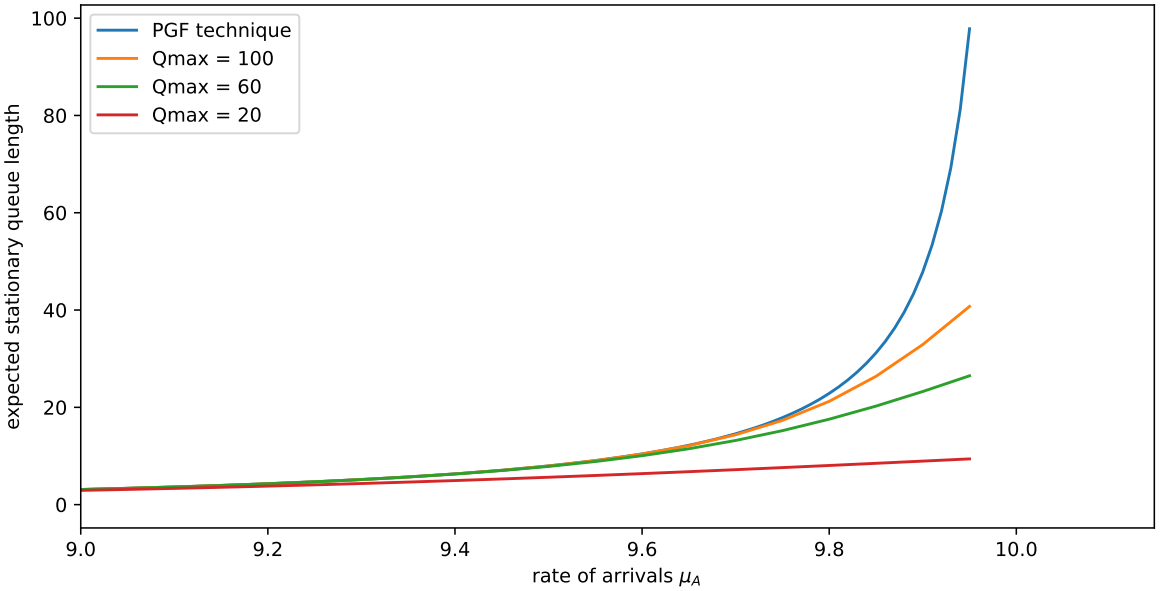


Figure 3.3: Stationary expected queue length $\mathbb{E}[X^{'}]$ computed analytically (using generating functions) for the bulk service queue compared to approximations $\mathbb{E}[Q_O(n)]$ computed for $n = 1000$ cycles with different settings of Q_{max}. We used $\nu = 10^{-7}$. The effect of the finite buffer is clearly visible for high rates of arrivals.

We will turn back to the issue of finding these roots z_k and inverting a pgf to find the full distribution of a random variable in Chapter 4.

The main benefit of the above stationary analysis is that we do not have to assume a finite queue space or *buffer size* as we did for the cycle-to-cycle model. However, we note that for most pratical applications in road traffic engineering, the assumption of a finite queueing space is desirable. In order to check our implementation of the cycle-to-cycle model and to study the effect of this finite buffer, we compare for Poisson arrivals A with varying μ_A and constant deterministic departures s the stationary expected queue length $\mathbb{E}[X^{'}]$ computed with (3.20) and the approximation $\mathbb{E}[Q_O(n)]$ that we obtain by computing (3.9) for a large number of cycles n. The results of this comparison are illustrated in Figure 3.3. As we expect, a higher rate of arrival always leads to a higher stationary overflow queue. Moreover, it is clearly visible that a smaller buffer size results in a lower expected queue length, for any rate of arrivals μ_A. It would be interesting to see if this stochastic ordering of models can be shown formally.

3.4 Continuous within-cycle extension

The cycle-to-cycle model and the bulk service queue only provide a description of how the overflow queue changes between cycles. Viti and van Zuylen extend their cycle-to-cycle to a continuous-time *within-cycle* model that describes the evolution of the queue length within a given cycle. For this description, they assume that the rate of departure μ_d, in vehicles per second, is constant for all cycles, so that the number of departures D_n that happen during the green phase is deterministic and given by $D_n = D = [\mu_d T_g]$ for all n, where brackets are used to denote the integer part. Instead of defining a single transition matrix, they reformulate the cycle-to-cycle model by describing the transitions directly in the following way. Given the current state $0 < i < Q_{max}$ and the previous state j, the number of arrivals that must have happened is given by $A_n = i + D - j \geq 0$, so we must have $j \leq i + D$. Because of the finite queueing space, we also have $j \leq Q_{max}$, so we define $j_{max} := \min\{i + D, Q_{max}\}$. Therefore, for deterministic departures, the equivalent of (3.8) is given

by

$$\mathbb{P}(Q_O(n) = i) = \sum_{j=0}^{j_{max}} \mathbb{P}(Q_O(n-1) = j) \cdot \mathbb{P}(A_n = i + D - j) \quad \text{for } 0 < i < Q_{max}. \tag{3.22}$$

Again, the boundaries $i = 0$ and $i = Q_{max}$ must be considered separately, because there are multiple values for the number of arrivals that lead to these states. When the queue empties ($i = 0$), the number of arrivals must have been such that $j + A_n - D \leq 0$, from which follows that $0 \leq A_n \leq D - j$. Therefore, we get

$$\mathbb{P}(Q_O(n) = 0) = \sum_{j=0}^{j_{max}} \mathbb{P}(Q_O(n-1) = j) \cdot \sum_{a=0}^{D-j} \mathbb{P}(A_n = a)\ . \tag{3.23}$$

Similarly in the case of spillback ($i = Q_{max}$) we need $j + A_n - D \geq Q_{max}$, so there is an infinite number of values possible for A_n, so the we get

$$\mathbb{P}(Q_O(n) = Q_{max}) = \sum_{j=0}^{j_{max}} \mathbb{P}(Q_O(n-1) = j) \cdot \sum_{a=Q_{max}+D-j}^{\infty} \mathbb{P}(A_n = a), \tag{3.24}$$

where the last summation may also be written in terms of a cumulative distribution function. Note that we dropped the assumption of identically distributed arrivals by writing A_n instead of A, so the above approach shows how to compute the cycle-to-cycle model for varying rates of arrivals, which we touched on briefly in Section 3.2. It is also possible to drop the assumption for departures, but that requires us to consider discrete convolutions, which further complicates the analysis.

Formulas (3.22), (3.23) and (3.24) may now easily be extended to continuous time by defining the queue length $Q(\tau_{n-1} + \Delta t)$ at any time whitin cycle n, such that $Q(\tau_n) = Q_O(n)$. Viti and van Zuylen allow the moments of arrivals to vary freely within a cycle while still requiring the total number of arrivals in a cycle to be independently and identically distributed accross cycles. Therefore, the probability that a arrivals have happened after Δt seconds since the start of cycle n may be provided as a function $f_n(a, \Delta t)$, satisfying $f_n(a, T_c) = \mathbb{P}(A_n = a)$.

During the red phase, no departures happen, so we do not have to consider the case of underflow ($i = 0$) separately. Therefore, the queue length after $0 \leq \Delta t < T_r$ seconds since the begin of the cycle can be described by

$$\mathbb{P}(Q(\tau_{n-1} + \Delta t) = i) = \begin{cases} \sum_{j=0}^{i} \mathbb{P}(Q(\tau_{n-1}) = j) \cdot f_n(i - j, \Delta t) & \text{for } 0 \leq i < Q_{max}, \\ \sum_{j=0}^{Q_{max}} \mathbb{P}(Q(\tau_{n-1}) = j) \cdot \sum_{a=Q_{max}-j}^{\infty} f_n(a, \Delta t) & \text{for } i = Q_{max}. \end{cases} \tag{3.25}$$

After $0 \leq \Delta t < T_g$ seconds from the start of the green period, the number of departures that could have happened is given by $[\mu_d \Delta t]$, so the queue length during the green phase can be described as

$$\begin{aligned} &\mathbb{P}(Q(\tau_{n-1} + T_r + \Delta t) = i) \\ &= \begin{cases} \sum_{j=0}^{[\mu_d \Delta t]} \mathbb{P}(Q(\tau_{n-1} + T_r) = j) \cdot \sum_{a=0}^{[\mu_d \Delta t]-j} f_n(a, \Delta t) & \text{if } i = 0, \\ \sum_{j=0}^{\min\{i+[\mu_d \Delta t], Q_{max}\}} \mathbb{P}(Q(\tau_{n-1} + T_r) = j) \cdot f_n(i + [\mu_d \Delta t] - j, \Delta t) & \text{for } 0 < i < Q_{max}, \\ \sum_{j=0}^{Q_{max}} \mathbb{P}(Q(\tau_{n-1} + T_r) = j) \cdot \sum_{a=Q_{max}+[\mu_d \Delta t]-j}^{\infty} f_n(a, \Delta t) & \text{if } i = Q_{max}. \end{cases} \end{aligned} \tag{3.26}$$

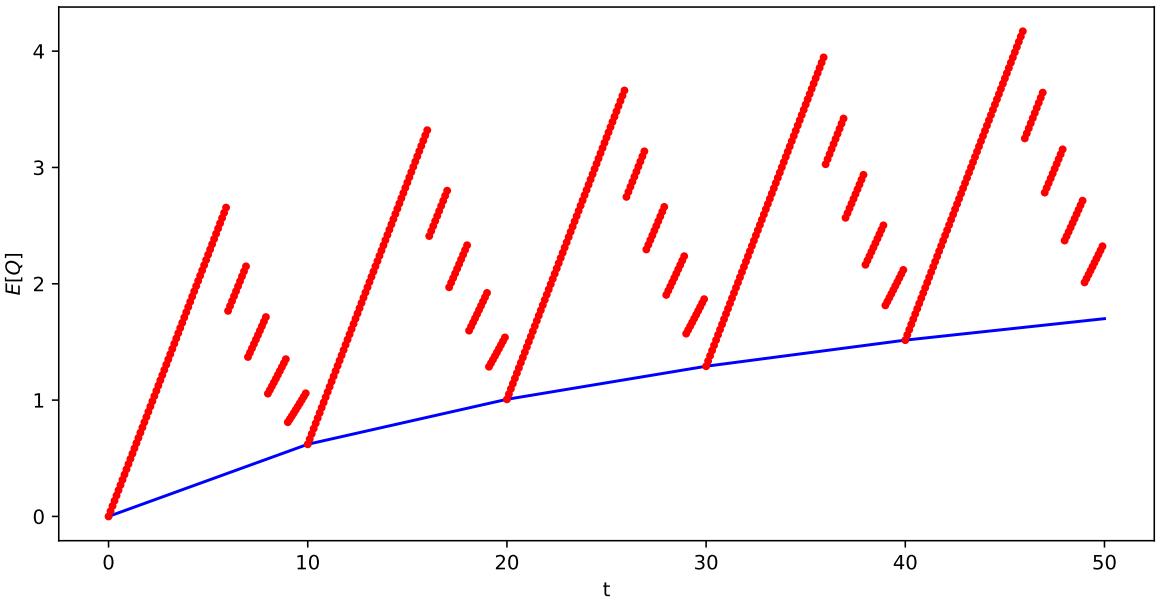


Figure 3.4: Expected queue length within the first 5 cycles using the same settings as in Figure 3.1 for the cycle-to-cycle model (blue) and the within-cycle model (red).

We added the above described calculation schema on top of our implementation of the cycle-to-cycle model, see Appendix B.6. For the same settings as in Figure 3.1, we calculated the queue length within the first 5 cycles and compared this with the overflow queue calculated using the cycle-to-cycle model, see Figure 3.4. We verify that it is indeed an extension. As we would expect, the expectation of the queue length grows linearly during the red phase. The instantaneous drops in the green phases are due to the fixed departure times.

3.5 Discussion

We believe that the original description of the cycle-to-cycle model by Viti and van Zuylen hides the fact that we are dealing with a generalized version of the classical bulk service model. We made the assumption of the maximum queue length Q_{max} (buffer size) explicit and studied its influence on the stationary expected queue length. We think that analytical techniques are also available for calculating the stationary queue length for the bulk service queue with finite buffer size. Unfortunately, we did not have time to search in the literature for relevant sources.

Furthermore, we have shown that the within-cycle model indeed extends the cycle-to-cycle model and that it may give some insight in how the queue fills and empties during a cycle. However, we will see in Chapter 4 that this same behavior can also be described by the FCTL model, which is more appropriate for traffic lights.

Chapter 4

FCTL model

The fixed cycle traffic light (FCTL) model is a classic description of the behavior of vehicle queues at traffic lights with fixed control. It is a time-slotted vertical queue model. It is assumed that the length of the red and green phases are predetermined and fixed throughout all cycles. Furthermore, it is assumed that during the green phase, exactly one vehicle will be able to pass the stop line per time slot. One could argue that this is not realistic because in reality vehicles need to accelerate, so the amount of vehicles that leave the queue is not adequately modelled as a linear function of the green time. However, we will show later in Section 5.2 by microscopic simulation that this assumption may not be as bad as it seems. The stationary distribution of the queue length was first analyzed in the 1960s by Darroch (1964) for (compound) Poisson arrivals. Almost 40 years later has this analysis been extended by van Leeuwaarden (2006) to allow for generally distributed arrivals and to provide a method of calculating the full distribution of the queue length. Because of the assumption of fixed phase lengths, the FCTL model cannot be directly used to model traffic lights with dynamic control, for example when induction loops are used.

4.1 Model description and stationary analysis

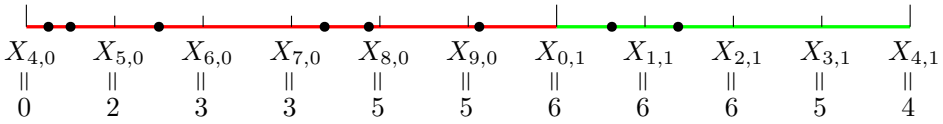
We will briefly introduce the FCTL model and its assumptions along the same lines as van Leeuwaarden (2006). The following three assumptions form the basis of the FCTL model:

Assumption 3.1 (Discrete-Time Assumption). Time is divided in fixed unit length time slots. The traffic light switches between a red phase of r time slots and a green phase of g time slots. Traffic engineers often use the effective green time, which is the portion of the green phase during which vehicles actually leave the queue. Therefore, amber light is not considered separately. A red phase followed by a green phase is called a cycle, which consists of $c = r + g$ time slots. Multiple vehicles may arrive during a time slot and join the queue at the end of this time slot. Vehicles depart from the queue during the green phase at a rate of one vehicle per time slot.

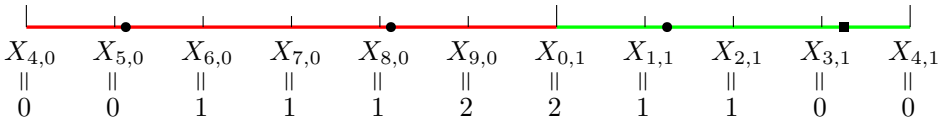
Assumption 3.2 (Independence Assumption). The number of arrivals that arrive at the intersection during slot k in cycle n, denoted by $Y_{k,n}$, is i.i.d. according to some discrete random variable Y, for all k and n. The probability generating function of Y is denoted by $Y(z)$.

Assumption 3.3 (FCTL Assumption). When the queue empties before the green phase is ended, all vehicles that arive after this moment can pass the intersection without any delay. The idea behind this assumption is that vehicles that arrive at an empty queue do not have to slow down for other vehicles that are halting in front of them. We will see that this assumption distinguishes the FCTL queue from the classical bulk service queue.

We now formulate the evolution of the queue length in each time slot. Let $X_{k,n}$ denote the number of vehicles in the queue at the end of time slot k in cycle n. Figure 4.1 shows two examples of how the queue length may evolve during consecutive red and green phases. The evolution of the queue



(a) Example situation in which there is an overflow queue.



(b) In this situation the FCTL assumption applies to the last arrival in the green phase.

Figure 4.1: Timelines showing arrivals in the red phase of cycles 0 and the green phase of cycle 1 with $r = 4, g = 6$. Each arrival is represented as a little circle (delayed) or a rectangle (not delayed due to FCTL assumption). The queue length $X_{k,n}$ after each time slot is given under the timeline.

length may be expressed as

$$X_{k+1,n} = \begin{cases} X_{k,n} + Y_{k+1,n} - 1 & \text{for } X_{k,n} \geq 1, \\ 0 & \text{for } X_{k,n} = 0, \end{cases} \tag{4.1}$$

for $k = 0, 1, \ldots, g-1$ (green phase) and

$$X_{k+1,n} = X_{k,n} + Y_{k+1,n} \tag{4.2}$$

for $k = g, g+1, \ldots, c-1$ (red phase). Furthermore, we have that $X_{c,n} = X_{0,n+1}$. Using expression (4.1), we may also express the queue length distribution during the green phase directly as

$$\mathbb{P}(X_{k+1,n} = i) = \sum_{j=1}^{i+1} \mathbb{P}(X_{k,n} = j)\mathbb{P}(Y_{k+1,n} = i - j + 1) \qquad \text{for } i \geq 1, \tag{4.3}$$

$$\mathbb{P}(X_{k+1,n} = 0) = \mathbb{P}(X_{k,n} = 0) + \mathbb{P}(X_{k,n} = 1)\mathbb{P}(Y_{k+1,n} = 0). \tag{4.4}$$

It is common in the queueing theory literature to analyze the distribution of the stationary queue length X_k, defined as $\mathbb{P}(X_k = j) = \lim_{n\to\infty} \mathbb{P}(X_{k,n} = j)$. By equating the pgf's as $X_{k,n+1}(z) = X_{k,n}(z)$, see van Leeuwaarden (2006), it then follows from (4.3) and (4.4) that the pgf of the stationary overflow queue is given by

$$X_g(z) = \frac{Y(z)^g(\zeta(z) - 1)\sum_{k=0}^{g-1} q_k \zeta(z)^k}{z^g - Y(z)^c}, \tag{4.5}$$

where $\zeta(z) = z/Y(z)$ and $q_k = \mathbb{P}(X_k = 0)$, which can be derived using the zeros of the denominator, to which we will return in Section 4.3. Using this pgf it is now possible to derive an expressions for the expected value by computing $(d/dz)X_g(z)|_{z=1}$, which is given by

$$\begin{aligned} \mathbb{E}X_g = &\frac{c\sigma_Y^2 + r^2\mu_Y^2 - g^2(1-\mu_Y)^2}{2(g - c\mu_Y)} - \frac{\sigma_Y^2}{2(1-\mu_Y)} + \frac{1-\mu_Y}{2} \\ &+ \frac{(1-\mu_Y)^2}{g - c\mu_Y} \sum_{k=0}^{g-1} k q_k. \end{aligned} \tag{4.6}$$

The expected value for the FCTL model with $g = r = 10$ and Poisson arrivals and binomial arrivals for varying rates of arrival μ_Y is shown in Figure 4.2. We note that a binomial distribution approximates a Poisson distribution for large values of n and $\lambda = np$ constant. We used this fact as a sanity check for our calculations.

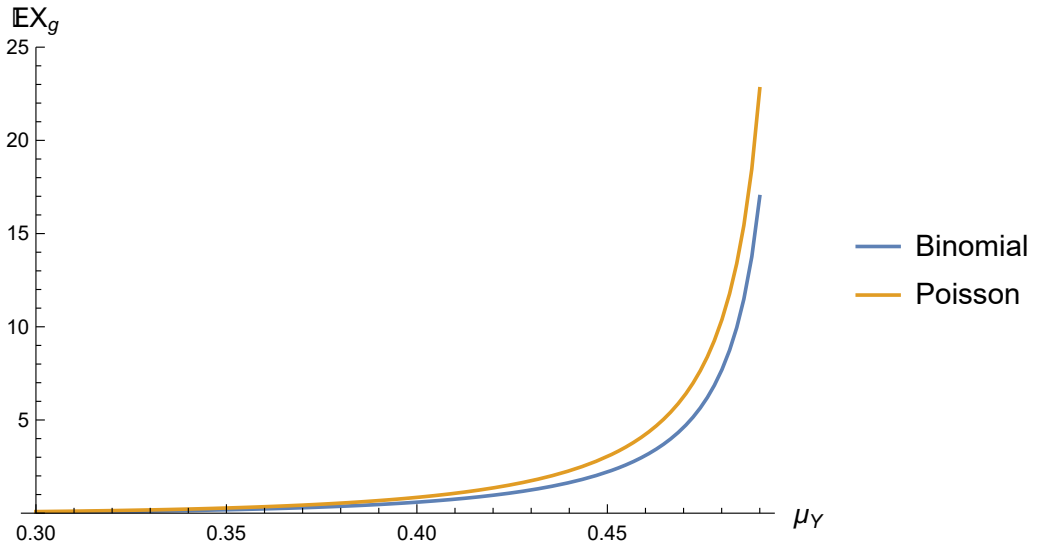


Figure 4.2: Expected value as computed with (4.6) for the FCTL model with $g = r = 10$ and binomial arrivals with shape parameter $n = 2$.

4.2 FCTL as extension of bulk service queue

The FCTL model may be regarded as an extension of the classical bulk service queue that we briefly discussed in Section 3.3. Without the FCTL assumption, the FCTL queue may be analyzed exactly like the bulk service queue because the exact moments of departures do not matter anymore. Furthermore, the FCTL assumption looses its effect when the number of arrivals per time slot is limited by one, which we will illustrate now. The number of arrivals during a consecutive red and green phase in the FCTL queue is given by

$$A_n = \sum_{k=g+1}^{c} Y_{k,n} + \sum_{k=1}^{g} Y_{k,n+1} \, .$$

Note that this notation is in line with the notation for the bulk service queue, cf. (3.16). van Leeuwaarden (2006) makes the following distinction for delayed vehicles

$$A_n = A_n^d + A_n^p, \tag{4.7}$$

with A_n^d the number of 'delayed' vehicles and A_n^p the number of vehicles that pass without any delay due to the FCTL assumption. They point out that when assuming $Y_{n,k}$ are Bernoulli random variables, then $A_n = A_n^d$, which would mean that all arrivals will be delayed and no arrivals will pass through on behalf of the FCTL assumption. Now consider the situation in which the queue is cleared before the green phase is over, i.e., there is a $0 \leq k' < g$ such that $X_{k',n+1} = 0$. Any arrival $Y_{k,n+1}$ that happens afterwards ($k > k'$) can immediately pass at the end of the time slot in which it arrived, and thus will have a delay of $D = 0$ time slots, if we use Definition 3.1 in van Leeuwaarden (2006). Therefore, saying that A_n^d consists of all vehicles that experience delay can be confusing. However, we can make the definition of A_n^d more intuitive if we would take into account the residual delay $D^R \in [0, 1]$, which is the delay that a vehicle still has to wait until the end of the time slot. Then we would define A_n^d as the number of vehicles that experiences positive delay $D^T = D + D^R$, and A_n^p as the number of vehicles that experience no delay at all ($D^T = 0$) because of the FCTL assumption. Suppose for example that $X_{k',n+1} = 0$ for some $0 \leq k' < g$, and that two vehicles arrive in the next time slot, so $Y_{k'+1,n+1} = 2$, then the first arrival will experience positive delay $D^T = D^R$, and thus will belong to A_n^d, but the FCTL assumption applies to the second arrival, and thus will belong to A_n^p. Now it is easily seen that the FCTL assumption makes no difference under the assumption of Bernoulli distributed arrivals.

When we consider the FCTL model with an arrival distribution that allows more than one arrival per time slot, the bulk service model provides us with an upper bound for the overflow queue. We will now illustrate the above described relation with some numerical results. In order to compare both models, we define the saturation rate as

$$x = \frac{\mu_A}{g} = \frac{\mu_Y \cdot T_c}{g}. \tag{4.8}$$

We compute the stationary overflow queue for the bulk service queue using (3.20) and for the FCTL queue using (4.6), where we assume that $s = g$. We note that the difference between both models becomes particularly clear when the ratio $y = g/r$ is small, because in that situation, the vehicles pass the intersection relatively more often because of the FCTL assumption. Figure 4.4 shows the expected overflow queue for each model with Poisson arrivals and binomial arrivals.

The relative distance computed as $(\mathbb{E}[X^{'}] - \mathbb{E}[X_g])/\mathbb{E}[X^{'}]$ is shown for Poisson arrivals in Figure 4.5. We see that this distance becomes smaller for larger saturation rates, as we would expect, because less vehicles get a chance of passing immediately with the FCTL rule. If we keep the ratio y constant while increasing the number of time slots for the green phase, we noticed that the relative distance becomes smaller, as can be seen from Figure 4.5.

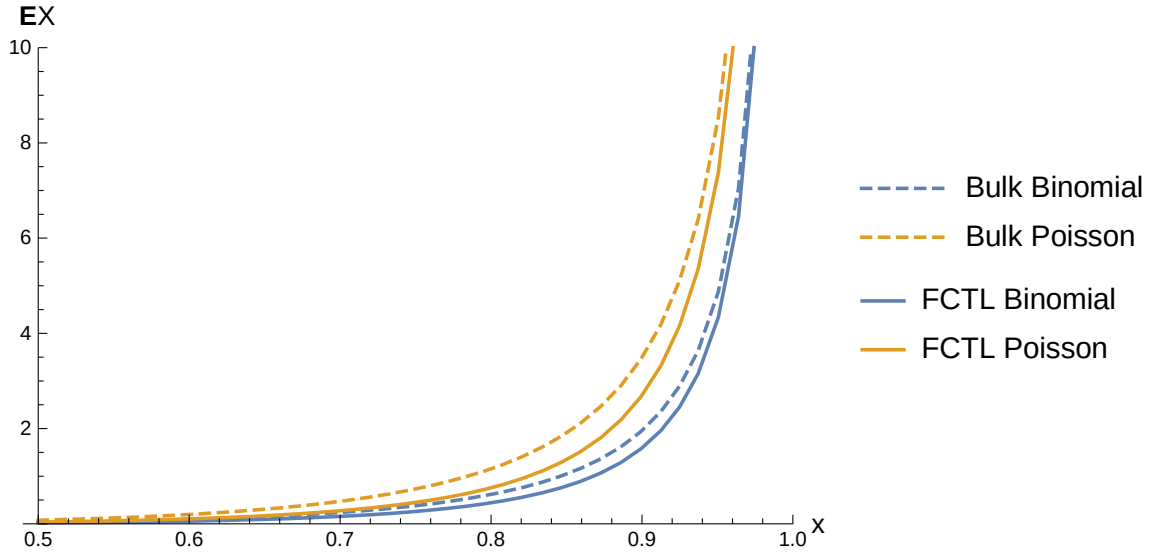


Figure 4.3: Expected stationary overflow queue

Figure 4.4: FCTL compared to bulk service for different degrees of saturation. Timing parameters are $s = g = 5, r = 1$. The rates of the Poisson and binomial arrivals (with shape parameter $n = 2$) have been chosen according to the saturation rate. It can be clearly seen that the bulk service queue always provides an upper bound. The relative distance is shown only for Poisson arrivals in Figure 4.5.

4.3 Determining empty queue probabilities

We will now discuss how to determine the probabilities q_k in (4.5). If $c\mu_Y < g$, so when the stationary distribution exists, it can be shown using Rouché's theorem, see for example Adan *et al.* (2005), that the denominator of (4.5) has g zeros $z_0 = 1, z_1, \ldots, z_{g-1}$ on or within the unit circle $|z| \leq 1$. Because a pgf is analytic and finite on and within the unit circle, the numerator of the right-hand side of (4.5) should vanish at these zeros. Assuming that $\mathbb{P}(Y = 0) > 0$, we have $|Y(z)^g| > 0$. The first zero $z_0 = 1$ yields a trivial equation, because $Y(1) = 1 \implies \zeta(1) - 1 = 0$. For the other zeros $|z_j| > 0$ we

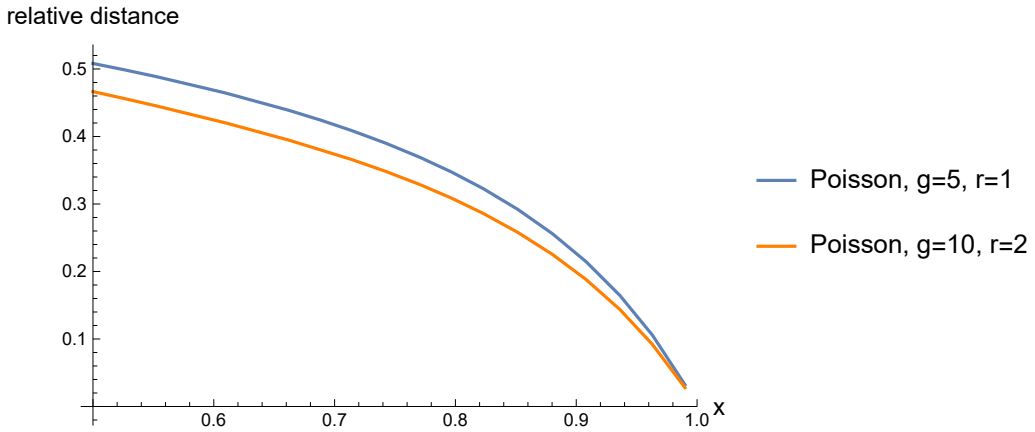


Figure 4.5: Relative distance $(\mathbb{E}[X^{'}] - \mathbb{E}[X_g])/\mathbb{E}[X^{'}]$ between FCTL and bulk service queue for Poisson arrivals.

have that $z_j^g = Y(z_j)^{g+r}$, so $z_j \neq Y(z_j)$ gives us that $\zeta(z_j) - 1 \neq 0$. Therefore, we end up with the $g-1$ equations

$$\sum_{k=0}^{g-1} q_k \zeta(z_j)^k = 0, \tag{4.9}$$

for $j = 1, \ldots, g-1$. We still need an additional equation in order to have a fully determined linear system. The normalization condition for pdfs $X_g(1) = 1$ provides an additional equation if we apply l'Hôspital's rule. We have

$$\frac{d}{dz}\zeta(z) - 1 \Big|_{z=1} = \frac{d}{dz} z Y^{-1}(z) \Big|_{z=1} = Y^{-1}(z) - z Y^{-2}(z) Y'(z) \Big|_{z=1} = 1 - \mu_Y \ .$$

Therefore, we obtain (using $\zeta(1) = 1$)

$$\frac{d}{dz}\left(Y(z)^g \sum_{k=0}^{g-1} q_k \zeta(z)^k\right)(\zeta(z)-1)\Big|_{z=1} = \left(Y(z)^g \sum_{k=0}^{g-1} q_k \zeta(z)^k\right) \frac{d}{dz}(\zeta(z)-1)\Big|_{z=1} = (1-\mu_Y)\sum_{k=0}^{g-1} q_k \ .$$

Furthermore, we see that

$$\frac{d}{dz} z^g - Y(z)^c \Big|_{z=1} = g z^{g-1} - c Y(z)^{c-1} Y'(z) \Big|_{z=1} = g - c\mu_Y \ ,$$

from which then follows that

$$\sum_{k=0}^{g-1} q_k = \frac{g - c\mu_Y}{1 - \mu_Y} =: \eta \ . \tag{4.10}$$

Now equations (4.9) and (4.10) together form the following system of linear equations

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \tau_1 & \tau_1^2 & \ldots & \tau_1^{g-1} \\ 1 & \tau_2 & \tau_2^2 & \ldots & \tau_2^{g-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \tau_{g-1} & \tau_{g-1}^2 & \ldots & \tau_{g-1}^{g-1} \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{g-1} \end{pmatrix} = \begin{pmatrix} \eta \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{4.11}$$

where $\tau_k = \zeta(z_k)$. van Leeuwaarden (2006) notes that this system may be solved using Cramer's rule and a result involving Vandermonde matrices. They provide the following explicit solution

$$\begin{aligned} q_j =& \eta(-1)^j \frac{1}{\prod_{k=1}^{g-1}(\tau_k - 1)} \\ & \cdot \sum_{1 \le i_1 < i_2 < \cdots < i_{g-1-j} \le g-1} \tau_{i_1} \tau_{i_2} \dots \tau_{i_{g-1-j}}. \end{aligned} \tag{4.12}$$

We recognize in the explicit expression some form of Leibniz formula for determinants, but we were not able to derive this ourselves. Furthermore, we note that using this expression to calculate the q_k directly becomes very slow for large g, because of the exponential run time ($O(2^g)$, because the indices of the sum define a subset of size $g-1-j$ out of a set of size $g-1$). For our numerical results we used the `LinearSolve`[1] method from the Wolfram Mathematica software. Note that there is also a hidden function in Mathematica that can be used for solving Vandermonde systems, which is called `LinearAlgebra`‘`Private`‘`VandermondeSolve` or `LinearAlgebra`‘`VandermondeSolve` in versions before 11.2.

Now we still need to determine the g zeros of $z^g = Y(z)^c$. The method depends on the chosen arrival distribution. We use the following expressions from Janssen & van Leeuwaarden (2005):

$$z_k = \sum_{l=1}^{\infty} c_l (e^{2\pi k i/g})^l, \tag{4.13}$$

for $k = 0, 1, \dots, g-1$, where the c_l depend on the chosen distribution. For Poisson arrivals $Y(z) = \exp(\lambda(z-1))$, we have

$$c_l = e^{-l\theta} \frac{(l\theta)^{l-1}}{l!} \tag{4.14}$$

and for binomial arrivals $Y(z) = (1-p+pz)^n$, we have

$$c_l = \frac{1}{l} p^{l-1} (1-p)^{l\beta - l + 1} \binom{l\beta}{l-1}. \tag{4.15}$$

We can now simply calculate the roots by truncating the sum in (4.13) to some finite L. For the case of $g = r = 10$ and binomial arrivals with shape parameter $n = 2$ and $\mu_Y = 0.45$, the roots are shown in Figure 4.6. The corresponding empty queue probabilities are shown in Figure 4.7.

4.4 Inversion of pgf

The generating function of the overflow queue needs to be inverted in order to obtain the full probability distribution, i.e., given $P(z) = \sum_{k=0}^{\infty} p_k z^k$, we want to retrieve $p_0, p_1, \dots, p_{g-1}$. We used the same method as van Leeuwaarden (2006), by using the following approximation of Abate *et al.* (2000):

$$p_k \approx \frac{1}{2kr^k} \sum_{j=1}^{2k} (-1)^j \mathrm{Re}(P(re^{ij\pi/k})). \tag{4.16}$$

In order to have accuracy up to the γth decimal, setting $r = 10^{-\gamma/2k}$ should be enough. However, we found that our calculations returned degenerate probability distributions (not summing to 1, negative values) when using $\gamma > 4$. Therefore, we set $\gamma = 4$ when calculating further results.

[1] https://reference.wolfram.com/language/ref/LinearSolve.html

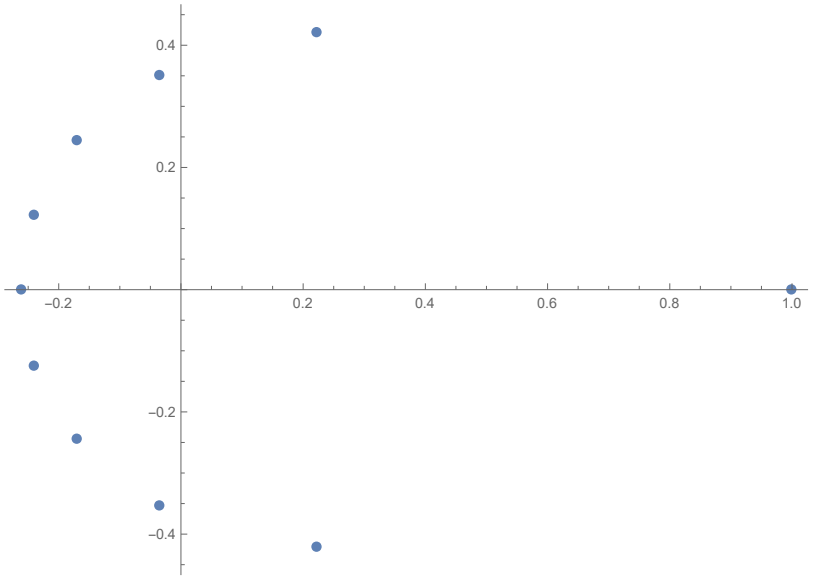


Figure 4.6: Example of roots calculated for the FCTL model with $g = r = 10$ and binomial arrivals with $n = 2$ and $p = 0.45$ (vertical axis is imaginary axis). The sum (4.13) was truncated to $L = 300$.

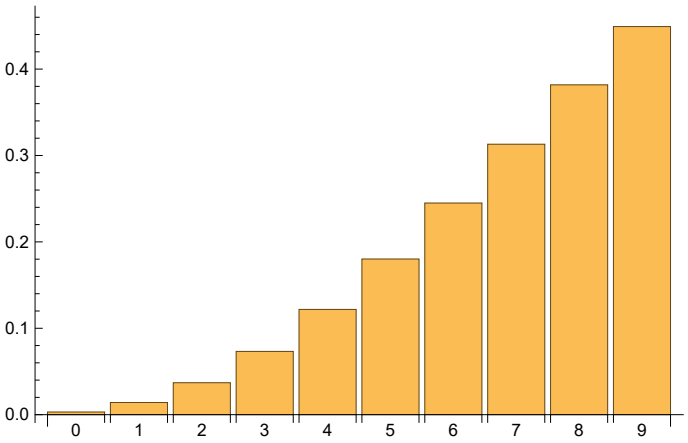


Figure 4.7: Example of empty queue probabilities q_k calculated for the FCTL model with $g = r = 10$ and binomial arrivals with $\mu_Y = 0.45$ and shape parameter $n = 2$.

4.5 Markov chain

Up until now we have only discussed the stationary analysis of the FCTL queue. We will now present a description of the FCTL queue in a similar way as we have done for the cycle-to-cycle model in Section 3.2. There, we showed how to construct a single transition matrix that enabled us to calculate the queue length distribution for any cycle, given the initial distribution, in a very efficient way. The major difference with that description is the fact that we now need a separate matrix for each phase due to the FCTL assumption. The states of the chain simply represent the value of $X_{k,n}$. We start our discussion by formulating matrices $P^{(g)}$ and $P^{(r)}$ that describe the transition to the next time step for the green and the red phase, respectively. The queue is governed by (4.1) during the green phase, from which it follows that a transition to a non-empy queue ($i > 0$) happens with probability

$$P_{ij}^{(g)} = \mathbb{P}(Y = i - j + 1), \qquad \text{for } 1 \leq j \leq i + 1. \tag{4.17}$$

The queue becomes empty when the last vehicle leaves and no arrivals happen, which gives

$$P_{01}^{(g)} = \mathbb{P}(Y = 0). \tag{4.18}$$

When the queue has already become empty, it stays empty due to the FCTL assumption, giving

$$P_{00}^{(g)} = 1. \tag{4.19}$$

The red phase allows for an even simpler description. From (4.2), it is easily seen that a transition to $i \geq 0$ happens with probability

$$P_{ij}^{(r)} = \mathbb{P}(Y = i - j), \qquad \text{for } 0 \leq j \leq i. \tag{4.20}$$

Transitions that are not described by the above expressions have probability zero. Let us now again slightly abuse notation to denote the distribution of $X_{k,n}$ as a column vector $(X_{k,n})_j = \mathbb{P}(X_{k,n} = j)$, then the evolution of this distribution can then simply be described as the matrix multiplications

$$X_{k+1,n} = P^{(g)} X_{k,n}, \tag{4.21}$$

for $k = 0, 1, \ldots, g - 1$ (green phase) and

$$X_{k+1,n} = P^{(r)} X_{k,n}, \tag{4.22}$$

for $k = g, g + 1, \ldots, c - 1$ (red phase).

In order to compute the evolution of the queue length numerically, we need to make sure that both transition matrices are finite, so we assume that the queue has a maximum length Q_{max}, exactly as we did in Section 3.2. For $i < Q_{max}$, expressions (4.17) and (4.20) still hold, but we now need to consider the case of overflow separately, which happens with probability

$$P_{Q_{max},j}^{(g)} = \mathbb{P}(j + Y \geq Q_{max} + 1), \qquad \text{for } j \geq 1, \tag{4.23}$$

during the green phase and with probability

$$P_{Q_{max},j}^{(r)} = \mathbb{P}(j + Y \geq Q_{max}), \tag{4.24}$$

during the red phase.

We checked our calculations by making sure that the above procedure leads to two proper stochastic matrices (columns summing to one), see Appendix C. The expected value for the first 100 time steps is shown in Figure 4.8. The discussion of the effect of Q_{max} from Section 3.2 applies equally well to the present model, so we omit it here.

The model may be regarded as a Markov chain if we also consider the current time step part of the state, i.e., the states are given by the tuples $(X_{k,n}, k, n)$. However, it is also possible to only consider the overflow queue (or 'cycle-to-cycle' evolution of any other time step), in which case the transition matrix is simply given by $(P^{(r)})^{c-g}(P^{(g)})^{g}$. Because this matrix needs to be computed only once, this may possibly help to speed up the calculations.

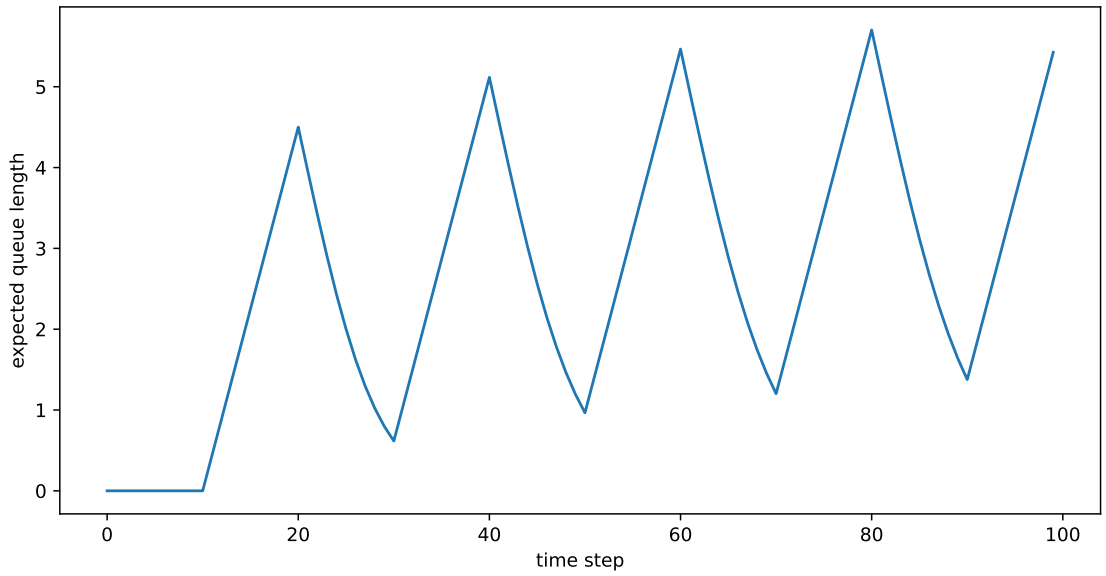


Figure 4.8: Expected queue length per time step computed with the FCTL model for $g = r = 10$ and $Q_{max} = 40$ with binomial arrivals with $\mu_Y = 4.5$ and shape parameter $n = 2$.

4.6 Discussion

The work done by van Leeuwaarden (2006) extends the classical analysis of Darroch (1964) for the FCTL model in two ways: (1) it allows us to compute the full distribution of the queue length and delay instead of only expected values, and (2) it supports general i.i.d. arrivals instead of certain classes of (compound) Poisson arrivals. However, some limitations still exist in the above described version of the FCTL model that we will shortly discuss here.

The analyis only considers steady-state behavior, so the framework is only applicable to under-saturated traffic conditions. Furthermore, the arrivals were considered to be identically distributed accross cycles. However, in reality, arrival rates are never constant, and periods of oversaturated traffic are no exception during rush hour, for example. It would be an interesting topic of further research to use real traffic measurements to assess if the approach of using non-stationary arrivals and departures described in Section 4.5 is usable for predictions or queue lengths.

The arrivals $Y_{k,n}$ are assumed to be independent, but this is not often the case in real traffic because of the position of a traffic light queue in a larger traffic network. Vehicles departing from an upstream intersection with a traffic light may form platoons, which causes a violation of the independence assumption for the next intersection. For these kinds of situations, a generalization of the FCTL queue has been proposed by Boon & van Leeuwaarden (2018), which allows for correlated arrivals. That means that $(Y_{1,n}, \ldots, Y_{c,n})$ can have any joint probability distribution. They study multiple intersections that are connected to each other in a series. In this setup, the output process of an intersection determines the input process of the next (downstream) intersection.

Another shortcoming is that the FCTL model uses time slots of fixed length, which is the time needed for a vehicle to leave a non-empty queue. Therefore, the interdeparture times are constant and deterministic. We observed (and has been observed by others, for example by Oblakova (2019), Table 4.1) that interdeparture times are very close to deterministic, see Section 5.2, but this was for homogeneous traffic. It may be interesting to see if this still holds for mixed traffic with freight trucks, for example.

Of course, the fixed-cycle traffic light model only considers traffic lights with fixed control. However, many modern traffic light systems in the Netherlands, for example, use some sort of sensors (induction loops under the road surface) to measure actual traffic demand. For these situations, the probabilistic framework discussed in Chapter 3 has also been extended to model dynamic control, see Viti & van Zuylen (2009).

Chapter 5

Microscopic simulation

This chapter will discuss how we may analyze the queueing behavior near a signalized intersection using microscopic simulation and we will compare this method to an analysis using the FCTL model from Chapter 4. We will discuss SUMO (Simulation of Urban MObility), which is a continuous-space, discrete-time microscopic traffic simulation software package. Initially developed by the German Aerospace Center (DLR), see Lopez *et al.* (2018), and now available as open-source software, it is used by practitioners to model and simulate realistic urban traffic networks. The package consists of a collection of separate tools. The `netedit` tool can be used to construct a road topology model and to define traffic lights. There is also a tool to construct a network from real maps in the OpenStreetMap dataset. The actual simulation program can be started with the `sumo` command line tool, which does not show any visual output. The `sumo-gui` tool provides a graphical user interface that visualizes the simulation, see Figure 5.1.

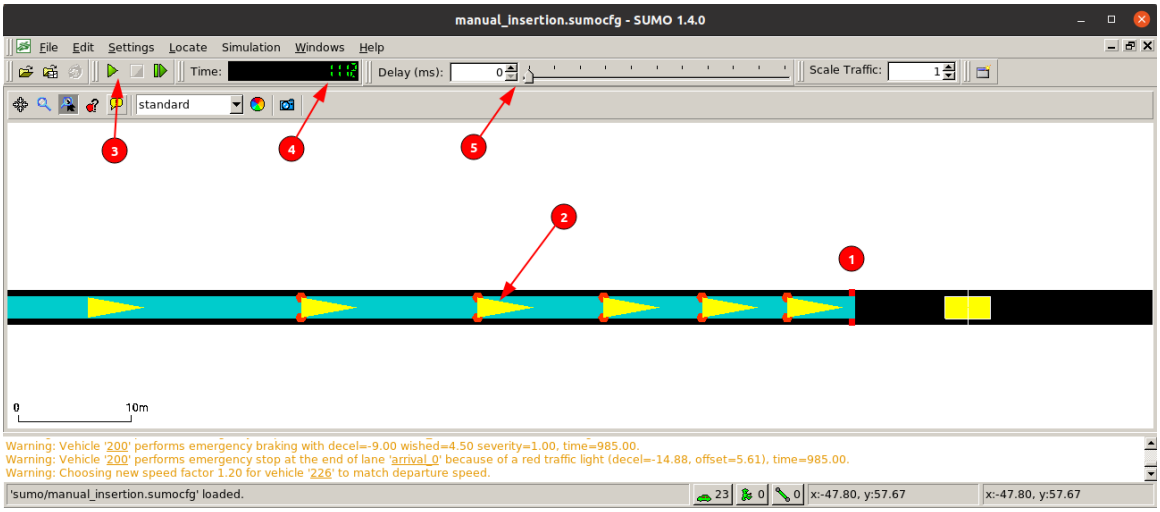


Figure 5.1: The graphical interface provided by `sumo-gui`. The traffic light is drawn in the visualization area as a vertical line (1) in the color of the traffic light state (currently red). Vehicles are shown as yellow triangles (2), with active brake lights drawn as two red circles. The user interface provides acces to various tools via the menus. There are also controls to start or step through the simulation one step at a time (3). The current time step is displayed (4), and the time between time steps may be set using the slider at (5). Without such a delay, the simulation would go so fast that distinguishing individual cars becomes difficult.

Figure 5.2: The simulation setup that was used for the model comparison. The horizontal axis has been truncated at the indicated locations for visibility reasons. Right after the traffic light at (2) is an E1 induction loop detector (2), visualized as a yellow rectangle, which is used to count the number of departures within a loop (used by the `departure-counter` tool). The long turquoise rectangle (3) represents an E2 lane area detector, which can measure the actual queue length, and spans from the beginning of the lane to the traffic light. The entry (4) and exit (5) detectors of the E3 detector are used to measure delay and are located at 400m and 700m, respectively.

5.1 Setup

In order to study the queueing behavior near a signalized traffic light, we constructed a very simple road model with a single lane on which a single traffic light is located, see Figure 5.2. The traffic light is located at 500 meter from the beginning of the lane, which is 800 meters long in total. Vehicles enter the system at the beginning of the lane and are removed from the system once they reach the end. In order to keep track of certain values during the simulation, SUMO requires us to specify one or more *detectors*.

There are several ways of collecting data from and interacting with the simulation. Measurements may be stored to XML files by providing flags to the simulation command. The online SUMO documentation contains a complete overview of the available measurements that can be requested. There is also a direct socket connection which may be used to receive real-time updates of the simulation status. On top of the low-level socket connection, the higher level TraCI (Traffic Control Interface) provides a way to programmatically read and alter settings at runtime.

Each step in the simulation is controlled by a Python script via TraCI, see Appendix D. In each step a decision can be made to insert a vehicle based on some probability p. We choose to insert vehicles at a cruise speed of $v = 14m/s$. Before the vehicle is actually inserted in the system, SUMO checks if there is enough space available. If not, then the insertion will be put in a so-called 'insertion queue'. This limits the maximum arrival rate in the system, which will be discussed further in Section 5.3. SUMO supports programming the traffic lights via `sumo-gui`, but for our purposes it was easier to change green and red times directly via TraCI.

5.2 Measuring departures

We will now briefly discuss the departure process of the different models that we have seen. First of all, in the FCTL model it is assumed that exactly one car can leave during a time slot. So there is a linear relationship between the length of the green phase and the number of departures. A similar assumption holds for the bulk service queue. We want to assess if this assumption can also be made for the microscopic model in SUMO. Note that this departure process may depend on the chosen car-following model. For our comparison, however, we stick with the default model in SUMO, which we discussed in Section 2.3.3.

We developed a tool `departure-counter`, see Appendix D.3, that measures the average number of vehicles that can leave during a given green time using the E1 induction loop detector in SUMO, see Figure 5.2. The tool first completely fills the lane with vehicles. After a fixed setup time the traffic light starts cycling between the given green time and a red time that is at least long enough to fill the lane again. The average number of departures is calculated over a predetermined number of cycles. The tool supports plotting this average number of departures against the specified green times, as can be seen in Figure 5.3, and also supports saving/loading results to file. We observe

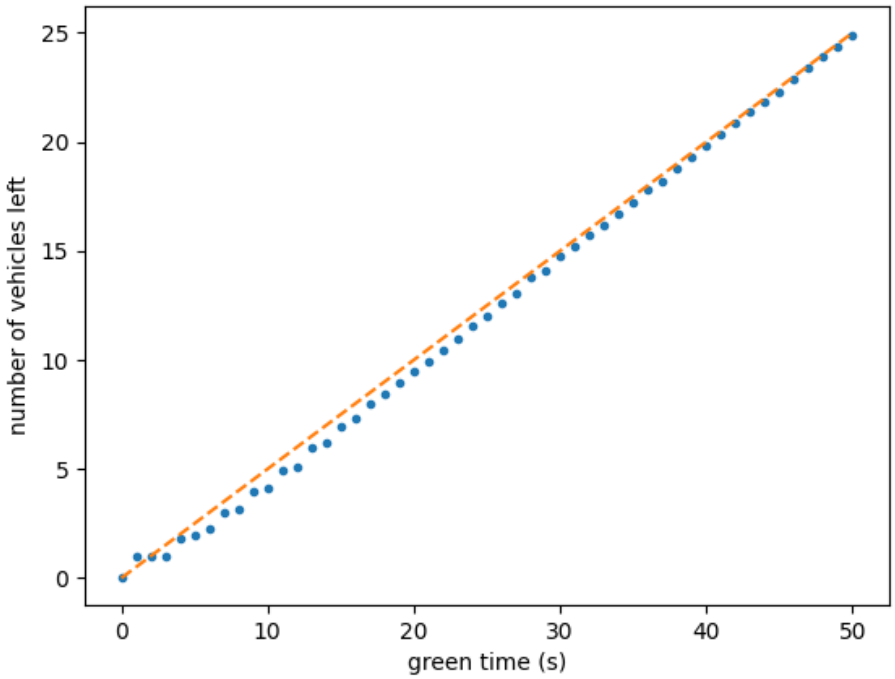


Figure 5.3: Average number of vehicles that could leave the traffic light queue during the given amount of green time. No yellow phase was added to the SUMO traffic light program. The orange reference line is the number of cars that would leave when each departure takes 2 seconds of time. Numbers are computed within one simulation only. The average was calculated over 100 cycles.

from the figure that the assumption of a linear relationship seems reasonable. However, for short green times, the number of departures is overestimated by the mathematical queueing models.

The linear relationship may also be interpreted as constant inter-departure times. We note that extensions have been proposed for the FCTL model to allow for varying inter-departure times. Oblakova (2019) proposed an extension of the FCTL queue where the first few inter-departures times are longer. To model driver distraction, they also allow randomness in the departures.

5.3 Measuring delay

Before we proceed with analyzing queue length, we will first consider delay in the SUMO model and in the FCTL model. For the FCTL model, we choose to have a red and green phase of 10 time slot each, which we will denote by $r_F = 10$ and $g_F = 10$, respectively. As we saw in Figure 5.3, one departure happens around every two time steps in the SUMO model, so we need twice as many time steps in the SUMO model, which we will denote by r_S and g_S. When we set $g_S = 20$, we find using `departure-counter` that on average 9.5 departures happen. Therefore, to make both models more equivalent for comparison, we set the green time in the SUMO model to $g_S = 21$, for which the average number of arrivals is found to be 9.96. We need to take this scaling into account when converting the arrival rate between both models. Because the SUMO model allows for an insertion each time step, we let the arrivals in the FCTL model be distributed as $Bin(2, p_F)$ random variables for some insertion probability parameter p_F. For the SUMO model, this insertion parameter then

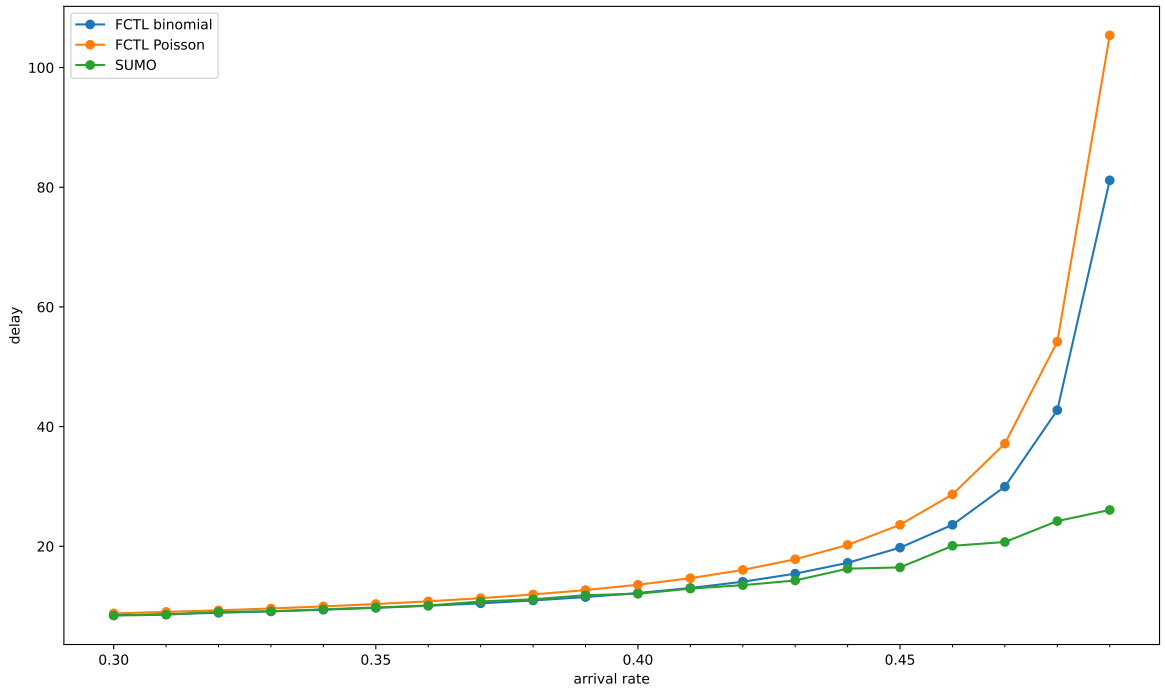


Figure 5.4: Delay in SUMO seconds computed with the FCTL model for binomial and Poisson arrivals compared to the delay found in the SUMO model after letting the simulation run for 100.000 SUMO seconds. The x-axis shows the insertion probability p_F. The equivalence degrades for higher arrival, probably due to the fact that it is not possible to create such high degrees of saturation in SUMO due to spatial constraints or speed limits. Note that the binomial arrival distribution shows a better fit than the Poisson arrivals, which is exactly what we expected.

becomes

$$p_S = \frac{r_F + g_F}{r_S + g_S} p_F \ . \tag{5.1}$$

When comparing the delay computed with the FCTL model with the delay found from the simulation, we also need to take this time slot scaling into account. Because the time slots in SUMO are supposed to represent seconds, we choose to express the delay in terms of these *SUMO seconds*, so the FCTL delay in SUMO seconds becomes

$$\mathbb{E}[D'_F] = \frac{(r_S + g_S)}{(r_F + g_F)} \mathbb{E}[D_F] \ . \tag{5.2}$$

Using the E3 entry-exit detector[1], see Figure 5.2, we can measure the average delay of vehicles in the SUMO simulation. For each setting of p_F, we ran the simulation for 100.000 SUMO seconds. We see in Figure 5.4 that both models compute very similar values for low degrees of saturation. For larger arrival rates, however, the SUMO model does not show the same explosion of delay that is characteristic for queueing models near saturation. We note that it is possible that setting a high arrival rate parameter p in SUMO does not necessary mean that this will be realized because of possible speed limitations and spatial constraints. This may be an argument in favor of the assumption of finite queueing space, as we saw in Section 3.2 and Figure 3.3 in particular.

[1] https://sumo.dlr.de/docs/Simulation/Output/Multi-Entry-Exit_Detectors_(E3).html

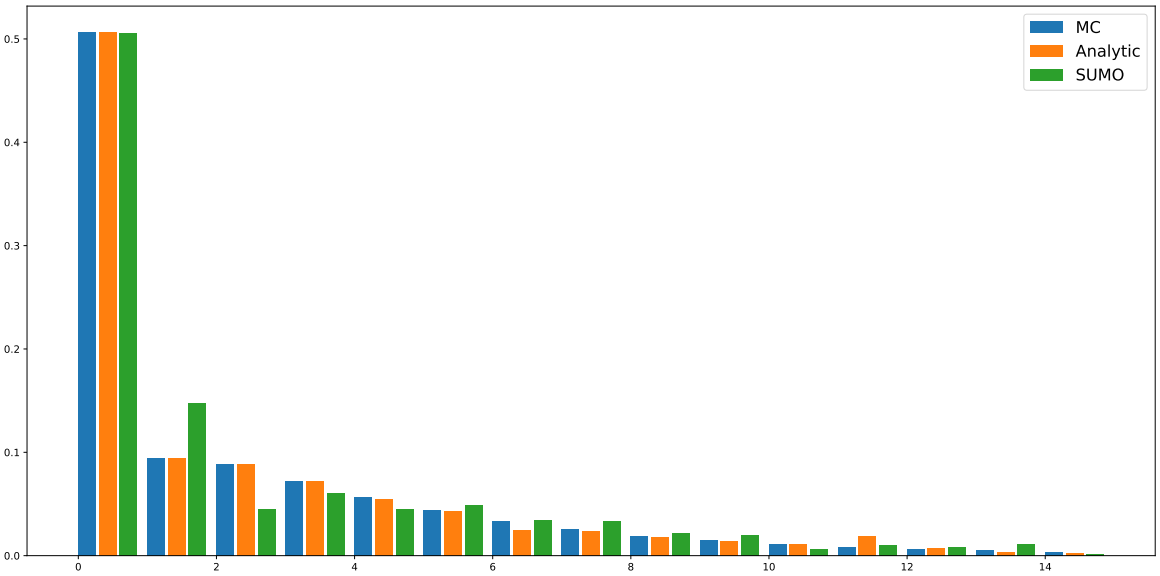


Figure 5.5: Comparison of stationary overflow queue for the FCTL model (via the stationary analysis and by computing the transient model for 1000 cycles) and as obtained from SUMO after 1000 cycles. The FCTL model has binomial arrivals with $\mu_Y = 0.45$ and shape parameter $n = 2$. The transient SUMO model has $Q_{max} = 40$. The same scaling of time slots is applied as described in Section 5.3.

5.4 Measuring queue length

We will now turn to analyzing the queue length using the FCTL model and the SUMO simulation. The classical queueing models can be used to calculate vertical queues, so numbers of vehicles. SUMO also supports various ways of retrieving a vertical queue estimation, which we will describe first.

The `sumo` `sumo-gui` command line tools support a `-queue-output` flag[2] that can be used to produce an XML file containing entries with the queue length and waiting time within a lane for a particular time step. However, the SUMO documentation does not specify how this queue length is defined. There is also an 'experimental queue length', which is defined as the number of vehicles up and including the last vehicle with a speed lower than 5 km/h. We could not find a method of changing this definition ourselves, which makes it hard to make a valid comparison with the FCTL model. To be able to inspect the queue length while the simulation is still running we used the socket connection that was mentioned in Section 5.1 to automatically update a graph of the queue length in each time step. Data is send over the socket as partial XML fragments, so we implemented a streaming XML parser that can read one time step at a time, see Appendix D.4. However, we noticed that redrawing the graph still takes a lot of time, so it is not possible to let the simulation run at high speeds (low time step delay).

Another method of measuring the queue length is provided by SUMO through E2 lanearea detectors[3]. As the name already indicates, this type of detector needs to be placed on a part of the lane and will produce data about jams in this area of the lane. Specifically, the length of the longest consecutive jam is reported (`jam max`) and the sum of all jams in the lane (`jam sum`, which is essentially the total number of halted vehicles). In contrast to the `-queue-output` flag output, the E2 detector gives us the ability to change the definition of when a vehicle is considered halting based on either speed or gap length by editing the `additional-files` configuration XML file[4].

Being able to change the definition of the vertical queue length in SUMO allows us to calibrate this definition to be in accordance with the FCTL queue. Although, in practise, it would be more

[2] https://sumo.dlr.de/docs/Simulation/Output/QueueOutput.html
[3] https://sumo.dlr.de/docs/Simulation/Output/Lanearea_Detectors_(E2).html
[4] https://sumo.dlr.de/docs/sumo.html#format_of_additional_files

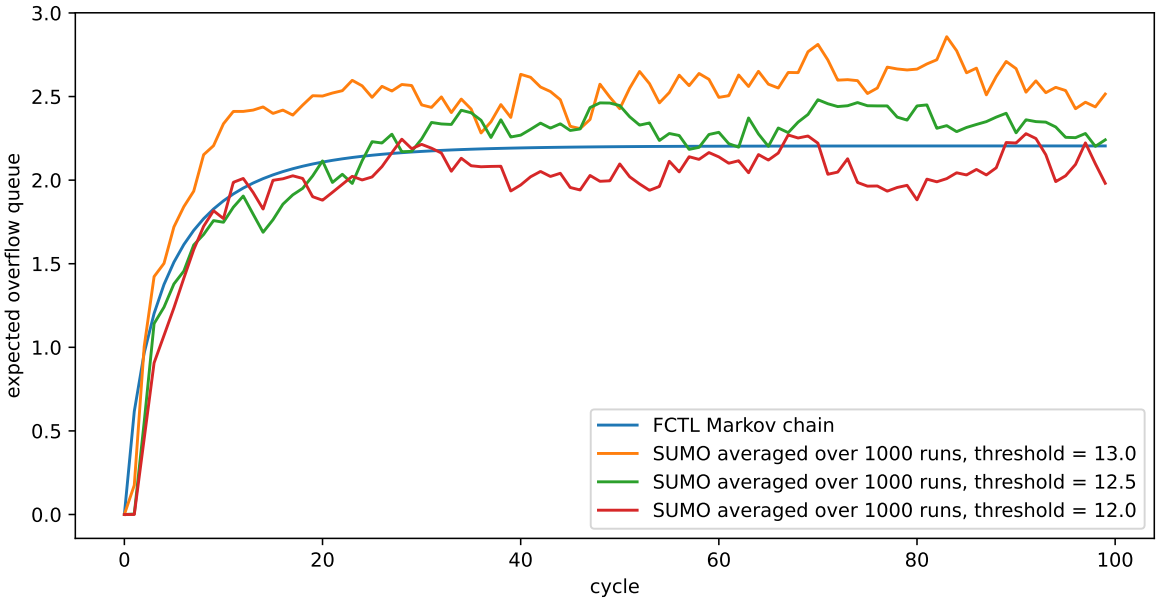


Figure 5.6: Expected overflow queue from FCTL Markov model and from SUMO by averaging over 1000 simulation runs for different values of the SUMO parameter *speed threshold*. All other parameters are the same as in Figure 5.5.

reasonable to calibrate the FCTL model based on the SUMO model, the former method at least allows us to compare the two models. For different settings of this definition, we compared the approximation of the stationary queue lenth, that we can obtain from SUMO by running the simulation for a long time, to the full distribution obtained for the FCTL model using calculations in Mathematica and an approximation using the transient FCTL calculations of Section 4.5. By manually inspection, we found that the following settings for the E2 lanearea detector

```
<laneAreaDetector id="lanearea_detector" lane="arrival_0"
    pos="0" endPos="500"
    freq="1" file="output/lanearea_counts.xml"
    speedThreshold="13"
/>
```

produced a reasonable similar distribution, as can be seen in Figure 5.5. We noticed that the distribution from our implementation of the stationary FCTL analysis differs from the approximation using the transient FCTL model. We believe that this is due to some numerical errors that occur in the stationary calculations, which may be related to the root-finding procedure. Unfortunately, we did not have time to investigate this issue further, which is why we included the transient FCTL approximation.

Vehicles are considered halting if they have a speed lower than the *speedThreshold* parameter. Furthermore, the *freq* parameter indicates the number of SUMO seconds over which the queue length measurements are computed. By settings this parameter to 1, we try to obtain the highest possible level of detail. This also means that we cannot measure the queue length at the exact end of the green phase. For the sake of this comparison, we choose to define the overflow queue as the number of vehicles in the first timeslot of the subsequent red phase.

After this calibration of SUMO parameters, we can compare the transient behavior of SUMO with the transient FCTL calculations. Figure 5.6 shows the transient expected queue length as computed with the FCTL model compared to the transient expected queue length computed using SUMO by averaging over 1000 simulation runs. We tried some different values for *speedThreshold*. It seems that after averaging over 1000 simulations, there is still a lot of variability left. Nevertheless, it seems that the speed at which the queue length converges to the stationary value is comparable for both models.

5.5 Discussion

We noticed that a lot of 'emergency stops' happen during the SUMO simulation due to the absence of the amber phase in the static traffic light program. These emergency stops happen because the traffic light switches to red immediately, so the driver has no way of adjusting the speed in due time. It would be interesting to see if adding the amber phase makes the behavior of the SUMO simulation more regular, possibly resulting in a smaller variance of the queue length at a particular time step.

Chapter 6

Conclusions and further work

The main theme of this study was the comparison of different models for the queue length at signalized intersections with fixed control. In order to understand the field of traffic modeling in a general sense, we also studied some history of traffic flow modeling and the development of microscopic traffic simulation software. After that, we turned to a comparison of the model proposed by Viti & van Zuylen (2010), the bulk service queue, the FCTL queue and a microscopic traffic simulation in SUMO.

The original description of the cycle-to-cycle model that Viti and van Zuylen provide is not very clear on details, so we first provided an alternative description which tries to make all assumptions clear. This led to the realization that this cycle-to-cycle model generalizes the classical bulk service queue by introducing a maximum queue size and allowing non-stationary arrivals and departures accross cycles. The bulk service queue is a well-known model in queueing theory, so we used existing results to calculate the stationary queue length numerically, which allowed us to study the effect of different settings for this maximum queueing space. We also briefly discussed the within-cycle model of Viti and van Zuylen, which extends the cycle-to-cycle model to give a continuous-time description of the queue length.

We presented existing results on the stationary analysis of the FCTL model. We included a description of the issues of root-finding and the inversion of probability generating functions, which are required for obtaining numerical results for the FCTL queue and the bulk service queue. We saw that the FCTL model may be viewed as an extension of the bulk service queue. We showed using numerical comparisons that the bulk service queue provides an upperbound for the stationary queue length at the end of cycles in the FCTL model. We found that this upperbound gets tighter when the number of time slots in the green phase gets larger. It would be interesting to see if this relationship can be made more formal.

Viti and van Zuylen note that their cycle-to-cycle model and their within-cycle model allow for arrivals and departures with varying rates. We recognize that this is also possible for the transient FCTL model. We did not consider this possibility in the present study. An interesting use case for this would be to model the effect of peak periods in a similar way as Oblakova (2019) has done.

We compared the delay obtained from a microscopic simulation model in SUMO with the delay that we get from the stationary analysis of the FCTL queue. We found that the FCTL formula gives a reasonable accurate prediction of the simulated delay for low arrival rates. For high arrival rates, we conjectured that the desired rate of arrival is not achieved in the SUMO model due to speed restrictions and spatial constraints. This could be verified by measuring the actual demand in the SUMO model. It would be interesting to see if the FCTL model with finite buffer size gives a more accurate prediction.

The transient description of the queue length with the FCTL model is similar to the cycle-to-cycle model of Viti and van Zuylen. This transient FCTL model is a Markov chain when it is restricted to the overflow queue. We compared this model to the queue in SUMO and found that the speed of convergence to the stationary limit is comparable. One issue that remains is the calibration of the parameter *speed threshold* that defines the vertical queue length measured by SUMO. We showed

that this calibration may be done by comparing the stationary distribution obtained from SUMO by running a large number of cycles and from the stationary analysis for the FCTL model. We note that in practical applications, it is desirable to do the calibration between SUMO and FCTL the other way around: given the *speed threshold*, find the parameters of the FCTL model that produce the best fit. Finding a more systematic approach for this calibration is absolutely necessary when we want to use the FCTL model in dynamic control applications and seems an interesting topic for further research. We note that fitting models to real data is an important topic in practical application in general. We also briefly touched upon this subject in our discussion of the cycle-to-cycle model. In combination with real-time traffic data, simple models could allow model-based optimization for use in traffic control systems. Simple models that can be computed fast are necessary to allow the control system to quickly adapt to the current situation.

Bibliography

Abate, J., Choudhury, G. L., & Whitt, W. 2000. An Introduction to Numerical Transform Inversion and Its Application to Probability Models. *Pages 257–323 of: Computational Probability.* Boston, MA: Springer US.

Adan, I.J.B.F., Leeuwaarden, van, J.S.H., & Winands, E.M.M. 2005. *On the application of Rouché's theorem in queueing theory.* SPOR-Report: reports in statistics, probability and operations research. Technische Universiteit Eindhoven.

Akçelik, R. 1980. *Time-Dependent Expressions for Delay, Stop Rate and Queue Length at Traffic Signals.* Tech. rept. AIR 367-1. Australian Road Research Board, Vermont South, Australia.

Bailey, N. T. J. 1954. On Queueing Processes with Bulk Service. *Journal of the Royal Statistical Society. Series B (Methodological)*, **16**(1), 80–87.

Bieker-Walz, L., Behrisch, M., Junghans, M., & Gimm, K. 2017. Evaluation of car-following-models at controlled intersections. *ESM 2017 European Simulation and Modelling Conference*, **31**, 247–251.

Boon, M. A. A., & van Leeuwaarden, J. S. H. 2018. Networks of fixed-cycle intersections. *Transportation Research Part B: Methodological*, **117**, 254–271.

Brackstone, M., & McDonald, M. 1999. Car-following: a historical review. *Transportation Research Part F: Traffic Psychology and Behaviour*, **2**(4), 181–196.

Brockmeyer, E., Halstrom, H. L., & Jensen, A. 1948. The Life and Works of A. K. Erlang. *In: Transactions of the Danish Academy of Technical Sciences.* Akademiet for de Tekniske Videnskaber.

Daganzo, C. F. 1994. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Transportation Research Part B: Methodological*, **28**(4), 269–287.

Darroch, J. N. 1964. On the Traffic-light Queue. *The Annals of Mathematical Statistics*, **35**(1), 380–388.

Ding, C., & Song, S. 2012. Traffic Paradoxes and Economic Solutions. *Journal of Urban Management*, **1**(1), 63–76.

Geroliminis, N., & Sun, J. 2011. Hysteresis phenomena of a Macroscopic Fundamental Diagram in freeway networks. *Transportation Research Part A: Policy and Practice*, **45**(9), 966–979.

Greenshields, B. D. 1934. The photographic method of studying traffic behavior. *Pages 382–399 of: Proceedings of the 13th annual meeting of the highway research board.*

Greenshields, B. D., Bibbins, J. R., Channing, W. S., & Miller, H. H. 1935. A study of traffic capacity. *Pages 448–477 of: Proceedings of the 14th annual meeting of the highway research board.*

Janssen, A. J. E. M., & van Leeuwaarden, J.S.H. 2005. Analytic computation schemes for the discrete-time bulk service queue. *Queueing Systems*, **50**(2-3), 141–163.

Kendall, D. G. 1953. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, **24**(3), 338–354.

Kimber, R., & Hollis, E. M. 1979. *Traffic queues and delays at road junctions.* Tech. rept. 909. Transport and Road Research Laboratory.

Knoop, V. L. 2017. *Macroscopic Traffic Flow Modelling.* Reader from and for graduate Students Course by TRAIL research school.

Krauss, S. 1998. *Microscopic modeling of traffic flow: investigation of collision free vehicle dynamics.* Tech. rept. 98-08. DLR Deutsches Zentrum fur Luft– und Raumfahrt.

Lighthill, M. J., & Whitham, G. B. 1955. On kinematic waves II. A theory of traffic flow on long crowded roads. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, **229**(1178), 317–345.

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

Miller, A. 1968. *The capacity of signalized intersections in Australia.* Tech. rept. Australian Road Research Board.

Nagel, K., & Schreckenberg, M. 1992. A cellular automaton model for freeway traffic. *Journal de Physique I*, **2**(12), 2221–2229.

Oblakova, A. 2019. *Queueing models for urban traffic networks.* Ph.D. thesis, University of Twente.

Richards, P. I. 1956. Shock Waves on the Highway. *Operations Research*, **4**(1), 42–51.

Sommer, C., German, R., & Dressler, F. 2011. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing (TMC)*, **10**(1), 3–15.

Timmerman, R. W., & Boon, M.A.A. 2021. Platoon forming algorithms for intelligent street intersections. *Transportmetrica A: Transport Science*, **17**(3), 278–307.

van Leeuwaarden, J. S. H. 2006. Delay analysis for the fixed-cycle traffic-light queue. *Transportation Science*, **40**(2), 189–199.

van Leeuwaarden, J.S.H. 2005. *Queueing models for cable access networks.* Ph.D. thesis, Technische Universiteit Eindhoven.

van Wageningen-Kessels, F., van Lint, H., Vuik, K., & Hoogendoorn, S. 2015. Genealogy of traffic flow models. *EURO Journal on Transportation and Logistics*, **4**(4), 445–473.

Varga, A. 2010. OMNeT++. *Pages 35–59 of:* Wehrle, K., Güneş, M., & Gross, J. (eds), *Modeling and Tools for Network Simulation.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Viti, F., & van Zuylen, H. J. 2009. The Dynamics and the uncertainty of queues at fixed and actuated controls: A probabilistic approach. *Journal of Intelligent Transportation Systems: Technology, Planning, and Operations*, **13**(1), 39–51.

Viti, F., & van Zuylen, H. J. 2010. Probabilistic models for queues at fixed control signals. *Transportation Research Part B: Methodological*, **44**(1), 120–135.

Wiedemann, R. 1974. *Simulation des Strassenverkehrsflusses*. Tech. rept. Institute for Traffic Engineering, University of Karlsruhe.

Wu, N., & Brilon, W. 1990. Delays at Fixed-Time Traffic Signals Under Time-Dependent Traffic Conditions. *Traffic Engineering and Control*, **31**(12).

Zhang, H. M. 1999. A mathematical theory of traffic hysteresis. *Transportation Research Part B: Methodological*, **33B**(1), 1–23.

Appendices

The following appendices show a selection of the scripts and SUMO definition files that we used for the numerical work presented in this report. The transient calculations for the bulk service queue and the FCTL model have been implemented using Python in a Jupyter notebook. These notebooks may be exported directly to LaTeX files[1], which we have directly included here. We did not include the calculations for the stationary analysis that we implemented using Mathematica, because we believe that they do not provide more insight than the formulas that we have discussed within the text.

[1] https://tex.stackexchange.com/a/314785

Appendix A

Cellular automaton

Python implementation of the cellular automaton of Nagel & Schreckenberg (1992), which was discussed in Section 2.3.2.

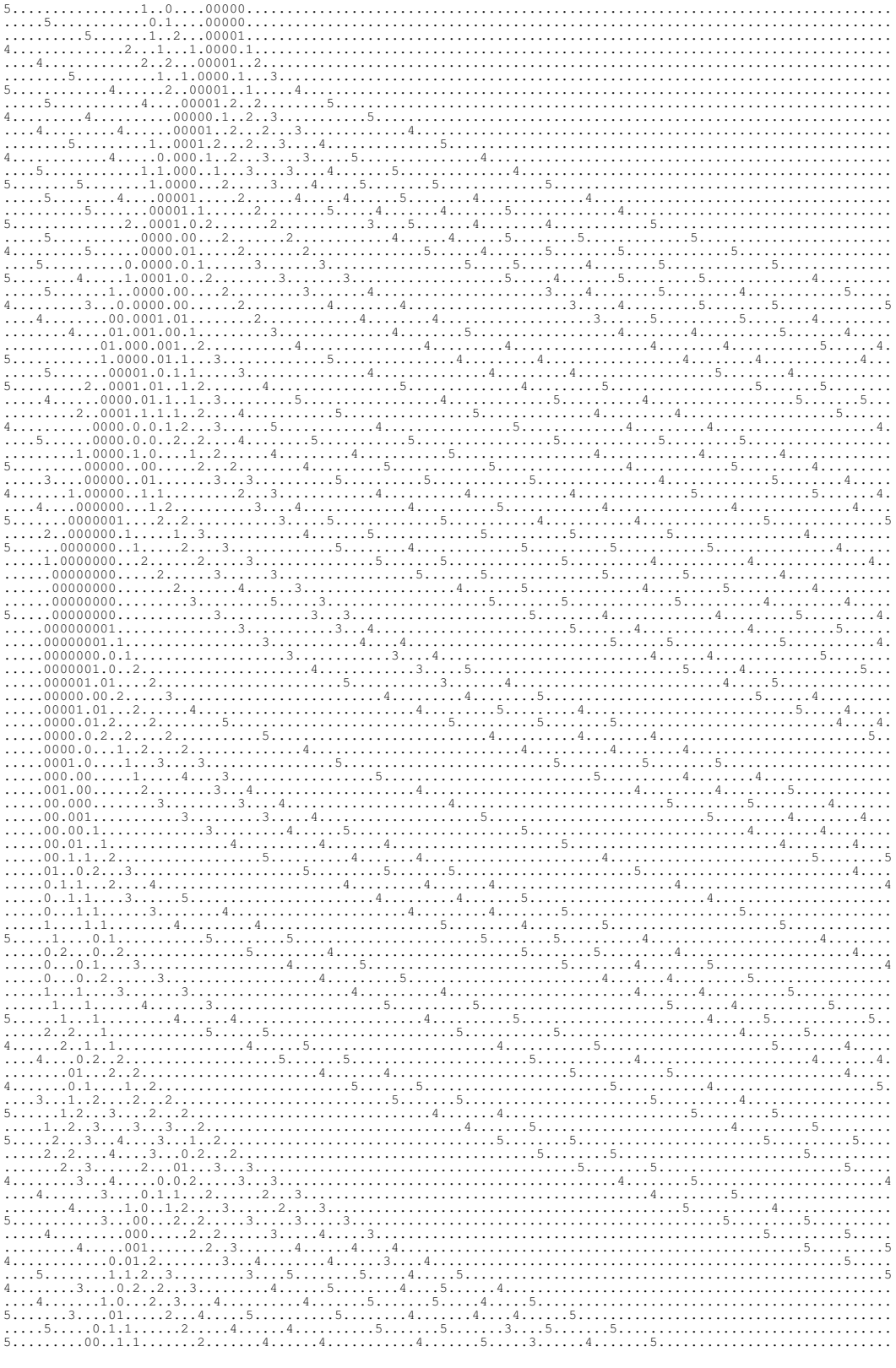
```
L = 110 # length of the lane in slots
v_max = 5 # maximum cruise speed of vehicles
p = 0.45 # probability of randomly decreasing velocity
t_max = 100 # number of timesteps to compute

p_insert = 0.8 # insertion probability
```

```
from random import random

def print_positions(positions):
    '''Prints the current state of the lane by showing
    the velocity of the vehicle in a slot, or 0 when
    no vehicle is present.'''
    for i in range(L):
        v = positions[i]
        if v == -1:
            print('.', end='')
        else:
            print(v, end='')
    print('')
```

```
def distance_to_next(positions, i, v):
    """For the car at i driving at v, compute min(j, v + 2),
    where j is the distance to the next car."""
    for j in range(1, v + 3): # j \in [1, v + 2]
        if i+j >= L: # if not in the system any more
            j = v + 2 # no car ahead
            break
        elif positions[i+j] >= 0: # if car in front
            break

    return j
```

```
def add_car(t, positions):
    # make sure the first position is free...
    if random() < p_insert and positions[0] == -1:
        v0 = 5
        j = distance_to_next(positions, 0, v0)

        if j >= v0 + 1: # ...and also make sure there enough head-space
            positions[0] = v0
```

```
# clear all vehicles
positions = [-1 for x in range(L)]

# create an artificial traffic jam
positions[20] = 0
positions[17] = 0
positions[25] = 0
positions[26] = 0
positions[27] = 0
positions[28] = 0
positions[29] = 0

for t in range(t_max):
    # add cars
    add_car(t, positions)

    new_positions = [-1 for x in range(L)]

    # update velocities
    for i in range(L):
        v = positions[i]
        if v >= 0: # if there is a car at this position

            j = distance_to_next(positions, i, v)

            if j <= v:
                # decelerate to match speed of vehicle ahead
                v = min(j - 1, v)

            if j > v + 1 and v < v_max:
                # accelerate
                v = v + 1

            if random() < p and v > 0:
                # randomly decrease velocity
                v = v - 1

            # we may create an artificial bottle-neck
#             if 45 <= i < 55:
#                 v = min([1, v])

            # assign the computed velocity
            positions[i] = v

            # compute next position
            if i+v < L: # let cars disappear when they pass position L-1
                new_positions[i+v] = v

    # show current state t with correct velocities
    print_positions(positions)

    # update all positions
    positions = new_positions
```



Appendix B

Transient bulk service queue

B.1 Transient queue length in bulk service queue

Instead of using the probability generating function approach for the bulk service queue, we can also compute the transient and stationary distribution of the (overflow) queue by iterating through the cycles one by one. This Markov chain is generalized by the cycle-to-cycle model of Viti and Van Zuylen that allows departures to be generally distributed, which makes the construction of the transition matrix a little bit more involved. We have to truncate the possible values for the arrivals and departures in order to calculate the transition matrix.

Like in the paper of Viti and Van Zuylen, we let the cycle start with the green phase.

B.2 Experiment loading

```
%run init.ipynb
```

```
Stored 'e0' (Experiment)
Stored 'e1' (Experiment)
```

```
%store -r e1
e = e1
e.f_cycle = dill.loads(e.f_cycle) # Python does not support pickling of lambda's by
    default
e.F_cycle = dill.loads(e.F_cycle)
e.g_cycle = dill.loads(e.g_cycle)
```

B.3 Calculating the queue length distribution for each cycle using a matrix

The evolution of the overflow queue distribution can easily be computed for each cycle by multiplying with the transition matrix.

```
def P_transition():
    """Construct the transition matrix for going to the next cycle."""

    P = np.zeros(shape=(e.Q_max + 1, e.Q_max + 1))

    # underflow (i==0)
    for j in range(0, e.Q_max + 1):
        for a in range(0, e.a_max + 1):
            for d in range(j + a, e.d_max + 1):
                P[0, j] += e.f_cycle(a) * e.g_cycle(d)
```

```
11
12      # overflow (i==Q_max)
13      for j in range(0, e.Q_max + 1):
14          for a in range(e.Q_max - j, e.a_max + 1):
15              for d in range(0, j + a - e.Q_max + 1):
16                  P[e.Q_max, j] += e.f_cycle(a) * e.g_cycle(d)
17
18
19      # all in between
20      for j in range(0, e.Q_max + 1):
21          for i in range(1, e.Q_max):
22              for a in range(i - j, e.a_max + 1):
23                  P[i, j] += e.f_cycle(a) * e.g_cycle(j + a - i)
24
25
26      return P
27
28
29  P_trans = P_transition()
30
31  # Use this to check if the matrix is indeed stochastic
32  #np.matmul(np.ones(e.Q_max + 1), P_trans)
```

We can now simply start multiplying the initial queue length distribution with the transition matrix for the desired number of cycles. The entry Q[t,k] of the 2d array that results from this represents the probability that the queue length is k at

```
1  Q = np.zeros(shape=(e.cycles + 1, e.Q_max + 1))
2  init_length = 0
3  Q[0,init_length] = 1 # initial queue length (or distribution if you want)
4
5  for c in range(e.cycles):
6      Q[c+1] = np.matmul(P_trans, Q[c])
7
8  Q_bulk = Q
9  %store Q_bulk
```

```
Stored 'Q_bulk' (ndarray)
```

B.4 Evolution of full distribution per cycle

We can visualize the evolution of the full distribution for some cycles by plotting a heatmap.

```
1  fig, ax = plt.subplots(1,1, figsize=(25,5))
2  ax.imshow(Q[:100].transpose(), origin='lower');
```

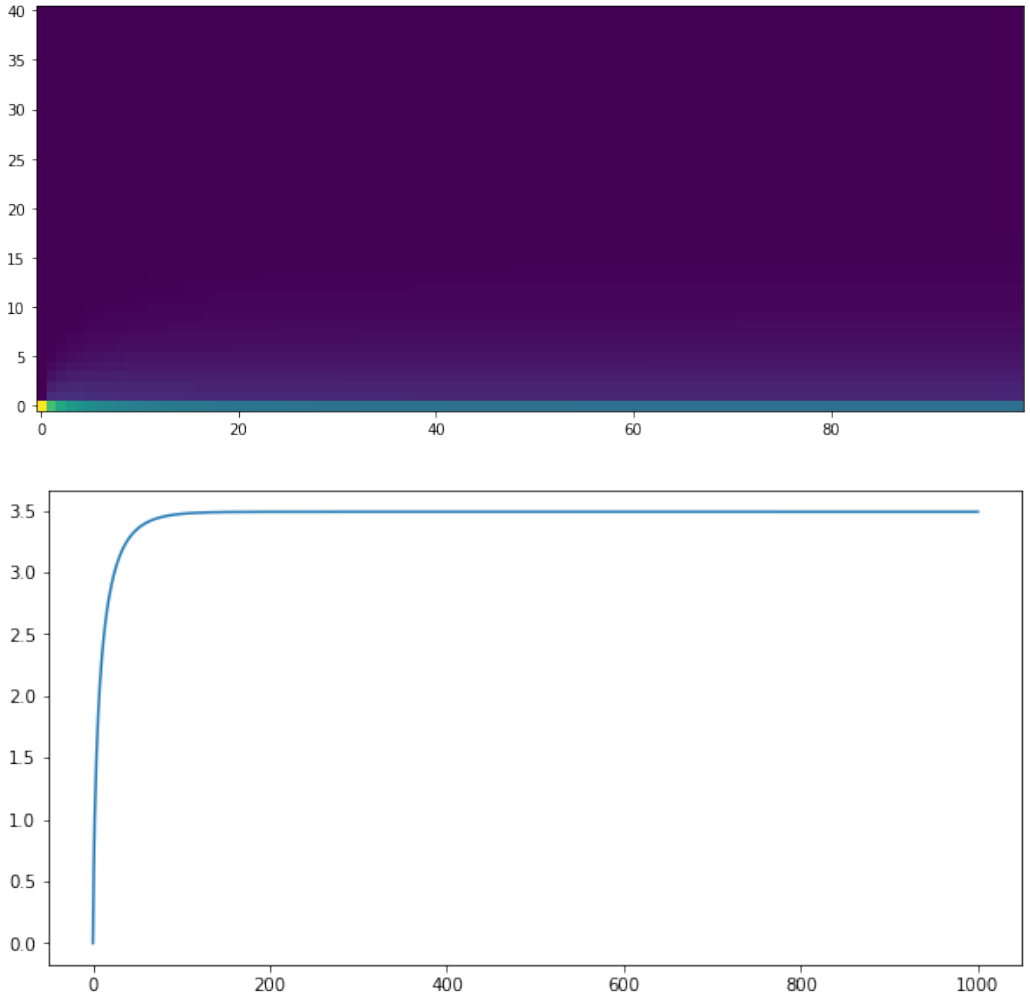
B.5 Expected value of overflow queue per cycle

Instead of the full distribution, we may also plot the expected value of the overflow queue for each cycle.

```
1  expected_overflow_queue = np.matmul(Q, np.arange(0, e.Q_max + 1))
```

```
1  fig, ax = plt.subplots(1,1, figsize=(10,5))
2  ax.plot(expected_overflow_queue);
```

The expected overflow queue of the last cycle that we computed is an approximation for the expected stationary overflow queue length, so this may be compared to the value found by the stationary bulk service analysis, or the stationary FCTL analysis.




```
bulk_stationary = expected_overflow_queue[-1]
%store bulk_stationary
```

```
Stored 'bulk_stationary' (float64)
```

B.6 Queue length within a cycle

Given the overflow queue for all cycles, we may now extend this with the within-cycle model of Viti and van Zuylen.

```
from math import floor

def P_Q(e, Qo, delta_t, f, F):
    """Compute the queue length distribution at time delta_t since the start of the
    cycle.

    :param Qo: the distribution of the overflow queue at the start of the cycle
    :param delta_t: time in seconds since the start of the current cycle
    :param f: arrival pdf f(a, dt) = P('# arrivals happened after dt seconds' == a)
    :param F: arrival cdf F(a, dt) = P('# arrivals happened after dt seconds' <= a)
    """

```

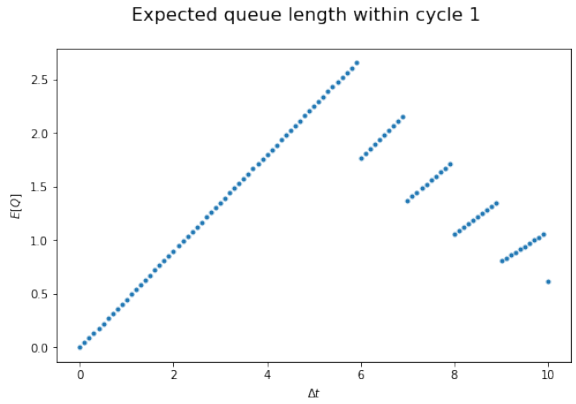
```
12      assert(0 <= delta_t <= e.cF)
13
14      def get_Q_red_end():
15          if P_Q.Q_red_end is None: # check if cache has already been created
16              P_Q.Q_red_end = P_Q_red(e, Qo, e.rF, f, F) # first get distribution after
        red phase
17          return P_Q.Q_red_end
18
19      if 0 <= delta_t <= e.rF:
20          # red phase
21          return P_Q_red(e, Qo, delta_t, f, F)
22
23      elif e.rF < delta_t <= e.cF:
24          # green phase
25
26          # note that we need to transform the time
27          return P_Q_green(e, get_Q_red_end(), delta_t - e.rF, f, F)
28
29      else:
30          raise ValueError
31
32  P_Q.Q_red_end = None # cacheing value as a function attribute
33
34
35  def P_Q_red(e, Qo, delta_t, f, F):
36      """Compute the distribution of the queue length during the red phase.
37
38      :param Qo: the distribution of the overflow queue at the start of the cycle
39      """
40
41      P = np.empty(shape=(e.Q_max + 1))
42
43      # underflow and general case
44      for i in range(0, e.Q_max):
45          P[i] = sum(
46              Qo[j] * f(i-j, delta_t)
47
48              for j in range(0, i+1)
49          )
50
51      def f_helper(j):
52          # P(a >= Q_max - j) = 1 - P(a < Q_max - j) = 1 - F_a(Q_max - j - 1)
53          return 1 - F(e.Q_max - j - 1, delta_t)
54
55      # overflow
56      P[e.Q_max] = sum(
57          Qo[j] * f_helper(j)
58          for j in range(0, e.Q_max+1)
59      )
60
61      return P
62
63  def P_Q_green(e, Q_red_end, delta_t, f, F):
64      """Compute the distribution of the queue length during the green phase.
65
66      :param Q_red_end: the distribution at tau+tr.
67      """
68
69      P = np.empty(shape=(e.Q_max + 1))
70
71      d_max = floor(e.sS * delta_t)
72
73      # underflow
74      P[0] = sum(
75          Q_red_end[j] * f(a, delta_t)
76
77          for j in range(0, d_max+1)
78            for a in range(0, d_max - j +1)
```

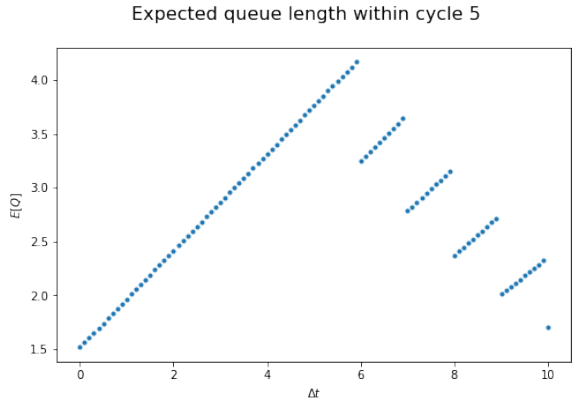
```
    )

    # general case
    for i in range(1, e.Q_max):
        j_max = min([i + d_max, e.Q_max])

        P[i] = sum(
            Q_red_end[j] * f(i + d_max - j, delta_t)

            for j in range(0, j_max+1)
        )

    def f_helper(j):
        # P(a >= Q_max + d_max - j) = 1 - P(a < Q_max + d_max - j) = 1 - F_a(Q_max + d_max - j - 1)
        return 1 - F(e.Q_max + d_max - j - 1, delta_t)

    # overflow
    P[e.Q_max] = sum(
        Q_red_end[j] * f_helper(j)
        for j in range(0, e.Q_max+1)
    )

    return P
```

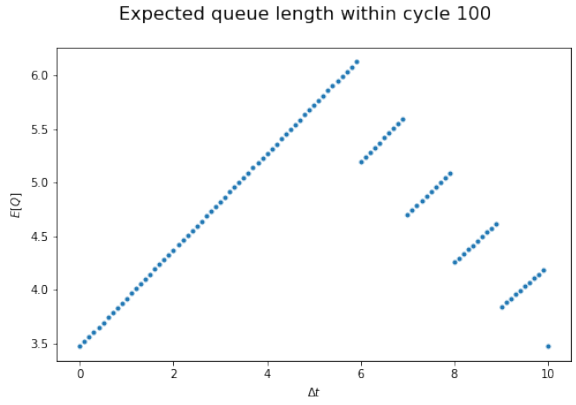
B.6.1 Plot within a single cycle

```
def plot_within_cycle(n):
    # the distribution at the start of the n'th cycle
    Qo = Q[n - 1, :]

    # the start of cycle n
    tau = (n - 1) * e.cF # seconds

    # arrivals within the cycle
    f = lambda a, dt: poisson.pmf(a, dt * e.aF)
    F = lambda a, dt: poisson.cdf(a, dt * e.aF)

    # DONT FORGET TO CLEAR CACHE
    P_Q.Q_red_end = None # cacheing value, function attribute

    x_data = np.arange(0, e.cF + 0.1, 0.1)
    distrs = [P_Q(e, Qo, delta_t, f, F) for delta_t in x_data]

    y_expected = [np.matmul(Q, np.arange(0, e.Q_max + 1)) for Q in distrs]

    fig, ax = plt.subplots(1,1, figsize=(8,5))
    ax.plot(x_data, y_expected, '.')
    ax.set_ylabel('$E[Q]$')
    ax.set_xlabel('$\Delta t$')
    fig.suptitle(f'Expected queue length within cycle {n}', fontsize=16);
    plt.savefig(f'./images/within_cycle_{n}.pdf')
```

```
plot_within_cycle(1)
plot_within_cycle(5)
plot_within_cycle(100)
```



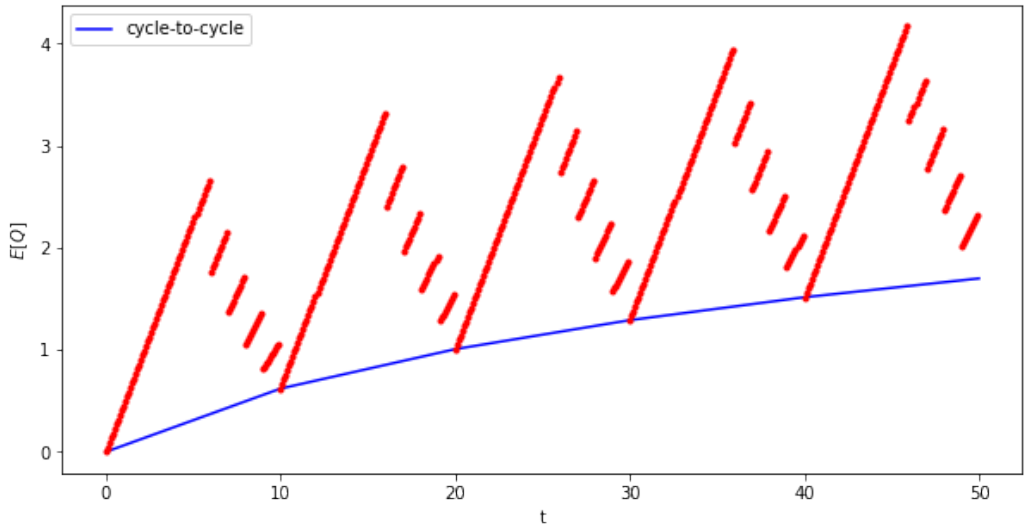




B.6.2 Verify with overflow queue

```
n = 5

fig, ax = plt.subplots(1,1, figsize=(10,5))
x_cycle = np.arange(0, (n + 1) * e.cF, e.cF)
ax.plot(x_cycle, expected_overflow_queue[:n+1], color='blue', label='cycle-to-cycle')
    ;

for i in range(n):
    # the distribution at the start of the n'th cycle
    Qo = Q[i, :]

    # the start of cycle i+1
    tau = i * e.cF # seconds
```

```
13
14      # arrivals within the cycle
15      f = lambda a, dt: poisson.pmf(a, dt * e.aF)
16      F = lambda a, dt: poisson.cdf(a, dt * e.aF)
17
18      # DONT FORGET TO CLEAR CACHE
19      P_Q.Q_red_end = None # cacheing value, function attribute
20
21      x_data = np.arange(tau, tau + e.cF, 0.1)
22      distrs = [P_Q(e, Qo, x - tau, f, F) for x in x_data]
23
24      y_expected = [np.matmul(Q, np.arange(0, e.Q_max + 1)) for Q in distrs]
25
26      ax.plot(x_data, y_expected, '.', color='red')
27
28  ax.set_xlabel('t')
29  ax.set_ylabel('$E[Q]$')
30  ax.legend()
31
32  plt.savefig(f'./images/within_first_{n}_cycles.pdf')
```



Appendix C

Transient FCTL queue

C.1 Transient queue length in FCTL queue

Instead of using the probability generating function approach, we can also compute the transient and stationary distribution of the (overflow) queue by iterating through the cycles one by one. Like the cycle-to-cycle model of Viti and Van Zuylen, this approach could also allows us to define non-stationary arrivals, but then we need to recompute the transition matrix if the arrival rate changes.

Note that we use the same conventions as in the models of Viti and Van Zuylen that a cycle starts with the green phase. This allows us to compare the transient behavior of both models better.

C.2 Experiment loading

```
%run init.ipynb
```

```
Stored 'e0' (Experiment)
Stored 'e1' (Experiment)
```

```
%store -r e0
e = e0
e.f = dill.loads(e.f) # Python does not support pickling of lambda's by default
e.F = dill.loads(e.F)
e.g = dill.loads(e.g)
```

C.3 Calculating the queue length distribution for each time step using two matrices

Because the queue lenght distribution evolves differently in the green and red phase, we define a transition matrix for each of them.

```
def P_green():
    """Get transition matrix for steps in green phase."""

    P = np.zeros(shape=(e.Q_max + 1, e.Q_max + 1))

    # underflow
    P[0,0] = 1
    P[0,1] = e.f(0)

    # overflow
    for j in range(1, e.Q_max+1):
```

```
        P[e.Q_max, j] = 1 - e.F(e.Q_max - j)

    # all in between
    for i in range(1, e.Q_max):
        for j in range(1, (i + 1)+1):
            P[i,j] = e.f(i - j + 1)

    return P

def P_red():
    """Get transition matrix for steps in red phase."""

    P = np.zeros(shape=(e.Q_max + 1, e.Q_max + 1))

    # overflow
    for j in range(e.Q_max):
        P[e.Q_max,j] = 1 - e.F(e.Q_max - j - 1)

    # once we have a full queue, it cannot grow anymore
    P[e.Q_max, e.Q_max] = 1

    # not yet overflow
    for i in range(0, e.Q_max):
        for j in range(0, i + 1):
            P[i,j] = e.f(i - j)

    return P


Pg = P_green()
Pr = P_red()
# Use this to check if the matrices are indeed stochastic
#np.matmul(np.ones(e.Q_max + 1), Pg), np.matmul(np.ones(e.Q_max + 1), Pr)
```

Now calculate the distribution for each time step by multiplying with each of these matrices in an alternating way. The entry Q[t,k] of the 2d array that results from this represents the probability that the queue is k at time step t.

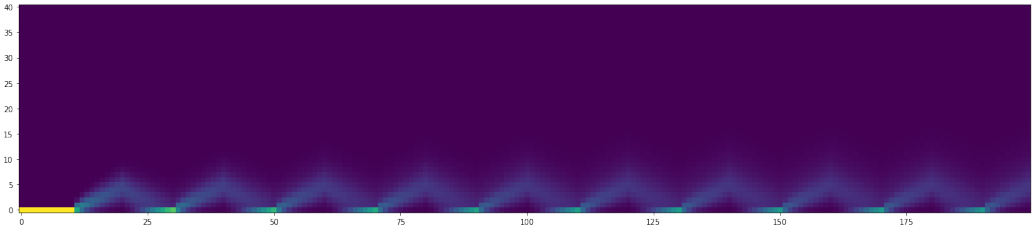
```
Q = np.zeros(shape=(e.cycles * e.cF + 1, e.Q_max + 1))
init_length = 0
Q[0,init_length] = 1 # initial queue length (or distribution if you want)

for c in range(e.cycles):
    for t in range(c*e.cF, c*e.cF + e.gF): # cycle starts with green phase...
        Q[t+1] = np.matmul(Pg, Q[t])

    for t in range(c*e.cF + e.gF, (c + 1)*e.cF): # ...and then red phase
        Q[t+1] = np.matmul(Pr, Q[t])

Q_FCTL = Q
%store Q_FCTL
```

```
Stored 'Q_FCTL' (ndarray)
```

C.4 Evolution of full distribution per timestep

We can visualize the evolution of the full distribution for some timesteps by plotting a heatmap.

```
fig, ax = plt.subplots(1,1, figsize=(25,5))
ax.imshow(Q[0:200].transpose(), origin='lower')
```

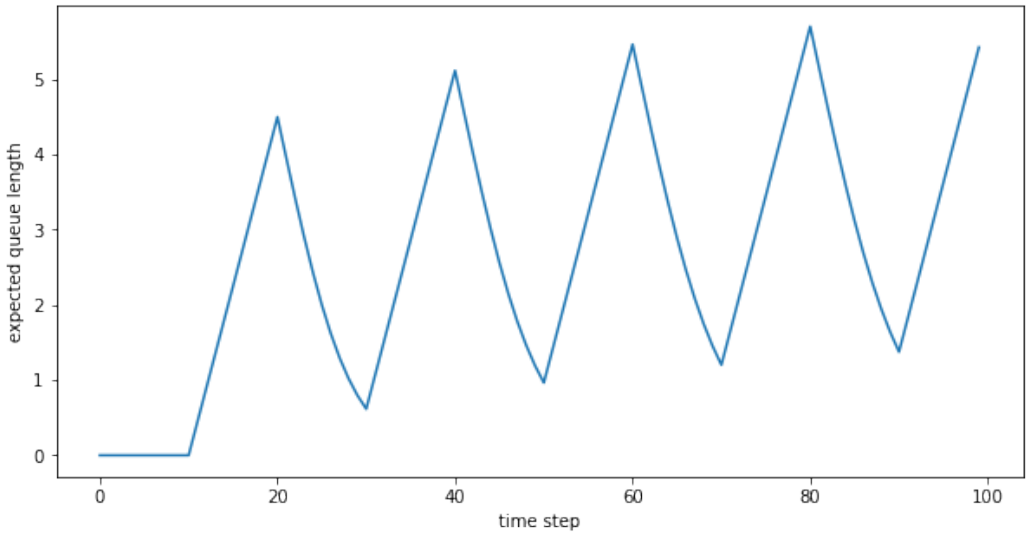


C.5 Expected value per time step

We can also show the evolution of the expected value for some timesteps. We calculate the expected value by multiplying with a row vector with entries 0,1,...,Q_max.

```
expected = np.matmul(Q, np.arange(0, e.Q_max + 1))
```

```
fig, ax = plt.subplots(1,1, figsize=(10,5))
ax.plot(expected[:100]);
ax.set_xlabel('time step')
ax.set_ylabel('expected queue length')
plt.savefig('images/FCTL_MC_timesteps.pdf')
```



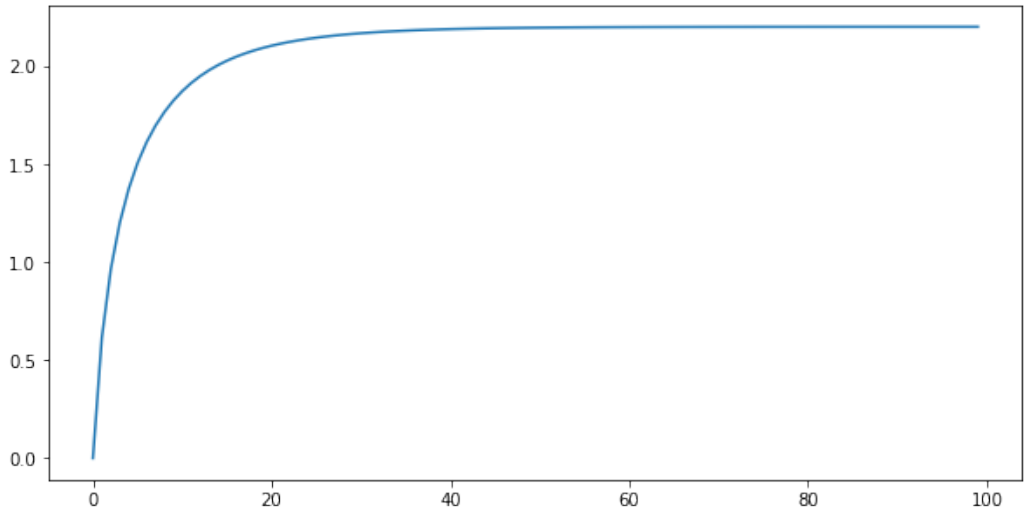
C.6 Expected value of overflow queue per cycle

Instead of the expected value, we may also choose to only plot the timesteps just after the green phase, which gives us the evolution of the overflow queue length.

```
expected_overflow_queue = np.take(expected, range(e.gF, e.cycles * e.cF, e.cF))
FCTL_expected_overflow = expected_overflow_queue
%store FCTL_expected_overflow
```

```
Stored 'FCTL_expected_overflow' (ndarray)
```

```
fig, ax = plt.subplots(1,1, figsize=(10,5))
ax.plot(expected_overflow_queue[:100]);
```



The expected overflow queue of last timestep that we computed is an approximation for the expected stationary overflow queue length, so this may be compared to the value found by the stationary FCTL formula.

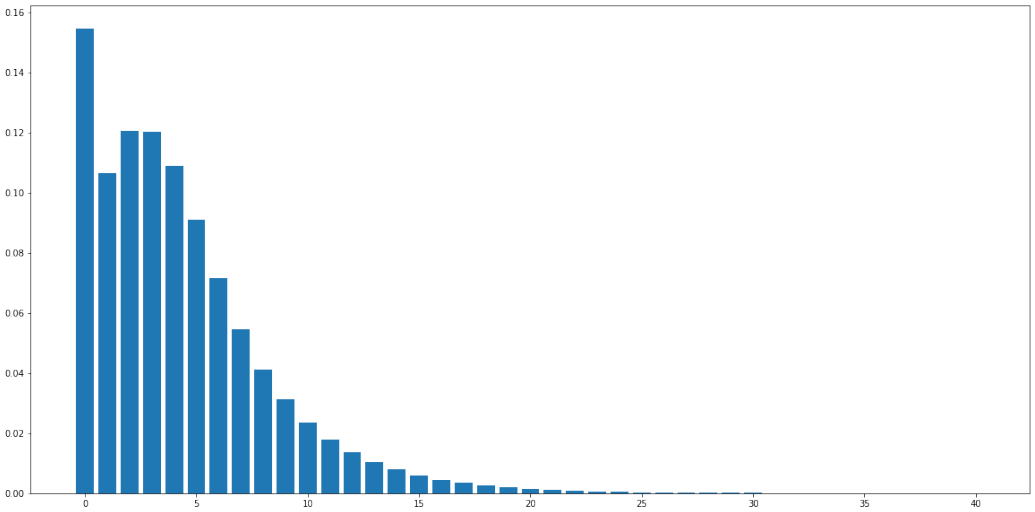
```
expected_overflow_queue[-1]
```

```
2.2046148205221536
```

C.7 Average distribution over all time steps

We may average the distributions over all time steps, giving the distribution of the queue length at an arbitrary point during a cycle.

```
averaged_distribution = np.matmul(np.ones(shape=(e.cycles * e.cF + 1, 1)).transpose()
    , Q) / (e.cycles * e.cF + 1)
averaged_distribution = averaged_distribution.flatten().tolist()

fig, ax = plt.subplots(1,1, figsize=(20,10))
ax.bar(list(range(e.Q_max + 1)), averaged_distribution);
```



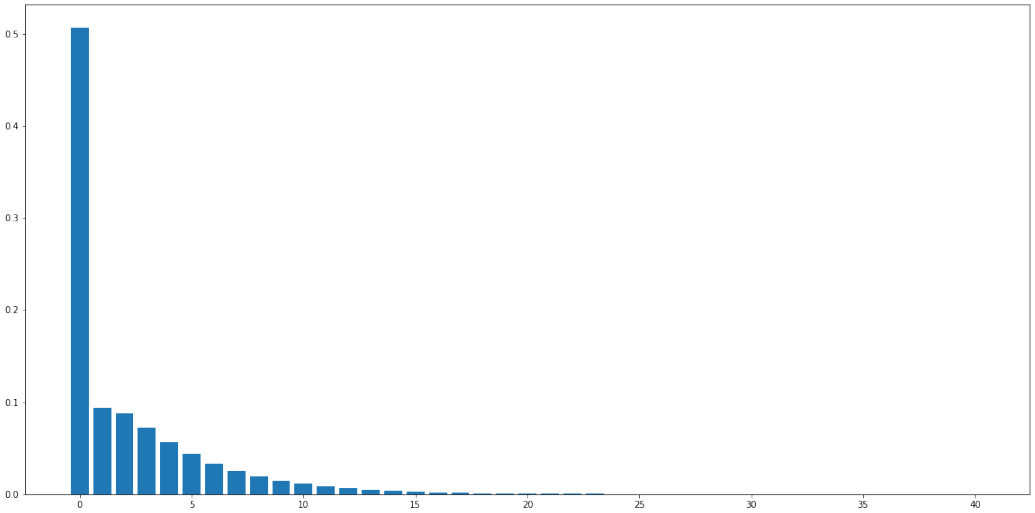
C.8 Overflow queue of last cycle

We consider the distribution of the overflow queue of the last cycle to obtain an approximation for the distribution of the stationary overflow queue. This distribution may be compared to the distribution that we find using the FCTL model, or to the distribution that we can find through the SUMO model. Note that in the latter case, we may need to calibrate the definition of queue length in SUMO in order to get similar results. However, in practice, it would be more relevant if we could calibrate the FCTL model, given a definition of queue length for the SUMO model.

```
FCTL_overflow_queue = Q[e.gF + (e.cycles - 1) * e.cF]
%store FCTL_overflow_queue
```

```
Stored 'FCTL_overflow_queue' (ndarray)
```

```
fig, ax = plt.subplots(1,1, figsize=(20,10))
ax.bar(list(range(e.Q_max + 1)), FCTL_overflow_queue);
```



C.9 Percentiles

We can also compute percentiles if we want. For example, the probability that the stationary overflow queue is larger than 20 is given by

```
1 - sum(FCTL_overflow_queue[0:20])
```

```
0.002995292846616371
```

Appendix D

SUMO script snippets

We wrote a collection of Python scripts and SUMO definition files for the numerical work that has been described in this report, which we cannot all include here. The two most important scripts for starting and controlling a SUMO simulation run are presented below. The `Experiment` data class is used to convert the different parameters for the FCTL and the SUMO model. The 'start' script starts a SUMO simulation and controls it via the TraCI interface to obtain measurements. We have also included the 'departure-counter' tool, because we think that it may be directly used by others. Other measurements are written directly to file. We wrote parsers for these files to create the figures in this report. We also included the script that we used for showing a graph of the live queue length while a simulation is running.

D.1 Experiment data class

```
from dataclasses import dataclass
from typing import Callable

@dataclass
class Experiment:
    """Data class containing parameters for SUMO model and for slotted models for
    comparison."""

    ### Arrivals ###

    # pdf and cdf of arrivals per time slot
    f:Callable[[int], float] = None
    F:Callable[[int], float] = None


    ### Departures ###

    g:Callable[[int], float] = None


    ### General parameters ###

    cycles: int = 100


    ### Slotted model parameters ###

    # number of green time slots in the slotted models (FCTL, Markov chain)
    gF:int = 10
    # number of red time slots in the slotted models (FCTL, Markov chain)
    rF:int = 10

    # mean of number of arrivals per timeslot
```

```
33      aF:float = 0.4
34
35      # maximum queue length
36      # used to model the occurance of spillback
37      # and this simplifies further calculations, because we only have a finite matrix
        of transition probabilities
38      Q_max:int = 40
39
40      # maximum arrivals and departures within one time step
41      # this makes the probabilistic calculations feasible
42      a_max:int = 40
43      d_max:int = 1
44
45
46      ### SUMO parameters ###
47
48      # saturation flow (veh/s) in SUMO
49      sS:float = 0.5 # 0.5 corresponds roughly to the 2s that are needed to depart in
        SUMO
50
51      # length of the effective green phase (s)
52      gS:int = 20
53      # length of the effective red phase (s)
54      rS:int = 21
55
56      # length in seconds of a SUMO time step
57      step_length = 1
58
59
60      ### Computed parameters ###
61
62      # timing
63
64      @property
65      def cS(self):
66          """total signal cycle (s) in SUMO"""
67          return self.gS + self.rS
68
69      @property
70      def cF(self):
71          """total number of time slots"""
72          return self.gF + self.rF
73
74      @property
75      def simulation_time(self):
76          """total number of SUMO seconds to simulate"""
77          return self.cycles * self.cS
78
79      # arrivals
80
81      @property
82      def aS(self):
83          """arrival flow rate (veh/s) in SUMO"""
84          return self.aF * (self.rF + self.gF) / (self.cS)
85
86      @property
87      def pS(self):
88          """insertion probability for each SUMO timestep"""
89          return self.aS * self.step_length
90
91
92      ### Miscellaneous ###
93
94      @property
95      def yS(self):
96          """fraction of green time in the cycle"""
97          return self.gS / self.cS
98
```

```
    @property
    def capacityS(self):
        """capacity of the signal (veh/s)"""
        return self.sS * self.yS

    @property
    def xS(self):
        """degree of saturation"""
        return self.aS / (self.sS * self.yS)

    @property
    def x0S(self):
        """Value above which the overflow queue can be considered nonzero."""
        return 0.67 + self.sS * self.gS / 600
```

D.2 Script for starting SUMO simulation runs

This is the main entry point of interacting with a running SUMO simulation via the TraCI interface.

```
import os, sys, argparse
from random import random
from math import ceil

from experiment import Experiment


def simulate(e: Experiment, gui=False, live_plot=False, count_departures=False):
    # we need to import python modules from the $SUMO_HOME/tools directory
    if 'SUMO_HOME' in os.environ:
        tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
        sys.path.append(tools)
    else:
        sys.exit("please declare environment variable 'SUMO_HOME'")
    import traci

    sumoBinary = 'sumo-gui' if gui else 'sumo'
    queue_output_file = 'localhost:1338' if live_plot else './sumo/network/output/
    queue_lengths.xml'
    fcd_output_file = './sumo/network/output/fcd.xml'


    ### Parameters ###
    simulation_time = e.simulation_time
    step_length = e.step_length

    red_seconds = e.rS
    green_seconds = e.gS

    initialSpeed = 14 # initial speed of vehicles that enter the lane

    p = e.pS # parameter for the Bernoulli arrivals
    ###################


    red = ceil(red_seconds / step_length) # number of steps during which the traffic
    light is red
    green = ceil(green_seconds / step_length) # number of steps during which the
    traffic light is green
    if red != int(red_seconds / step_length):
        raise ValueError('red_seconds is not a multiple of step_length')
    if green != int(green_seconds / step_length):
        raise ValueError('green_seconds is not a multiple of step_length')

    step = 0
    last_step = ceil(simulation_time / step_length)
    n_cycles = last_step // (red + green) + 1
```

```
45
46      routeID = 'route_0'
47      vehID = 0
48
49      # total number of vehicles left (for counting departures per cycle)
50      departures_in_cycle = [0 for x in range(n_cycles)]
51
52      # start sumo as a subprocess and connect to it
53      traci.start([sumoBinary, '-c', 'sumo/manual_insertion.sumocfg', '--queue-output',
         queue_output_file, '--fcd-output', fcd_output_file])
54
55      # at this point, the user needs to click the 'step' button before we activate
        step 0
56      print('start simulation, begin of step=0')
57      print(f'n_cycles={n_cycles}')
58
59      traci.trafficlight.setPhase('1', 0) # traffic light is initially red
60      traci.simulationStep() # we need a step to get at time=0, strange enough
61
62      while step < last_step:
63          cycle = step // (red + green) + 1 # first cycle is 1
64
65          if count_departures:
66              # count departures during last step
67              departures_in_cycle[cycle - 1] += traci.inductionloop.
        getLastStepVehicleNumber('departure_loop')
68
69          # insert new vehicle for next step
70          if random() < p:
71              traci.vehicle.add(str(vehID), routeID, departSpeed=str(initialSpeed))
72              vehID += 1
73
74          # set the traffic light
75          if step % (red + green) in range(red):
76              traci.trafficlight.setPhase('1', 0) # red
77          elif step % (red + green) - red in range(green):
78              traci.trafficlight.setPhase('1', 1) # green
79
80          step += 1
81          traci.simulationStep()
82
83      traci.close()
84
85      if count_departures:
86          print(2*'\n')
87          print(f'average departures per cycle={sum(departures_in_cycle) / n_cycles}')
88
89
90  def get_options():
91      parser = argparse.ArgumentParser(description='Start SUMO simulation')
92      parser.add_argument('--gui', action='store_true', help='show SUMO gui')
93
94      parser.add_argument('--cycles', type=int, default=100, help='number of cycles to
        simulate')
95      parser.add_argument('--step-length', type=float, default=1, help='time step
        length passed to sumo')
96      parser.add_argument('--arrival-rate', type=float, default=0.3, help='arrival rate
         a from FCTL model, from which probability p for inserting vehicles is computed')
97
98      return vars(parser.parse_args())
99
100
101 if __name__ == '__main__':
102     options = get_options()
103
104     e = Experiment()
105     e.rF = 10
106     e.gF = 10
```

```
    e.rS = 20
    e.gS = 21 # not exactly 20 to make sure that around 10 departures happen on
    average

    e.aF = options['arrival_rate']

    e.cycles = options['cycles']
    e.step_length = options['step_length']

    print(f'starting simulation with {e.simulation_time} SUMO seconds')
    print(f'insertion probability p={e.pS}')

    simulate(e, gui=options['gui'], live_plot=False)
```

D.3 Tool `departure-counter`

The tool that we discussed in Section 5.2.

```
import os, sys, argparse, pickle
import matplotlib.pyplot as plt

# we need to import python modules from the $SUMO_HOME/tools directory
if 'SUMO_HOME' in os.environ:
    tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
    sys.path.append(tools)
else:
    sys.exit("please declare environment variable 'SUMO_HOME'")

from sumolib import checkBinary
import traci


def run(green_time, cycles=1):
    """execute the TraCI control loop"""
    setup_time = 50
    red_time = 1 + 2 * green_time

    last_step = setup_time + cycles * (red_time + green_time)
    end_time = 10 # to make sure that all vehicles cross the induction loop

    step = 0
    cycle = 1

    departures = 0 # total number of vehicles left

    print(f'start simulation, with {cycles} cycles, green={green_time}, red={red_time
    }')

    traci.trafficlight.setPhase('1', 0) # start with red

    traci.simulationStep() # we need a step to get at time=0, strange enough

    while traci.simulation.getMinExpectedNumber() > 0 and step < last_step + end_time
    :
        departures += traci.inductionloop.getLastStepVehicleNumber('departure_loop')

        # modulo cycle size, counting from setup_time
        within_cycle_step = (step - setup_time) % (red_time + green_time)

        inner_loop = setup_time <= step < last_step

        if inner_loop and within_cycle_step == 0:
            traci.trafficlight.setPhase('1', 0)
            # print(f'swithing to red in step {step}')

        if inner_loop and within_cycle_step == red_time:
```

```
47              traci.trafficlight.setPhase('1', 1)
48              # print(f'swithing to green in step {step}')
49
50          if not inner_loop:
51              traci.trafficlight.setPhase('1', 0)
52
53          step += 1
54          cycle = (step - setup_time) // (red_time + green_time) + 1
55          # print(f'now showing begin of step={step}, cycle={cycle}, inner_loop={
    inner_loop}')
56          traci.simulationStep()
57
58      traci.close()
59
60      average = departures / cycles
61
62      print(2*'\n')
63      print(f'last_step={step} average departures={average}')
64
65      # average departures per cycle
66      return average
67
68
69  def get_options():
70      parser = argparse.ArgumentParser(description='Measure number of departures for a
    given green time')
71      parser.add_argument('--load', action='store_true', help='do not simulate, but
    load previous values')
72      parser.add_argument('--gui', action='store_true', help='show SUMO gui')
73      parser.add_argument('--data-file', default='plot_green_departures', help='input/
    output file were results are stored and loaded')
74      parser.add_argument('--show-plot', action='store_true', help='show plot')
75      parser.add_argument('--image', help='save plot to this file')
76
77      # parser.add_argument('--step-length', type=float, default=1, help='time step
    length passed to sumo')
78      parser.add_argument('--cycles', type=int, default=1, help='number of cycles to
    average over')
79      parser.add_argument('--green', type=int, help='specific green time')
80      return vars(parser.parse_args())
81
82
83  # this is the main entry point of this script
84  if __name__ == "__main__":
85      options = get_options()
86
87      if options['load']:
88          print(f"opening file {options['data_file']}")
89          (x, y) = pickle.load(open(options['data_file'], 'rb'))
90      else:
91          # this script has been called from the command line. It will start sumo as a
92          # server, then connect and run
93          if options['gui']:
94              sumoBinary = checkBinary('sumo-gui')
95          else:
96              sumoBinary = checkBinary('sumo')
97
98          # step_length = options['step_length']
99          # print(f'step length: {step_length}')
100
101         # perform the experiments
102         if options['green'] is not None:
103             x = [options['green']]
104         else:
105             x = range(50 + 1)
106         y = []
107         for green_time in x:
108             # start sumo as a subprocess and connect to it
```

```
109             traci.start([sumoBinary, "-c", "simple_queue.sumocfg"]) #, "--step-length
    ", str(step_length)])
110
111             # run the simulation
112             y.append(run(green_time, cycles=options['cycles']))
113
114         print(f"saving to file {options['data_file']}")
115         pickle.dump((x,y), open(options['data_file'], 'wb'))
116
117
118     y_default = [0.5 * i for i in x]
119
120     # plot the results
121     fig, ax = plt.subplots()
122     ax.plot(x, y, '.')
123     ax.plot(x, y_default, '--')
124     ax.set_xlabel('green time (s)')
125     ax.set_ylabel('number of vehicles left')
126
127     if options['show_plot']:
128         plt.show()
129
130     if options['image'] is not None:
131         print(f"saving plot to {options['image']}")
132         plt.savefig(options['image'])
```

D.4 Live queue length graph (streaming XML parser)

We shortly mentioned the possible use case of this script in Section 5.4. Note that the socket connection must also be enabled in the 'start' script by setting `liveplot=True`.

```
1  import socket
2  import xml.sax
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from matplotlib.ticker import MultipleLocator
6
7
8  class StreamHandler(xml.sax.handler.ContentHandler):
9
10     lastEntry = None
11     lastName  = None
12
13     def __init__(self, data_callback):
14         self.data_callback = data_callback
15
16     def startElement(self, name, attrs):
17         print('startElement')
18         self.lastName = name
19         if name == 'data':
20             self.lastEntry = {'data': { 'attrs': attrs }}
21         elif name != 'queue-export':
22             self.lastEntry[name] = { 'attrs': attrs }
23
24     def endElement(self, name):
25         if name == 'lane':
26             timestep = self.lastEntry['data']['attrs']['timestep']
27
28             self.data_callback({
29                 'timestep': timestep,
30                 'queueing_length': self.lastEntry['lane']['attrs']['queueing_length'
    ],
31                 'queueing_length_experimental': self.lastEntry['lane']['attrs']['
    queueing_length_experimental'],
32             })
```

```
33          elif name == 'data':
34              self.lastEntry = None
35          elif name == 'queue-export':
36              raise StopIteration
37
38
39  if __name__ == '__main__':
40      """Plot the current queue length using the direct socket connect
41      to the running SUMO simulation."""
42
43      # necessary, because we must fix the scale of the graph immediately
44      max_t = 200
45
46      t = np.arange(max_t)
47      q = np.zeros((max_t,))
48      q_exp = np.zeros((max_t,))
49
50      plt.ion()
51
52      fig = plt.figure()
53      ax = fig.add_subplot(111)
54      ax.set_ylim([0, 100]) # queue length range in meters
55
56      q_plot, = ax.plot(t, q)
57      q_exp_plot, = ax.plot(t, q_exp)
58
59      ax.set(xlabel='timestep', ylabel='queue length',
60          title='Queue length at intersection')
61
62      ax.xaxis.set_major_locator(MultipleLocator(10))
63      ax.xaxis.set_major_formatter('{x:.0f}')
64      ax.xaxis.set_minor_locator(MultipleLocator(1)) # no minor tick labels
65
66      ax.yaxis.set_major_locator(MultipleLocator(10))
67      ax.yaxis.set_major_formatter('{x:.0f}')
68      ax.yaxis.set_minor_locator(MultipleLocator(1)) # no minor tick labels
69
70
71      def data_callback(data):
72          q_length = float(data['queueing_length'])
73          print(f'queueing length: {q_length}')
74
75          timestep = int(float(data['timestep']))
76          q[timestep] = q_length
77          q_exp[timestep] = float(data['queueing_length_experimental'])
78
79          q_plot.set_ydata(q)
80          q_exp_plot.set_ydata(q_exp)
81
82          # see https://stackoverflow.com/questions/4098131/how-to-update-a-plot-in-
        matplotlib
83          fig.canvas.draw()
84          fig.canvas.flush_events()
85
86
87      port = 1338
88      with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
89          s.bind(("localhost", port))
90          s.listen(1)
91
92          conn, addr = s.accept()
93          with conn.makefile('b', buffering=0) as f:
94              parser = xml.sax.make_parser()
95              parser.setContentHandler(StreamHandler(data_callback))
96
97              parser.parse(f)
```