Freeway control under stochastic capacity in a connected vehicle environment based on a dynamic bargaining game approach

Heshami, Seiran

UNIVERSITY OF CALGARY

Freeway control under stochastic capacity in a connected vehicle environment based on a dynamic bargaining game approach

by

Seiran Heshami

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN CIVIL ENGINEERING

CALGARY, ALBERTA

DECEMBER, 2020

Abstract

Traffic congestion on urban freeways has become a serious problem in major metropolitan areas, causing delays, pollution, reduced road safety and degradation of infrastructure. Predictive freeway control measures are shown to be effective in reducing traffic congestion on urban freeways. Each predictive freeway control measure includes three major components: 1) freeway capacity constraints 2) a traffic prediction model, and 3) an optimization problem formulation with respective solution.

Most of the freeway control models considered deterministic values of capacity, occupancy or density as the physical constraints. However, previous research confirmed that the observed freeway capacity follows a probabilistic behavior. In terms of the traffic prediction models, the majority of control approaches used deterministic macroscopic traffic flow models to predict the traffic parameters. These models are not suitable in capturing lane by lane and stochastic traffic behavior caused by uncertainties in driving behaviors of road users and network conditions.

Finally, the current optimization approaches mainly try to achieve system-wide benefits while overlooking the impact of local stochastic constraints and equity issues of such systems.

In this thesis, I initially investigated and modeled the probabilistic behavior of freeway capacity based on real-world traffic data. The results not only confirmed probabilistic capacity, but also indicated that different weather conditions result in the distinct parameters of the probability distribution functions.

Thereafter, I developed a traffic state prediction approach based on a stochastic microscopic three-phase model. The rigorous analysis carried out showed that the proposed method predicts traffic parameters with an accuracy comparable to that of data-driven models without the same intensive data requirements.

Finally, I developed a predictive ramp metering approach that facilitates cooperative control using a bargaining game theory approach. This configuration allows the controllers to communicate their state and decision information, and find the control solution with a compromise between local and global performance. This unique property allows local equity considerations, in regard to a fair distribution of occurrence of breakdown events, while seeking system-wide efficiency. The results showed that the proposed model outperformed the deterministic capacity-based models in terms of the effectiveness and equity of the ramp metering solutions.

Acknowledgements

First and foremost, I would like to express my sincere appreciation to my supervisor, Dr. Lina Kattan, for her insightful guidance, encouragement, trust and confidence that she showed upon me throughout my graduate studies at the University of Calgary. She is not only my supervisor but also my role model, a great mentor, an outstanding teacher, a great friend and an affectionate sister to me. Without her substance of a genius, convincing guidance, and persistent help, the goal of this project would not have been realized.

I would like to pay my special regards to my supervisory committee, Dr. Chan Wirasinghe, and Dr. Alexandre de Barros for their constant guidance and support.

I would like to express my gratitude to the members of my examination committee, Dr. Chan Wirasinghe, Dr. Alexandre de Barros, Dr. Nigel Waters, Dr. Alireza Sabouri, and Dr. Montasir Abbas for their valuable advice and helpful suggestions.

Many thanks go to my friends and colleagues at the AMA Active Traffic and Demand Management Laboratory in University of Calgary for contributing in continuously learning from each other especially Dr. Mohammad Ansari Esfeh, and Dr. Soheila Aalami. My sincere thanks go to Arash Afshar for his assistance with the programming.

I am thankful for the support I got from the Alberta Motor Association, Transportation Association of Canada (TAC), Canadian Institute of Transportation Engineers (CITE), Canadian Transportation Research Forum (CTRF), Queen Elizabeth II and John F. Morrall Graduate Scholarships for their generous support.

This work is financially supported through Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery and Discovery Accelerator Supplement (DAS), Urban Alliance Chair in Transportation Systems Optimization research funds, Mitacs Accelerate with

ISL Engineering, Calgary and the Alberta Innovate Strategic Research Grant on Integrated Urban Mobility through Emerging Transportation Technologies.

I would like to express my deepest gratitude to Shahoo for his unwavering love and support, and my dearest daughter, Narvin, who has made me stronger and more fulfilled than I could have ever imagined. My heartfelt thanks go to my beloved parents Ala and Marzieh, and my brother Khabat and my sister Neda for always loving, supporting, caring, and believing in me and encouraging me to pursue my dreams.

Finally, I am grateful to have had the privilege of attending the prestigious University of Calgary. This experience has afforded me the opportunity to work with some of the best and brightest, and the resources for me to achieve great success. Thank you for this opportunity.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Acronyms

AKF	Adaptive Kalman filter
ALINEA	Asservissement Line´aired’Entre´e Autoroutie´re
ARIMA	Auto-regressive integrated moving average
ASDA	Automatic tracking of moving traffic jams
BM	Breakdown minimization
CA	Cellular automata
CAV	Connected and automated vehicle
CDF	Cumulative distribution function
CORSIM	Corridor simulation
CTM	Cell transmission model
CV	Connected vehicle
DCRC	Digital cellular radio communication
FD	Fundamental diagram
FDL	Fusion deep learning
FFS	Free flow speed
FIFO	First-in-first-out
FOTO	Forecasting of traffic objects
GARCH	Generalized autoregressive conditional heteroskedasticity
GRU	Gated recurrent unit
HCM	Highway capacity manual
I2I	Infrastructure to infrastructure
IDM	Intelligent driver model
KK	Kerner-Kelnov
LSTM	Long short term memory
LWR	Lighthill-Whitham-Richards
MAPE	Mean absolute percentage error
MLP	Multilayer perception
MOBIL	Minimizing overall braking induced by lane change
MPC	Model predictive control
NGISM	Next generation simulation
PDF	Probability distribution function
RM	Ramp metering
RMSE	Root mean square error
RSU	Road-side unit
RTMS	Remote traffic microwave sensor
SARIMA	Seasonal auto-regressive integrated moving average
SCTM	Stochastic cell transmission model
SD	Standard deviation
SDMPC	Stochastic distributed model predictive control
SWARM	System-wide adaptive ramp metering
TTS	Total time spent
V2I	Vehicle to infrastructure
VSD	Virtual spot detector
VSL	Variable speed limit

List of Notations

λ	Shape
μ	Scale
σ	Location
v_n	Vehicle speed
v_{free}	Free flow speed
$v_{c,n}$	Desirable speed
$v_{s,n}$	Safe speed
x_n	Vehicle location
τ	Simulation interval
g_n	Space gap
D_n	Synchronization distance
d	Vehicle length
a_n	Acceleration
b_n	Deceleration
ξ_a	Impulsive random variables for acceleration
ξ_b	Impulsive random variables for deceleration
$\mathrm{p_a}$	Probabilities of random acceleration
$\mathrm{p_b}$	Probabilities of random deceleration
x_k	Traffic state vector at time t
$\hat{x}_t$	Estimated traffic state vector at time t
w_t	State equation noise
v_t	Measurement equation noise
y_t	Noisy measurement of state
K_t	Kalman gain
Q_t	Process noise covariance
R_t	Measurement noise covariance
N	Observation step counter
v_{syn}	Synchronization speed
v_{jam}	Wide moving jam speed
q_{jam}	Wide moving jam flow
ρ_{max}	Maximum density
N_c	Control horizon
N_p	Prediction horizon
$u(k)$	Control input vector at time k
W	Penalty for the rate of change in the state parameters
V	Penalty for the rate of change in the input parameters
T_r	Average link travel time upstream of the controller r
L_r	Expected queue length on the ramp r
d_r	Optimal merging flow on ramp r
D_r	Arrival inflow ramp r
α_d	Disagreement coefficient
F_{right}	Mainstream flow on the right lane
$d_{disgr-r}$	Disagreement merging flow on ramp r

CHAPTER 1: INTRODUCTION

1.1 Background

Traffic congestion is a common condition in large and growing metropolitan areas across the world. One of the typical occurrences of traffic congestion is on critical urban freeways that carry high volumes of vehicles. The resulting traffic breakdowns on freeways lead to various problems including travel delays, and higher risk of collisions. Traffic operation is further hindered by the reduction in the discharge flow rate at the downstream of bottlenecks due to the capacity drop phenomenon (Cassidy & Bertini, 1999; D. Chen & Ahn, 2018; Leclercq et al., 2016; Srivastava & Geroliminis, 2013; K. Yuan et al., 2015). Additionally, idling in long traffic queues causes fuel consumption, air pollution and has severe environmental impacts. The efficiency of freeways can be optimized by using real-time traffic control and management systems which respond to the dynamic and the random nature of traffic almost instantaneously. Considerable safety, environmental and performance enhancements result from developing efficient freeway traffic control solutions.

Several freeway control strategies have been developed in recent years for improving the operation of freeway traffic and alleviating traffic congestion. The main freeway control methods are ramp metering, variable speed limits, and variable message signage. These methods were initially proposed to be applied individually in local control plans (Alessandri et al., 1999; Messmer & Papageorgiou, 1994; Papageorgiou et al., 2003; Papageorgiou & Kotsialos, 2002; Smulders, 1990). With further developments in computation and communication technologies, coordinated and integrated freeway control plans were developed in which a combination of control methods were simultaneously applied in several segments of freeways (D. Li et al., 2014; X. Liu et al., 2013; Li Zhang et al., 2012).

Ramp metering is one of the most efficient control measures on freeways in which the number of merging vehicles to the freeway mainstream is calculated and implemented to reduce congestion. Previous studies have shown that, applying optimal ramp metering strategies can decrease the total time spent (TTS) on the network by up to 50 percent (Papageorgiou & Kotsialos, 2002). Various ramp metering methods have been developed in the literature that can be categorized into three main groups based on the level of responsiveness to the real-time traffic dynamics including: 1) Off-line or pre-timed control 2) Reactive or responsive control, and 3) Predictive or proactive control. A brief explanation of each group is presented bellow.

Off-line or fixed-time ramp metering is mainly developed based on historical traffic data for a particular time of day. In fixed-time ramp metering a linear or quadratic programming formulation is developed and solved, off-line, to derive the optimal ramp metering rates. In this method, real-time traffic information is not utilized which results in considerable errors in the computation and consequently underutilization or overloading of the freeway (Ghods et al., 2010; Papageorgiou & Kotsialos, 2002).

Rather than the historical traffic data, real-time traffic measurements are used to develop reactive control strategies to respond to the real traffic conditions on freeways. Reactive strategies determine control variables based on maintaining traffic conditions closed to predefined levels. Various reactive ramp metering strategies are developed and applied around the world in local or coordinated control applications. ALINEA (Asservissement Line´aired'Entre´e Autoroutie´re) and demand-capacity methods are well-known local ramp metering strategies (Hadj-Salem et al., 1990; Masher, 1975). More advanced reactive ramp metering plans such as helper algorithm and system-wide adaptive ramp metering (SWARM) are mainly formulated based on optimizing an objective criterion such as total time spent considering the capacity constraints on the freeway and on-ramps (Lipp et al., 1991; Paesani, 1997). The solution to such optimization problem is a dynamic vector of optimized ramp metering rates for a relatively long segment of the freeway. The reactive freeway control strategies have been shown to be more efficient compared to the off-line methods (Papageorgiou & Kotsialos, 2002). However, the main drawback of such control measures is the relatively long computation time due to the large number of variables specially in the coordinated control applications. In other words, by the time the control plan is formulated and deployed, the traffic conditions may have changed. Predictive control schemes that are explained next, were developed to overcome the limitation of their reactive control counterpart.

Predictive control strategies are developed based on responding to the predicted traffic conditions over a short future horizon (e.g. 5 min), and they have been shown to be more effective compared to the reactive schemes by eliminating the time lag between real-time observations and control actions (Ghods et al., 2010). Model predictive control (MPC) is a widely adopted predictive traffic control approach that finds optimal control measures over a rolling horizon. An MPC-based predictive freeway control model includes a traffic prediction model, responsible for multiple-step ahead traffic state predictions, and an optimization problem formulation that finds the optimal control solutions subject to the operational and physical constraints of the network. The performance of MPC-based freeway control strategies is highly affected by the adaptability and accuracy of the traffic prediction model and the system constraints model. However, there are still several gaps in the literature regarding these main components and procedures of MPC-based traffic control models.

1.2 Motivations for a predictive stochastic capacity-based ramp metering model

So far, the majority of MPC-based control approaches use deterministic macroscopic traffic flow models (e.g. METANET, CTM) to predict traffic parameters and the value of the objective function (Bellemans et al., 2003; Ferrara et al., 2012; Ghods et al., 2010; A. Hegyi, 2004; Hegyi et al., 2005; Karimi et al., 2004; M. H. Ma et al., 2015; Papageorgiou, 1995; Papamichail et al., 2010; Zegeye et al., 2009). However, the indeterminate impacts of the stochastic driver behaviors may violate the theoretical assumptions of such models and cause suboptimal operations on the freeway segments (Papamichail et al., 2010; G. Zhang & Wang, 2013).

In addition, decades of research on breakdown phenomena in freeway bottlenecks confirmed that the observed freeway capacity follows certain probabilistic behaviors (Brilon et al., 2005; Brilon & Geistefeldt, 2009; Y. Y. Chen et al., 2016; Elefteriadou et al., 2011; Elefteriadou & Lertworawanich, 2003; Geistefeldt, 2010; Han & Ahn, 2018; Ozguven & Ozbay, 2008; Persaud et al., 2001; K. Yuan et al., 2015). In other words, at a given flow rate under free flow conditions, traffic breakdowns may occur, but they do not necessarily occur. However, the majority of the previously developed freeway control models did not take into account the stochastic behavior of freeway capacity, and the models mainly considered deterministic values of capacity, occupancy, or density (Hegyi et al., 2005; Lu et al., 2011; M. H. Ma et al., 2015; Papageorgiou, 1995).

Under saturated traffic conditions, drivers must adjust their speed to the preceding vehicles while keeping a space gap. To maintain this gap, drivers need to make decisions regarding acceleration, deceleration, and lane changing maneuvers. These driving behaviors vary greatly from one driver to another depending on their aggressivity, reaction time, and vehicular characteristics. When traffic volume is high, microscopic driving decisions, such as sudden deceleration or lane changing, can lead to uneven headway distributions, which create shockwaves and possibly traffic breakdown. Stochastic behavior can be macroscopically represented by either the well-known scatter-plot in the congested branch of the fundamental diagram of traffic flow as demonstrated in empirical studies or, the synchronized flow area in the three-phase theory (Kerner, 2002; X. Wu et al., 2010). Such behavior cannot be fully modeled using a macroscopic traffic flow model, the model of choice in most freeway control strategies. Macroscopic flow models utilize aggregate point detector data (e.g., average speed, flow, and density), which often obscure many features of interest such as any abrupt changes in the traffic that may be the root cause of traffic breakdowns.

With the advancement of probe and connected and automated vehicles (CAVs) technologies, in addition to stationary data, copious precise and accurate microscopic data is expected to be gathered and disseminated in real time. Microscopic stochastic traffic prediction models are yet to be augmented with such microscopic data that can be harnessed to closely reflect the stochasticity of a driver's behaviors. The advantage of developing such microscopic prediction models is that they are capable of estimating traffic flow parameters on a lane by lane basis.

Analysis of empirical traffic data shows that speed breakdowns on freeway bottlenecks do not occur at the same time for all lanes (D. Ma et al., 2013). For instance, next to a lane drop section, congestion often starts in the right lane due to the lane changing activity of merging vehicles. Predicting the onset of breakdown in the right lane of a merging bottleneck can be used to trigger proactive controls, which can avoid breakdown occurrence and its propagation to other lanes. While still in its infancy, per lane traffic state prediction is critical to the development of proactive and adaptive control strategies (Nagalur Subraveti et al., 2019). An example of an application of this technology is the recently growing area of microscopic traffic control schemes such as trajectory optimization for connected and automated vehicles CAVs (Hu & Sun, 2019).

In addition, most of the MPC-based traffic control measures find the optimized control solutions in a non-cooperative manner. In other words, the controllers do not communicate and utilize the information regarding the traffic state and control decisions of other controllers. However, such communications can be exploited to improve the decision making of each controller. Future development of CAVs and infrastructure to infrastructure (I2I) and vehicle to infrastructure (V2I) communications can play a substantial role in enabling real-time controllers to work cooperatively to achieve overall objectives of improved freeway performance (Shaaban et al., 2016).

1.3 Objective and scope

The main objective of this research is to enhance the existing predictive ramp metering methods by developing a dynamic predictive and cooperative ramp metering strategy that considers the probabilistic behavior of freeway capacity. For this purpose, three sub-problems are required to be examined: 1) probabilistic freeway capacity modeling, 2) stochastic microscopic-based short-term traffic state prediction model, and 3) predictive and cooperative ramp metering under probabilistic capacity. Each subproblem targets a critical component of the general predictive freeway control structure, shown in Figure 1.1. The following is a brief overview of each sub-problem.

1.3.1 Probabilistic freeway capacity modeling

In recent decades, it has been shown that freeway capacity varies depending on the segment of a freeway, such as weave, merge, or diverge areas, and on the time of day. Data analysis showed that breakdowns do not necessarily occur at the same traffic demand; this finding challenged the traditional deterministic capacity definition (Elefteriadou & Lertworawanich, 2003; Persaud et al., 2001). In general, stochastic breakdowns and capacity behaviors are investigated through model-

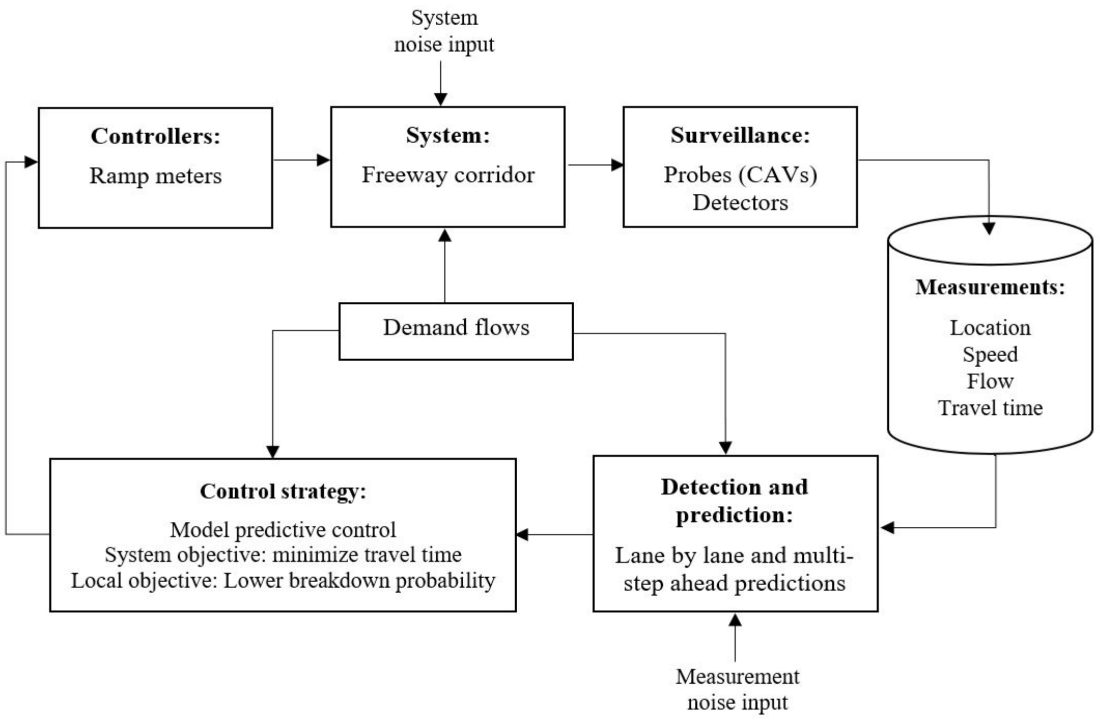


Figure 1.1. A schematic structure of predictive freeway control

based analysis and data-based analysis approaches. In the model-based analysis, a probabilistic model is selected, and data is used to calibrate model parameters. In contrast, in the data-based analysis, no predefined probabilistic model is determined, and the best model is selected according to compatibility with the dataset. These studies considered pre-breakdown flows, breakdown flows, and discharge flows to investigate stochastic behaviors. Some particular models include Markov chains, Product Limit Method, Bayesian estimators (Brilon et al., 2005; Brilon & Geistefeldt, 2009; Evans et al., 2001; Modi et al., 2014; Ozguven & Ozbay, 2008).

Regardless of the various analytical approaches, all studies confirmed the stochastic behavior of freeway capacity, especially at merging bottlenecks. However, most traffic control schemes are designed based on a constant (deterministic) bottleneck capacity, which leads to a latent possibility of control failure or capacity underutilization depending on the aggressiveness or conservativeness of the control measure.

In order to achieve a better understanding of breakdown behavior on freeway merging bottlenecks, in this research, real observed traffic data on Highway 2 in Calgary, Alberta is used to derive and compare both deterministic and probabilistic behaviors of pre-breakdown flows.

1.3.2 Stochastic short-term traffic state prediction model

Connected and automated vehicle initiatives, moving close to deployment, will provide a wealth of individual vehicle trajectories data that in conjunction with fixed sensor data can be used as input to next generation of advanced traffic control devices. Data fusion methods that combine traffic information from such mobile sensors (e.g. CAVs) and stationary sensors have been shown to improve the accuracy and efficiency of traffic prediction results (Di et al., 2010; Guo et al., 2014a; Nantes et al., 2016; Ruppe et al., 2012; van Erp et al., 2018; Vlahogianni et al., 2014). A few analytic models such as the LWR, CTM, METANET, and RENAISSANCE macroscopic stochastic models are used to capture some aspects of probe vehicle information and fuse them with detector data to produce more accurate traffic state predictions (Allström, 2016; Bekiaris-Liberis et al., 2016a; Duret & Yuan, 2017; Kawasaki et al., 2019; Mazaré et al., 2012; Papadopoulou et al., 2018; Van Hinsbergen et al., 2010; Y. Wang et al., 2008; Work et al., 2008; Y. Yuan et al., 2012).

However, it is important to examine if additional insights can be gained with the use of microscopic models that can reflect the unique stochastic driver behavioral aspects that are captured through the collected mobile data. Such insights would be helpful to identify the location and magnitude of any microscopic disturbance that are the main trigger of traffic breakdown in heavy traffic. Examples include shock wave formation that are usually created at the individual vehicle level (e.g. vehicle changing lanes or coming to a sudden stop). A few studies have explored the use of microscopic traffic flow models to estimate and predict traffic conditions over a short prediction horizon; however, the stochastic car following parameters are not adequately considered. (Chrobok et al., 2002; Miska, 2007; Schreckenberg & Wahle, 2001).

In addition, studies showed that fundamental diagram parameters vary significantly among lanes, and aggregated parameters may result in a non-equilibrium fundamental diagram attributed to mixing different traffic states. Thus, lane-specific behavior analysis is critical to predict the traffic parameters, the occurrence of breakdown at bottlenecks, and the propagation of congested patterns over lanes (Duret & Audin, 2009; Pan et al., 2019; Shiomi et al., 2015).

1.3.3 Predictive and cooperative ramp metering under probabilistic capacity

A series of existing studies on stochastic-capacity-based control approaches showed significant improvements in freeway operations in terms of increased throughput of freeways, delayed breakdowns, and reduced average travel time and congestion duration (Dong et al., 2018; Han & Ahn, 2018; Jin et al., 2018; Pan et al., 2019; Schmitt & Lygeros, 2020; H. Wang et al., 2010; Zhong et al., 2014). In the analysis conducted in the literature, several challenges are revealed that still need to be addressed in developing predictive ramp metering strategies. While there is limited research on modeling stochastic capacity in merging bottlenecks from a lane by lane perspective, there are even fewer studies that incorporate these lane level stochastic behaviors into freeway control models.

In addition, only a few works in the literature utilized microscopic models in the predictive traffic control methods. For instance, Zegeye et al. (2009) used the Gazis-Herman-Rothery (GHR) stimuli-response car-following model in an MPC-based speed limit control approach, along with an average-speed-based emission model to simultaneously optimize total time spent (TTS) and emissions. However, the car following model in this study considered only the longitudinal kinematic behavior of vehicles and drivers. To our knowledge, no study incorporated the stochastic microscopic traffic models to the MPC-based ramp metering strategies.

In most MPC-based ramp metering models the system-wide efficiency is achieved at the expense of some controllers losing performance compared to others. This issue raised concerns regarding the equity of such systems. A cooperative ramp metering solution that is able to provide a balance trade-off among local and global performance of the control strategy while considering the stochastic local breakdown probabilities, is yet to be explored.

1.4 Proposed methodology and research contributions

In this thesis, a dynamic predictive and cooperative ramp metering strategy is proposed that considers the probabilistic behavior of freeway capacity. The developed model relies on a stochastic microscopic traffic state and travel time prediction model that fuses data from probe and future connected and automated vehicles (CAVs) with stationary detector data to obtain short-term traffic predictions based on an adaptive Kalman filter (AKF) on a lane by lane basis. The multi-step ahead traffic state prediction outputs are used to identify occurrences of stochastic breakdowns and to trigger and operate the cooperative ramp metering control.

The problem of cooperative ramp control is formulated to minimize system-wide travel time while considering the stochasticity of local bottleneck capacity. The proposed approach considers the need for flexibility in the solution to the ramp metering problem by integrating the system-wide and local benefits through a bargaining framework

This thesis documents several key contributions made to the field of freeway traffic prediction and control. The contributions of this research can be summarized in three categories that are outlined below.

Contributions to the probabilistic freeway capacity modeling: The following contributions are made to the freeway capacity modeling:

- Modeling deterministic freeway capacity based on real observed data and using a new regression analysis method to calibrate the fundamental diagram and evaluating the impacts of various weather conditions on FD parameters.
- Modeling probabilistic freeway capacity and evaluating the impacts of various weather conditions on the parameters of the probability distribution models.

Contributions to the short-term traffic state and spatial-temporal traffic pattern prediction: The model developed for this part of the research offers several contributions. The contributions are listed as follows:

- Developing a short-term microscopic-based traffic prediction model that incorporates the stochasticity of driver behaviors such as lane changing, deceleration, and acceleration decisions to the lane-based traffic state prediction problem. This analytical based prediction model has the explanatory power to reconstruct and predict congestion phenomena resulting from both recurrent and non-recurrent traffic conditions without the need of intensive historical data.
- Predicting the lane-by-lane spatiotemporal congestion patterns to provide timely information on anticipated breakdown occurrence.

- Addressing the inconsistency of measurement problem generated by the multi-type sensors (i.e. detector and CAV data) by the utilization of the adaptive Kalman filtering (AKF) methods. The self-adaptability properties of AKF makes the traffic states prediction problem on a lane level basis computationally efficient for multiple short time steps ahead.

Contributions to the predictive ramp metering under probabilistic capacity: In this work, I attempt to address the existing gap in the literature through the following contributions:

- Developing a predictive and cooperative ramp metering model based on a distributed model predictive control approach. The bargaining properties of the proposed model give the controllers the choice to cooperate depending on the benefits achieved from the cooperative decision and prioritizes avoidance of localized traffic breakdown.
- Incorporating the stochasticity in traffic demand and freeway capacity in both the traffic prediction and control models.
- Modeling the probability of breakdowns in individual lanes, that is embodied in the novel ramp metering framework.

1.5 Thesis organization

This thesis consists of five chapters that are laid out as follows:

Chapter 2 is devoted to the modeling and calibration of deterministic and probabilistic freeway capacity at merging bottlenecks. This chapter begins with a comprehensive overview of previous studies on breakdown analysis and stochastic capacity modeling. The study site and data collection and processing are explained, followed by the deterministic and probabilistic modeling of freeway capacity under various weather conditions. The analysis in this chapter is based on real observed field data with the main focus of calibraring realistic modelling of stochastic capacity and breakdown capacity. Thus, unlike the next two chapters, the analysis conducted in this chapter does not consider CAVs.

Chapter 3 presents a short-term traffic state and spatial-temporal pattern prediction model, developed to produce multi-step ahead predictions of traffic flow, speed, and travel time on a lane by lane basis. This chapter also covers spatial-temporal congested pattern predictions and a comprehensive sensitivity and comparative analysis of the results.

Chapter 4 includes the formulation of a predictive and cooperative ramp metering based on a distributed model predictive control. The bargaining game solution algorithm and the incorporation of probabilistic capacity to the control process are also explained. Several control scenarios are implemented to evaluate the efficiency, effectiveness, and equity of the models.

Chapter 5 summarizes the findings of this research and concludes the work described in this dissertation. The contributions of this research to the greater body of literature are described and recommendations for future research are made.

CHAPTER 2: DETERMINISTIC AND STOCHASTIC FREEWAY CAPACITY ANALYSIS AND THE IMPACTS OF WEATHER CONDITIONS [1]

2.1 Introduction and background

The capacity of road transportation facilities is traditionally considered as a constant value representing the maximum vehicular traffic that the facility can carry. According to the Highway Capacity Manual, *capacity* is defined as "the maximum hourly rate at which persons or vehicles can be reasonably expected to traverse a point or a uniform section of a lane or roadway during a given time-period under prevailing roadway traffic and control condition" (HCM 2010). The term "reasonably expected" is somewhat arbitrary, which allows capacity to stochastically change according to the traffic condition at a given location.

In the majority of studies, breakdown occurrence and capacity drop are considered as interrelated concepts. When the demand exceeds capacity, the traffic state transits from an un-congested state to a congested state which is known as the breakdown state (Elefteriadou & Lertworawanich, 2003). The congested traffic state after breakdown is an undesirable phenomenon that results in increased travel time and decreased freeway efficiency. Determining the empirical freeway capacity which represents the real-world traffic conditions is critical for effective traffic management and control strategies to effectively avoid breakdown occurrence and reduce congestion on freeways.

Several studies were conducted to develop macroscopic traffic flow models to derive the freeway capacity and represent its stochastic nature. In recent traffic management strategies, the

[1] The contents of this chapter have been used in the paper entitled: "Deterministic and stochastic freeway capacity analysis based on weather conditions". Published in "Journal of Transportation Engineering, Part A: Systems", 145(5), May 2019.With permission from ASCE (Appendix II).

demand fluctuation is monitored closely using real-time traffic data. Several reactive and proactive control methods are developed to adapt with the fluctuation of the demand. However, most of these studies still assume that the capacity has fixed value and thus the variability of the supply side of the control problem has still not been given enough attention. Only a few studies challenged the definition of capacity as a stationary value and a few models were proposed to model the stochasticity of freeway capacity and, in devising freeway control strategy based on this concept of freeway capacity (Elefteriadou et al., 2011a).

To our knowledge, the factors that contribute to the probabilistic nature of capacity were not fully explored yet. Factors such as weather and road conditions were shown to have a major impact on fixed capacity and on the shape of the fundamental diagram (FD) (Lam et al., 2008; B. L. Smith et al., 2003; K. Yuan et al., 2015). However, the impact of the weather conditions on the shape of the distribution of the stochastic capacity and on the jam density is still not examined. Modeling and incorporating weather impacts are important steps in understanding and modeling the stochastic capacity. Such analysis is a crucial step for providing a more realistic representation of capacity of freeways that can be used in devising traffic control and management strategies.

In this research, real observed traffic data on Highway 2 in Calgary, Alberta is used to derive the stochastic behavior of pre-breakdown flows. In this study, pre-breakdown flow is defined as the flow rate in the time interval prior to the occurrence of breakdown as a representative of a traffic flow at which the average speed is above the minimum acceptable desired speed. A new regression analysis method is used to calibrate the fundamental diagram and to investigate the effects of various weather conditions on the freeway capacity and other FD parameters. For this purpose, the macroscopic behavior of the traffic stream at the study site is analyzed and the Newell triangular model is calibrated using an algorithm developed by Muggeo (2003) to calibrate a regression model for the data sets with unknown break-points. Weather condition as one of the most important causes of stochastic behavior of capacity is also investigated to determine their impacts on FD parameters, especially on the freeway capacity. Thereafter, breakdown phenomenon and stochastic capacity are analyzed as probabilistic concepts. Probability distribution of pre-breakdown flows are derived from the data set and compared under various weather conditions. Finally, the results from the probabilistic approach are compared to the deterministic value of capacity derived from the fundamental diagram.

Traffic flow theories and fundamental diagrams have been developed in recent decades to describe the mathematical relationship among fundamental characteristics of traffic streams including flow, speed, and density, and to determine capacity on freeways. These parameters are essential elements of all analysis tools in design, operation and control of urban streets and highways (Newell, 1993; Dhingra & Gull 2008).

The capacity of freeway as a critical parameter in freeway design and control was assumed as a constant value for several years. However, it has been shown that freeway capacity varies in different segments of a freeway such as weaving, merge or diverge areas and for different times of day. However, the variance of capacity and other traffic parameters has not been thoroughly explored (Rakha & Zhang, 2006; Yao et al., 2009; Yeon et al., 2009). Moreover, some recent studies investigated the concept of capacity as a stochastic parameter (Polus & Pollatschek, 2002).

Dervisoglu et al. (2009) calibrated the fundamental diagrams for a freeway network using data filtering and an approximate quantile regression model. The authors estimated the capacity of freeway using the maximum observed flow and showed that capacity does not necessarily appear during breakdown (Dervisoglu et al., 2009).

Data analysis showed that breakdown does not necessarily occur at the same traffic demand; this finding challenged the traditional definition of capacity as a vague conceptual definition (Kittelson & Roess, 2000). Elefteriadou & Lertworawanich (2003) investigated various definitions of freeway capacity and tried to develop a more accurate definition and an estimation method for freeway capacity. For this purpose, the authors examined traffic operations at two freeway bottlenecks, considering pre-breakdown flow which is the flow rate at the time interval prior breakdown occurrence, breakdown flow which is the flow rate at the breakdown interval, and discharge flow as the flow rate in the interval after breakdown at which the breakdown is dissolved. The most important finding was that the maximum flows were from pre-breakdown flows set for one of the sites, and from the discharge flows for the other site; however, breakdown flows were less than other groups for both study sites. Discharge flow rate was also shown to vary according to the congestion state and the average speed at the upstream of the bottleneck (Yuan et al., 2015, 2017).

Kerner, (2002) defined traffic breakdown as a transition from free flow to synchronized flow (F to S transition) and freeway capacity in three-phase theory framework is a range, including an infinite number of traffic flows between a minimum threshold flow for breakdown and the maximum freeway capacity. The nucleation nature of traffic breakdown was also analyzed through empirical and spatiotemporal traffic information and it was proved to be triggered by speed disturbances and stochastic driver's responses to these fluctuations (Kerner, 2017).

In general, developed stochastic breakdown and capacity studies can be categorized in two major groups which are model-based analysis and data-based analysis. In the model-based analysis, a probabilistic model is selected, and data is used to calibrate the model parameters. On the other hand, in the data-based analysis, no predefined probabilistic model is determined, and the best model is selected according to the compatibility with the data set. Elefteriadou (1994) developed a probabilistic model which determined the probability of breakdown based on traffic flow on the freeway and the merging vehicles cluster size on the on-ramp (Elefteriadou 1994). This model was further developed using Markov chains to determine the probability distribution of breakdown before a given time (Evans et al., 2001).

In another model-based study, traffic breakdown was considered as a failure event and a probability distribution function was developed using the analogy with statistics of life analysis based on the product limit method. Traffic dynamics, traffic reliability, and traffic efficiency and capacity drop phenomenon were also investigated in this study (Brilon et al., 2005; Brilon & Geistefeldt, 2009). This method was recently used to estimate reasonable values of capacity for Florida highways (Modi et al., 2014). In another model-based approach, the three different estimators of Baysian, Kaplan-Meier, and Nelson-Aalen were calibrated and compared in terms of modeling probabilistic breakdown. It was found that the non-parametric Baysian estimator model represents a more complete probability curve compared to the other two estimators (Ozguven & Ozbay, 2008).

The majority of stochastic breakdown analysis in the literature fit in the data-based analysis group. Statistical exploration of breakdown phenomenon was initiated by a numerical investigation of pre-queue and queue-discharge flows probabilities (Persaud et al. 1998). This study was further developed to calibrate a logistic model to calculate the probability of breakdown and to evaluate the effect of fixed-rate and variable-rate ramp metering on those probabilities (Persaud et al., 2001). In another data analysis approach, stochastic momentary capacity was determined based on the intersection of best fit regression lines for dense and unstable flow regimes in observed speed-flow data (Polus & Pollatschek, 2002).

Geistefeldt (2010) investigated the consistency of model-based and data-based breakdown probability models. The results indicated that while the data-based method overestimates the breakdown probability at low flow rates it underestimates it at highest volumes. On the other hand, the product limit method as a model-based approach showed more consistency especially for the highest volumes; however; it requires a considerable data to achieve acceptable results (Geistefeldt, 2010). Laflamme (2013) determined the stochastic capacity independent of the breakdown information by an extreme value distribution function to the daily maxima data set (Laflamme, 2013). In a recent study, a combined stochastic capacity and stochastic differential equation model was developed to describe the traffic conditions on freeways and showed that breakdown phenomenon and the congestion recovery are stochastic processes (Ossenbruggen, 2016).

There are few studies in the literature that investigated the impact of weather conditions on traffic flow parameters. However, none of them explored the effects of different weather conditions on the stochastic behavior of capacity and its probabilistic distribution models. Smith et al. (2003) explored the influence of intensity of rainfall on the capacity and operating speed of freeways and found that heavy rain decreases the capacity up to 30% (B. L. Smith et al., 2003). The influence of rainfall with various intensity levels and different probabilities of occurrence formed a set of scenarios to include the demand and supply uncertainties in a traffic assignment problem using weather forecasted data (Lam et al., 2008). These studies mainly reported ranges of reduction percentage of capacity for various rainfall intensities. However, these ranges do not provide sufficient information about the stochastic effects of weather conditions in estimating the probability of traffic breakdown; such consideration is crucial for online traffic flow management and control.

2.2 Study site description and data collection

The study area for this research is a northern segment of Highway 2 (Deerfoot Trail) in Calgary, Alberta, located between McKnight Boulevard and 32nd Avenue (Figure 2.1). Deerfoot Trail is a freeway section of the Queen Elizabeth II highway and is the major north-south transportation route through the city of Calgary. The majority of Deerfoot Trail is 6 lanes in total, but there are several 4, 8, and 10 lane sections. The segment that is studied in this research is in southbound direction and includes 4 lanes with a posted speed limit of 100km/hr.

Remote traffic microwave sensors (RTMS) were used to collect data for this study. RTMS measures the distance to objects in the path of its microwave beam. This ranging capability allows it to detect moving and stationary vehicles in multiple direction zones and a single RTMS can monitor traffic in up to twelve lanes. The internal processor calculates volumes, occupancy, headway, average speed, and vehicle classification for each lane and transmits the information using its communication interface (RTMS G4, User Guide). For this study, the RTMS located at 300 meters upstream of the intersection of Deerfoot Trail and 32nd avenue was used to collect traffic information. Traffic data was collected from August 2015 to February 2016 for the total duration of 150 days at 30-second time intervals. The aggregated data for 15-min intervals was calculated for the analysis carried out in this study which formed a sample size of 8470 and 130 data points for free flow and congested flow respectively. Traffic information from other sensors located between the observed on-ramp and the downstream bottleneck was also investigated to detect propagated congestion from downstream bottleneck.

Weather condition reports provided by the Government of Canada for the Calgary International Airport station were used to categorize the collected data set into four weather conditions including: snowy, rainy, low visibility and clear. The government of Canada provides access to the historical weather, climate data, and related information for numerous locations across Canada. The reports are available as the hourly, daily and monthly summaries. In this study, the hourly information of the qualitative description of weather conditions was used (Historical weather and climate data 2016). However, some of the descriptions were combined to avoid numerous weather categories. Table 2.1 shows the weather conditions classification based on different reported subcategories and the number of available data points for each weather condition.

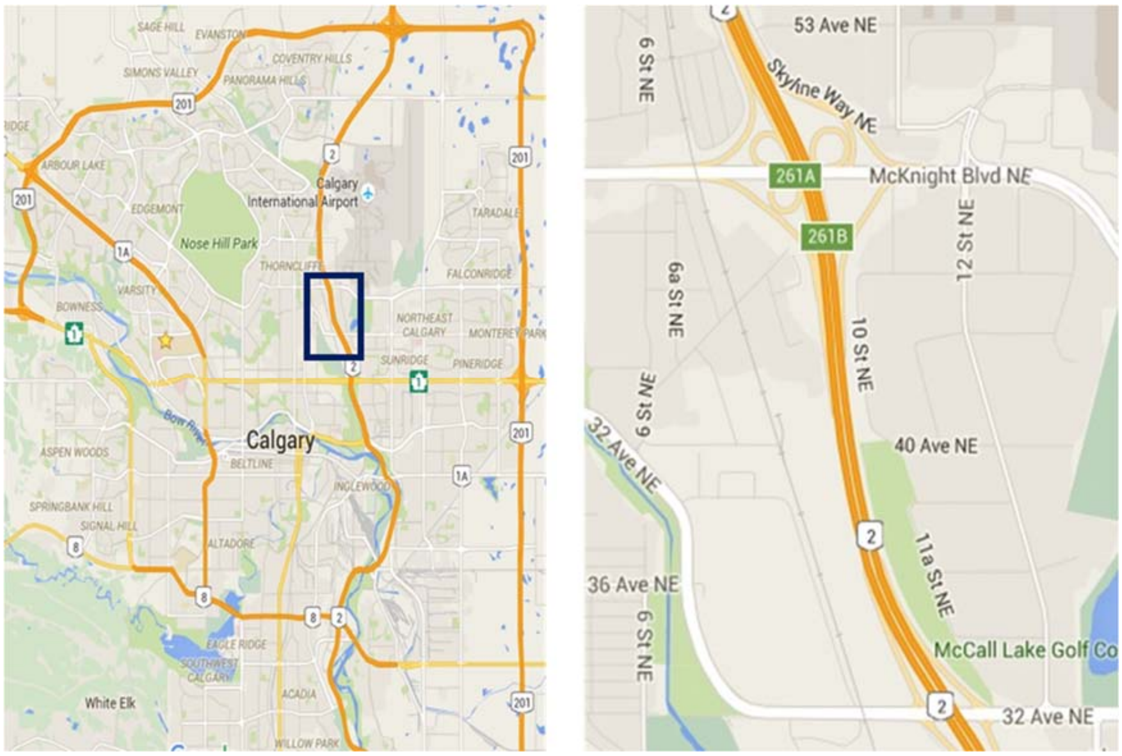


Figure 2.1. Study site for breakdown analysis

Table 2.1. Various weather conditions and subcategories

Weather conditions	Clear	Low Visibility	Rainy	Snowy
Weather subcategories	Clear Mainly clear Cloudy Mostly cloudy	Fog Freezing fog Freezing drizzle Ice crystals Smoke Blowing dust Haze	Rain Rain shower Moderate rain Thunderstorm	Snow Blowing snow Snow shower Snow grains
Number of data points	4854	1992	550	1204

2.3 Calibration of the Newell triangular model

The collected data is used to investigate the interaction between macroscopic specifications of traffic behavior on Deerfoot Trail, Calgary, Canada. The maximum observed flow as a constant value for capacity is determined through the fundamental diagram. The collected data set is categorized based on four weather conditions including snowy, rainy, low visibility, and clear. The Newell triangular model is calibrated for the whole data set and for the categories separately.

2.3.1 FD calibration for the full data set

FD calibration is conducted based on the recently developed statistical approach of Muggeo algorithm. This method was initially developed for biomedical applications in which effect of some risk factors may change before and after some threshold value (Muggeo 2003). In this segmented regression algorithm, an iterative calculation process is conducted to calculate the break-point, which is the point of slope change in the triangular regression. Thus, first, a fixed break-point is assumed based on the condition that the first order Taylor's expansion holds around the break-point. Then, the regression line slopes are calculated, and the break-point is improved based on the new fit. This process continues until convergence (Muggeo 2003). The details of the Muggeo's regression algorithm are presented in Appendix I.

The advantage of using Muggeo algorithm compared to previously developed fundamental diagram regression models is that in this method the calibration is completely based on the statistical analysis of the whole data set and avoids imposing statistically biased thresholds to the analysis. Muggeo algorithm is used to adopt a triangular model for the data set directly and the break-point of the regression is found at the density of 15.8 veh/km through this process. The jam density which is the x-intercept of the regression line for the congested part, is 154.4 veh/km/l. However, it can be observed from the scatterplot that most of the observation in the sample space are from the free flow. The subsequent step of analysis is conducted to avoid the errors caused by the unbalanced data set.

In the next step, the data set is categorized into two groups, with the densities lower than the break-point, corresponding to the free flow state, and the densities above the break-point, corresponding to the congested state. The maximum observed capacity and the corresponding density (i.e. critical density) are obtained from the intersection of the extended regression line of the free flow part and the horizontal line passing the maximum observed flow. This deterministic maximum value was considered as an estimation of capacity to model the ideal performance of the freeway in the calibrated macroscopic model. Various parameters such as weather conditions, incidents, and drivers' behavior may decrease the capacity to a lower value. Thus, the choice of maximum observed capacity provides a basis to evaluate the impact of various weather conditions on the ideal freeway operation (Dervisoglu et al., 2009). The regression analysis results for both steps are illustrated in Table 2.2.

Table 2.2. Summary of the triangular regression using Muggeo algorithm

Steps	Parameters	Estimated	P-value
Step1	Free flow slope	110.1	<2e-16
	Congested state slope	-12.6	<2e-16
	Break-point	15.8	NA
Step 2	Free flow slope	110.1	<2e-16
	Maximum flow on congested line	2092.9	<2e-16
	Congested state slope	-16.4	<2e-16
	Notes: Residual standard error (free flow): 74.07 on 54340 degrees of freedom Multiple R-squared (free flow): 0.98 F-statistic (free flow): 2.419e+06 on 1 and 54340 DF, p-value: < 2.2e-16 Residual standard error (congested flow): 326.6 on 1004 degrees of freedom Multiple R-squared (congested flow): 0.2286, Adjusted R-squared: 0.2278 F-statistic (congested flow): 297.5 on 1 and 1004 DF, p-value: < 2.2e-16		

The summary of the results in Table 2.3 shows that the calibrated line for the free flow state has a high R-squared value of 0.98 and a low p-value, which shows the strong linear relationship between flow and density with the free flow speed of 110.1 km/hr and the critical density of 20.7 veh/km/l. The maximum flow based on the discussed method is found to be equal to 2288 veh/hr/l. However, the R-squared value of 0.23 for the congested flow shows a weak linear regression. This result is compatible with the empirical studies conducted by Kerner who developed accordingly the three-phase theory of traffic flow in which the congested flow covers an area on the flow-density plane. The regression line represents the backward shockwave speed in the flow-density or the J line in the context of three-phase theory; however, it may not be an accurate representative of the entire congested flow behavior (Kerner, 2009). The p-value and the

F-statistics are small, which show the statistical significance of a jam density value of 127.4 veh/km/l.

Table 2.3. FD Parameters for proposed regression method and simple regression

Parameters	Proposed approach	Simple regression
free flow speed	110.1	110.7
jam density	127.4	126.8
critical density	20.7	20.5
Flow reduction	465.8	461.2
Std. Error	Free flow part: 74.0 Congested flow part: 326.6	Free flow part: 46.8 Congested flow part: 322.3
R-squared	Free flow part: 0.98 Congested flow part: 0.23	Free flow part: 0.99 Congested flow part: 0.23
F-statistics	p-value: < 2.2e-16	p-value: < 2.2e-16

Figure 2.2 illustrates the data points and regression lines for the full data set in color red; a clear flow drop is observed when the density reaches the critical density. The flow reduction at the break point in this analysis is equal to 465.8 veh/hr/l, which is about a 20 percent decrease in flow.

The results of the proposed fundamental diagram calibration approach are compared to those obtained based on a previous study (Dervisoglu et al., 2009) that used the simple linear regression to find the Newell triangular model by imposing the minimum free flow speed threshold of 95 km/hr. Thus, the data points that correspond to speed values below 95 km/ hr were deliberately ignored. However, since there is no clear evidence that these records were wrongly reported, from a statistical analysis perspective, it is not recommended to disregard such observations from the data set. To address this shortcoming, the segmented regression model adopted in this research is used to filter the data and avoids the problem of manually manipulating the data. The comparison of the results, shown in Table 2.3, indicates that the fundamental parameters of the triangular model including free flow speed, critical density, and jam density are similar in both methods.

2.3.2 FD calibration for various weather condition categories

In this section, the collected data points are categorized based on hourly weather conditions reports for the selected time-period (Historical weather and climate data, 2016). Four categories of weather condition are included: clear, rainy, snowy and low visibility. Similar to the previous section, Muggeo calibration approach is applied separately to the four categories to evaluate the influence of weather condition on FD parameters. The results of the analysis are illustrated in Table 2.4 and the plot of regression lines in Figure 2.2.

Table 2.4. FD Parameters for various weather conditions

Parameters	Clear	Low Visibility	Rainy	Snowy	All conditions
free flow speed (km/hr)	110.7	109.8	106.2	98.5	110.1
Capacity (veh/hr/l)	2288	2208	1992	1964	2288
critical density (veh/km/l)	20.6	20.1	18.6	19.8	20.7
Congested state slope	-13.9	-14.6	-20.4	-20.8	-16.4
jam density (veh/km/l)	155.6	143.7	102.4	89.1	127.4
Average Spacing (m/veh)	6.4	7.0	9.8	11.0	7.8
Flow reduction(veh/hr)	368.0	357.1	237.1	430.6	465.8
R-squared for Free flow	0.98	0.98	0.97	0.91	0.98
R-squared for congested flow	0.26	0.47	0.33	0.16	0.23
F-statistics (p-value)	<2e-16	<2e-16	<2e-16	<2e-7	<2e-16

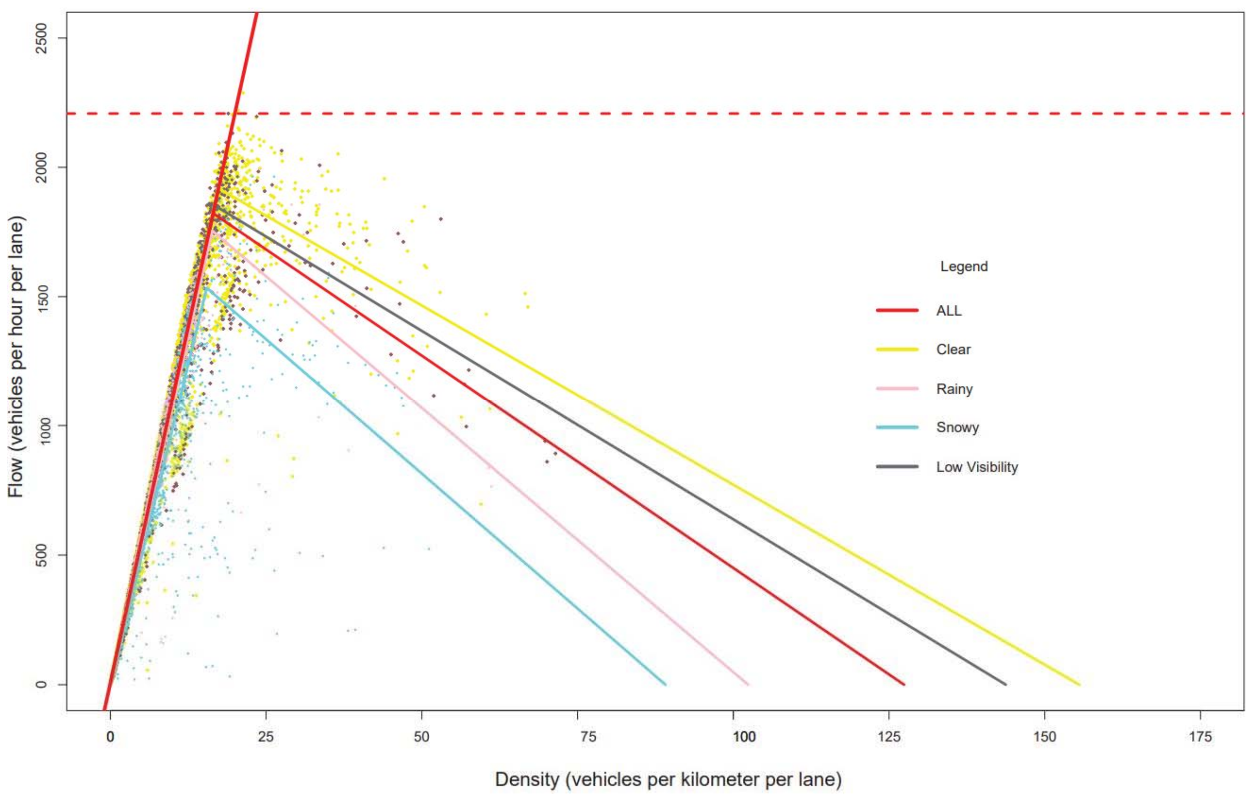


Figure 2.2. Calibrated FD for various weather conditions and all data

Comparing the results shows that the influence of weather conditions on the FD parameters is significant. This is especially the case for rainy and snowy conditions in which the maximum observed flow decreases from 2288 veh/hr/l for clear weather to 1992 and 1964 veh/hr/l, respectively. The effect on jam density is also significant. The results of the calibrated model show that jam density decreases from 155 veh/km/l in clear condition to 102 and 89 veh/km/l for rainy and snowy conditions, respectively. This is an important finding that might explain the impact of weather on drivers' lane following behavior. In addition, this finding that different jam densities exist under different weather condition is expected to have a significant contribution to devise more robust freeway real-time queue warnings and management schemes. The variation in jam densities under adverse weather condition might be attributed to drivers' attempt to keep larger and safer spacing with the leading vehicle. More specifically, the results show that rainy and snowy weather induce drivers not only to drive slower but also to keep a larger distance headway to the leading vehicle (i.e. smaller jam density). On the other hand, in clear weather conditions, drivers are willing to accept shorter space and time headways to the leading vehicle. Unfortunately, there is a lack of research on human factors to further support such findings. More research needs to be conducted on car following models under different weather conditions, to further explain our findings.

According to the regression analysis results, the R-squared value is high for the free flow part and it is acceptable for the congested part except for the snowy condition. The low R-squared value corresponding to the snowy days might be explained by the different intensity levels of snow precipitation such as heavy snow, moderate snow, snowy/rainy, etc., however, all these levels were aggregated under snowy condition. Using more data points and more accurate weather condition information may help to improve the regression results.

2.4 Stochastic breakdown and capacity analysis

In this section pre-breakdown flows are analyzed to investigate the stochastic nature of breakdown phenomenon and freeway capacity. For this purpose, aggregated data in 15 min time intervals is used. Breakdown events are identified based on the speed threshold of 70 km/hr, which was selected based on the FFS (Free Flow Speed) curves for multi-lane highways and according to prior similar studies (Brilon et al., 2005; Elefteriadou & Lertworawanich, 2003; Ozguven & Ozbay, 2008; HCM 2010). Speed threshold can be calibrated through experiments for different locations, weather conditions, and based on the desirable level of service. A combination of speed with other traffic flow parameters such as density, flow, and occupancy may provide a more robust basis to identify breakdown phenomenon based on classic flow theories (Wu et al., 2010). However, it is not consistent with empirical findings of three-phase theory in which traffic breakdown (F to S transition) occurs at a critical synchronization speed (Kerner, 2009). The aggregated flow for the last time interval before breakdown is considered as pre-breakdown flow. The analysis is conducted for the full data set and for the categorized data set based on weather conditions.

2.4.1 Probability distribution for the full data set

In the first step, the empirical cumulative distribution function is plotted and smoothed using the Gaussian Kernel method to calculate the density of data using R programing software. Based on the observation of the smoothed density function and the empirical cumulative distribution, and the nature of the sample space of pre-breakdown flows, it is statistically reasonable to assume that the sample belongs to an extreme value distribution.

The Weibull distribution is one of the most popular life distributions with a flexible shape which enables it to model a wide range of failure events, and it can be theoretically derived as an extreme value distribution. The Weibull distribution is selected in this study to model the probabilistic characteristics of breakdown events (Reiss and Thomas 2007). Probability distribution function (PDF) and cumulative distribution function (CDF) for three-parameters Weibull distribution are shown in equations (2.1) and (2.2) respectively. The most likelihood estimation (MLE) method is used to calculate the distribution parameters, and several basic tests are conducted as diagnostics (Meeker and Escobar 1994).

$$P(x) = \frac{\lambda}{\mu}\left(\frac{x-\sigma}{\mu}\right)^{\lambda-1} exp(-\left(\frac{x-\sigma}{\mu}\right)^{\lambda}) \tag{2.1}$$

$$C(x) = 1 - exp(-\left(\frac{x-\sigma}{\mu}\right)^{\lambda}) \tag{2.2}$$

Where,λ, μ, and σ are shape, scale and location parameters.

In MLE, a likelihood function is the probability of observed data written as a function of distribution parameters including shape, scale, and location which is shown in equations (2.3) to (2.6). Thereafter, model parameters are calculated to maximize the likelihood function (Meeker and Escobar 1994). The transformation technique is used to simplify the calculation and to improve the accuracy of results given the number of data points (200 points). Thus, the Weibull model is transformed into the Gumbel model and MLE is used to derive the parameters. The analysis results are summarized in Table 2.5 and PDF plot and CDF plot for the full data set is derived as equations (2.5) and (2.6) respectively.

$$L(\lambda, \mu, \sigma | x) = \prod_{i=1}^{n} f(x_i | \lambda, \mu, \sigma) \tag{2.3}$$

Where,n is the number of samples. The log of the likelihood function is simplified to:

$$logL(\lambda, \mu, \sigma | x) = -n\lambda \log(\mu) + nlog(\lambda) + (\lambda - 1) \sum_{i=1}^{n} \log(x_i - \sigma) - \mu^{-\lambda} \sum_{i=1}^{n} (x_i - \sigma)^{\lambda} \tag{2.4}$$

$$P(x) = \frac{-0.33}{289.2} \left(\frac{x - 1462.4}{289.2} \right)^{-0.33-1} exp(- \left(\frac{x - 1462.4}{289.2} \right)^{-0.33}) \tag{2.5}$$

$$C(x) = 1 - exp(- \left(\frac{x - 1462.4}{289.2} \right)^{-0.33}) \tag{2.6}$$

The estimated distribution function is compared to the empirical distribution function to conduct the diagnostics through generating the probability plots, quantile plots, density plots and return period plots (Figure 2.3). The comparison of the calibrated model results with the empirical results in all four plots reveals the adequacy of the model.

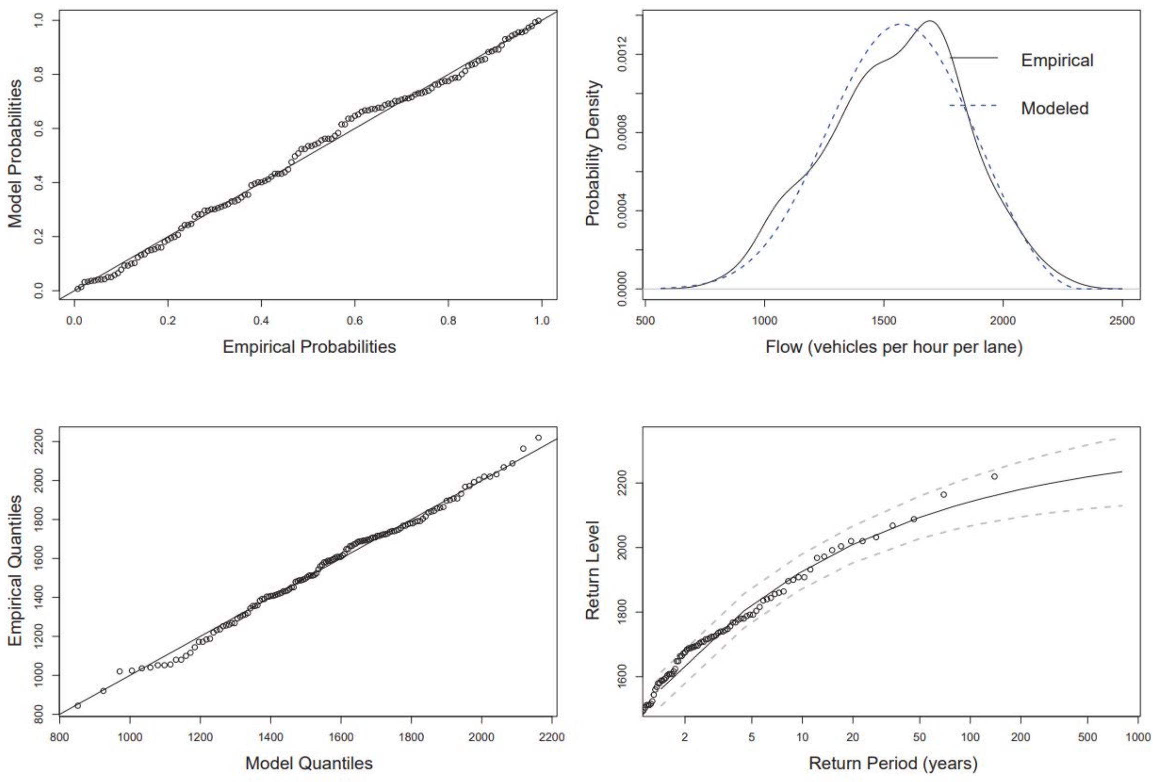


Figure 2.3. Diagnostics for the fit distribution function of all data

2.4.2 Probability distribution under different weather conditions

The above approach is applied for the data classified under the four described weather conditions to evaluate and compare the stochastic behavior of capacity under different weather. Since there are only five data points in rainy condition, the model is derived for only the clear, snowy and low visibility categories. Cumulative probability and density plot for the full data set and for different weather conditions are shown in Figure 2.4.

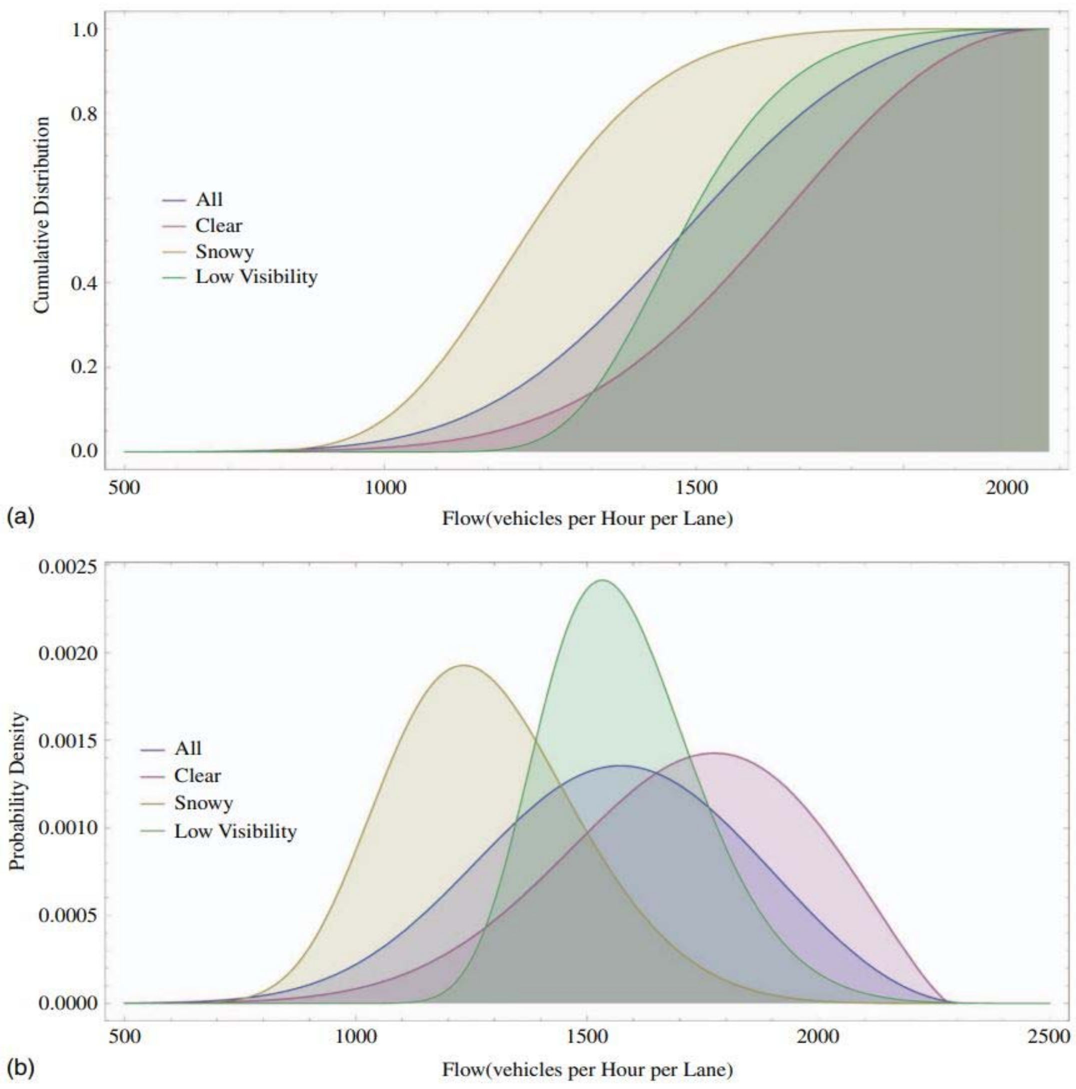


Figure 2.4. Cumulative probabilities and density plots for various weather condition (a & b)

The parameter estimation results are summarized in Table 2.5. The results show that Weibull distribution fits pre-breakdown flows for all weather conditions; however, the parameters are different. The Kruskal-Wallis rank sum test is conducted to evaluate the similarity between probability distribution parameters for the full data set and for the parameters derived for different weather conditions. The test results of Kruskal-Wallis chi-squared = 33.56, degree of freedom = 3, and p-value = 2.45e-07 showed that the null hypothesis of similar distributions for the sample sets is rejected and it is concluded that the sample of full data set and the samples for clear, snowy, low visibility conditions are from different distributions.

Table 2.5. Estimated parameters and standard error of Weibull distributions

Weather condition	Shape	Location	Scale
All conditions	-0.33	1462.4	289.2
Clear	-0.44	1627.1	289.3
Snowy	-0.17	1197.6	193.8
Low Visibility	-0.14	1510	154.0

The scale parameter (μ) describes the dispersion of the data. A larger value of scale parameter results in a distribution that is more spread out. Results of this study show that the distribution of the pre-breakdown flows for clear condition and the full data set is more spread out compared to the snowy and low visibility conditions. The resulting reduction in dispersion of the pre-breakdown flows under adverse weather conditions can be attributed to the fact that, under inclement weather conditions the majority of the drivers tend to reduce their speed, adopt a safer following distance and avoid lane changing. These resulting driving behaviors lead to a more synchronization in driving behavior and thus a reduction in the stochasticity of observed pre-breakdown flows. Thus, the breakdown will occur in the shortest range of traffic flows.

A summary of the results' analysis is illustrated in Table 2.6, which compares the capacity estimation using the fundamental diagram to the probabilistic capacities under various weather conditions. Comparing the results of constant capacity from the fundamental diagrams for different weather conditions shows that the capacity decreases about 324 and 296 veh/hr/l from clear weather to snowy and rainy weather conditions, respectively. Ignoring such differences in freeway control strategies such as ramp metering and variable speed limit results in reduction in their functionality.

Stochastic capacity analysis reveals even more fluctuations in freeway capacity under different weather conditions. If the capacity is considered as the 95% quantile of the probabilistic model, the freeway capacity is 2108 veh/hr/l for clear weather condition, and 1881 and 1650 veh/hr/l for low visibility and snowy conditions, respectively. Comparing these results to those corresponding to the full data breakdown capacity of 2007 veh/hr/l shows a reduction of 357 veh/hr/l in capacity in snowy condition.

Table 2.6. Deterministic and stochastic capacities for various weather conditions

Analysis approach	Probability	Clear	Rainy	Snowy	Low visibility	All conditions
Constant capacity from FD	NA	2288	1992	1964	2208	2288
Probabilistic capacity	95%	2108	NA	1650	1881	2007
	75%	1905	NA	1415	1685	1757
	50%	1725	NA	1266	1565	1562

The PDF and CDF plots also show that the probability of breakdown has a considerable fluctuation under different weather conditions and also as compared to the full data set. The comparison clearly shows that, modeling the stochastic behavior of freeway capacity under different weather conditions provides a more realistic illustration of traffic-carrying ability of freeways. For instance, for a given traffic flow of 1500 veh/hr/l, while the corresponding probability of breakdown is 41% for the full data set, it is only 20%, for clear weather condition 34%, for low visibility condition and as high as 78% in snowy weather condition. In other words, there is a 78% chance that the freeway capacity is below 1500vh/hr/l in snowy weather. This finding is significant since if the full data set is considered to model the stochastic capacity, it results in an over-estimation of capacity for snowy condition which will diminish the effectiveness of freeway control schemes and ultimately results in traffic breakdown on freeways.

The outcomes of this chapter is used as a solid empirical basis to calibrate a proper probabilistic model of pre-breakdown flows. This model informed the modelling and simulation analysis in Chapter 4.

CHAPTER 3: A STOCHASTIC MICROSCOPIC BASED FREEWAY TRAFFIC STATE AND SPATIAL-TEMPORAL PATTERN PREDICTION IN A CONNECTED VEHICLE ENVIRONMENT

3.1 Introduction

3.1.1 Background and motivations

Traffic estimation and prediction models play a crucial role for both road users and traffic managers as these models are the primary input to traffic management and control systems. The stochastic nature of driving behavior often is the root cause of fluctuating traffic patterns and traffic flow characteristics. For instance, in heavy traffic, microscopic triggers, such as changing lanes or suddenly hitting the brakes, create a domino effect of stop-and-go waves (Khondaker & Kattan, 2015; Zheng et al., 2011). These stochastic behaviors are the core mechanisms for the transition from uncongested to congested traffics states which can be further amplified under adverse weather conditions (Heshami et al., 2019; Suh & Yeo, 2016); thereby suggesting the need to incorporate stochasticity into traffic prediction models.

According to the empirical findings of the three-phase theory developed by Kerner (2002), traffic patterns are categorized into: free flow (F), synchronized flow (S), and wide moving jam (J). Transitions between the traffic phases, such as F to S or S to J, are stochastic phenomena that are triggered and caused by driver decisions such as lane changing, over-acceleration, or speed adaption (Kerner, 2002; Kerner & Klenov, 2003). The downstream front of a synchronized flow is fixed at a bottleneck, and vehicles move at a synchronized steady speed slower than the free flow speed. In contrast, a wide moving jam is triggered by drivers' over deceleration behavior with an initial synchronized flow while the downstream front propagates upstream with a steady speed.

Identifying and predicting these spatial-temporal patterns provides valuable information to determine the onset of congestion and associated phase transition from pre-congested to congested patterns; this input could be vital for advanced control schemes. Considering spatial-temporal pattern predictions, which is still largely overlooked in the literature, suggests using microscopic traffic prediction models.

This research develops a stochastic microscopic model that predicts traffic parameters including flow, speed, and travel time based on the three-phase theory using microscopic trajectory data from connected vehicles. To make the prediction more robust for the next time intervals, an adaptive Kalman filter (AKF) is used as a data fusion tool to combine the noisy measurements of trajectory and travel time data from CAVs and other floating car data resources (e.g., taxi, uber, etc.) with traditional fixed traffic detector data. It is to be noted that only the vehicle to Infrastructure (V2I) connectivity feature of CAVs is considered. In other words, this thesis only considers that CAVs share their speed, location and positions with a Road Side Unit (RSU). Utilizing and fusing complementary stationary detector and floating car data is shown to significantly improve the performance of prediction algorithms. In addition, forecasting of traffic objects (FOTO) and automatic tracking of moving traffic jams (ASDA) models, developed by Kerner et al. (2005), are used to dynamically predict traffic patterns and locations of jam fronts. The online application of FOTO and ASDA models, where parameters are calibrated offline, on various real freeway networks showed the effectiveness of these models. The developed analytical model in this research is unique in that it provides lane by lane stochastic prediction of flow and speed information for multiple timesteps ahead.

3.1.2 Review of the previous studies

In the literature, Oh et al., (2015) classified short-term traffic state prediction models into three broad categories: parametric data-based, non-parametric data-based, and model-based analytical methods. These approaches can be further categorized based on the study area (i.e., arterial roads versus freeway), application (i.e., route guidance versus real-time traffic control), and data type and availability (i.e., stationary traffic detectors, probe vehicles, etc.). The reader can refer to Vlahogianni et al. (2014) for a detailed review of the literature.

Parametric data-based approaches include linear regression and time series such as auto-regressive integrated moving average (ARIMA) and the Kalman filter (KF). ARIMA methods apply a smoothing filter to estimate average traffic conditions, thereby missing transitions from free flow to stop-and-go conditions and vice versa. These oscillations in traffic states are non-stationary phenomena that cannot be captured by ARIMA methods, but they are important traffic behaviors to capture (Vlahogianni et al., 2014). KF processes include real-time observations sequentially following an autoregressive relation and allow additional measurements to be incorporated. Another important feature of the KF is its ability to fuse various sources of data and to consider noisy measurements (e.g., during incidents). The KF has been successfully used for short-term traffic prediction for recurrent congestion (Xie et al., 2007). The main issue in utilizing a basic KF in traffic estimation and prediction is considering known and fixed noise statistics from historical data. Even if carefully pre-identified via offline tuning, the traffic state estimator may deviate under sudden endogenous changing stochastic traffic fluctuations (e.g. vehicle coming to a sudden stop) as well as exogenous conditions (e.g. collision). As a remedy, adaptive KF processes were introduced for more robust traffic state prediction (Guo et al., 2014b).

Non-parametric data-based methods include the application of artificial intelligence (AI) and pattern searching techniques (e.g., neural networks, Kth nearest neighbor, and deep learning) using historical data and real-time data (Cai et al., 2016; Kumar et al., 2013; Oh et al., 2015; Polson & Sokolov, 2017; Y. Wu et al., 2018). Despite increased prediction accuracy, these data-driven models do not fully reflect inherent traffic characteristics as they cannot explain the underlying traffic phenomena as incorporated in spatial and temporal traffic behavior. Incorporating this behavior is necessary to explain and model the underlying stochastic and fast-changing traffic flow behaviors that can result from either abrupt disturbances caused by non-recurrent congestion, weather conditions, and/or control measures such as ramp metering rates and variable speed limits that can change traffic conditions. Moreover, data processing, parameter calibration, and prediction modules are time-consuming processes, and therefore, the transferability and application of data-driven approaches in real-time control strategies is limited.

Model-based approaches, also known as analytical traffic state prediction approaches, are adaptable and responsive to the dynamic changes in traffic conditions (Oh et al., 2018; Vlahogianni et al., 2014). The advantage of these models lies in their explanatory power and capability of providing insights into examined systems that are typically harder to obtain from their data-driven counterparts. Model-based traffic prediction models are mainly classified into macroscopic and microscopic approaches. These approaches can in turn be either deterministic or stochastic.

In macroscopic models, a roadway is subdivided into homogenous segments and the time is discretized into short time intervals. The aggregated traffic state parameters need to be calibrated for each segment and updated for each time step. To achieve accurate estimations, the segments need to be short (300 m- 500 m) and traffic data must be collected for each segment. Deterministic macroscopic models such as the Cell Transmission Model (CTM) and Burgers equation have been used to develop travel time prediction methods (Chow et al., 2009; Kachroo et al., 2001; Oh et al., 2018). Another commonly used macroscopic model in traffic state prediction literature is the METANET model, which is a discretization and modification of the Payne model. The METANET calibration and validation studies have shown its effectiveness in traffic simulation and prediction applications, especially where the geometry of a corridor and traffic conditions are simple. However, a corridor with complex geometric characteristics and traffic conditions may require segment-specific parameter values to reflect segment-specific behaviors, and this requirement needs a large data collection and significant calibration efforts (Wang et al., 2018).

Lane-level traffic management is an important tool to alleviate congestion by balancing lane-flow distribution. So far, most macroscopic models focus on overall section traffic dynamics with no consideration to individual lane flows. Studies showed that fundamental diagram parameters vary significantly among lanes, and aggregated parameters may result in a non-equilibrium fundamental diagram attributed to mixing different traffic states. Thus, lane-specific behavior analysis is critical to predict the occurrence of breakdown at bottlenecks and the propagation of congested patterns over lanes (Duret & Audin, 2009; Pan et al., 2019; Shiomi et al., 2015). Bekiaris-Liberis et al. (2016b, 2017) proposed a macroscopic traffic estimation model using CAVs to estimate lane by lane density and overall speed and flow for a 400 m long freeway segment. In a recent paper, (Nagalur Subraveti et al., 2019) extended the original CTM to model and manage the flow dynamics on a lane by lane basis where lane change rates are computed as a function of various incentives such as maintaining route, keep-right bias, changing to lower density lanes, etc. The results showed the sensitivity of density estimations to FD parameter; yet the application of this model to traffic state prediction has not been investigated (Nagalur Subraveti et al., 2019).

A few non-parametric data-based models attempted to predict lane-level traffic state parameters based on AI techniques (Gu et al., 2019; Ke et al., 2020; Raza & Zhong, 2017). A fusion deep learning approach proposed by Gu et al. (2019) combined long short-term memory (LSTM) and gated recurrent unit (GRU) neural networks to learn the spatial-temporal correlations of lane section variables from the historical speed series information. Comparison of the results with benchmark models such as ARIMA, LWR and MLP showed the superior performance of this model for short-term 2- minute speed prediction on a lane- level basis.

Stochastic macroscopic models were introduced to incorporate the probabilistic nature of traffic flow behavior. Sumalee et al. (2011) extended the CTM to estimate stochastic freeway traffic states based on stochastic fundamental flow–density diagrams and stochastic travel demand. The developed stochastic CTM (SCTM) was adopted as a network loading model to generate traffic flow profiles from traffic data collected by detectors to capture the randomness in both demand and supply. The first-in-first-out (FIFO) concept was extended to estimate travel time distribution based on the stochastic cumulative inflow and outflow curves. The SCTM was successful in representing a more realistic model of stochastic traffic flow behavior compared to traditional macroscopic models, especially in the case of incidents (Sumalee et al., 2011, 2013). Extending deterministic macroscopic models to stochastic macroscopic models improved the ability of such models to represent dynamic and stochastic traffic behavior. However, they still suffer from several restricting features such as numerous boundary variables for subsystems, simplifying assumptions regarding various traffic modes and the probability of occurrence of each mode, inability to address non-recurrent conditions, and limitations in adapting to severe environmental conditions.

A few studies have explored the use of microscopic traffic flow models to estimate and predict traffic conditions over a short prediction horizon. Most of these studies used cellular automaton (CA) models as a simulation tool that combines present traffic measurements with historical data to estimate future traffic conditions (Chrobok et al., 2002; Miska, 2007; Schreckenberg & Wahle, 2001). Liu et al. (2006) used the CORSIM microscopic traffic simulator combined with a KF to predict travel time. The nearest neighbor method and a decision tree were also used to predict traffic volumes. Sunderrajan et al. (2016) developed an agent-based microsimulation approach to estimate traffic flow and density on homogenous expressways using floating car data. Intelligent driver model (IDM) and the general model of Minimizing overall braking induced by lane change (MOBIL) were utilized to reproduce acceleration and lane changing behaviors, respectively (Treiber et al., 2000; Treiber & Kesting, 2018). Recently, Treiber & Kesting (2018) extended their original IDM to reflect stochasticity of traffic flow oscillations due to three generic mechanisms including string instability, external white acceleration noise, and action point thresholds. The authors provided valuable insight into the importance of considering heterogeneous driving characteristics and instabilities arising from lane changing behaviors in traffic estimation.

Inspired by the three-phase theory, a few studies explored traffic speed and travel time estimation and prediction approaches. Rempe et al., (2016) proposed a Phase-based Smoothing Method for traffic speed estimation that was able to distinguish three traffic phases. Rehborn and Palmer (2008) compared travel time estimation for current traffic conditions from FOTO and ASDA models to real historical time series by calculating a quality index as a measure of accuracy. The developed model was able to reconstruct traffic patterns and predict travel time for time horizons shorter than one minute. In another three-phase theory-based method moving bottlenecks were predicted based on the phase transition events detected by probe vehicles (Wegerle et al., 2019). Tian et al. (2015) reproduced Kerner's three-phase theory findings including reconstructing synchronized flow and wide moving jams and their transitions through a new cellular automaton model. In this model, the space gap between vehicles oscillated around a desired space gap rather than considering a deterministic space gap used by the FD approaches. Deng et al. (2013) analyzed probe data on data cubes, which are tools that organize data sets into multidimensional aggregations with respect to the data dimensions and user-specified aggregation hierarchies. Data cubes were used to identify traffic "congestion events" as dynamic spatial-temporal progresses that were then aggregated at different levels of granularity. The proposed model was effective in identifying recurrent congested patterns. However, the model required extensive historical floating data to identify the similar congested patterns, and it was unable to capture non-recurrent congestion events.

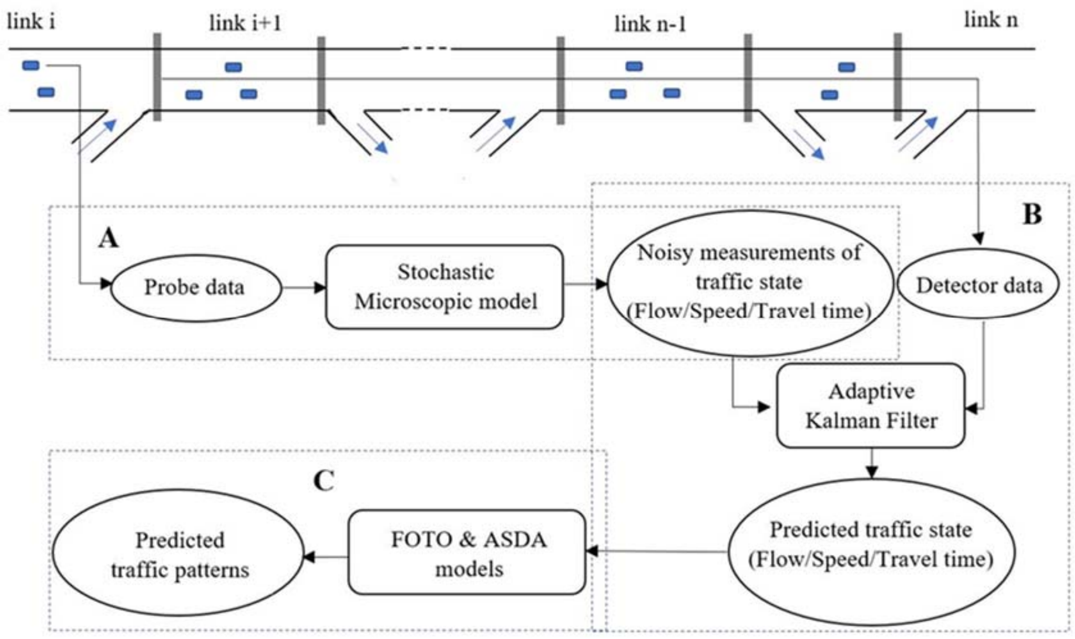
3.2 The proposed Traffic state prediction and pattern tracking approach

A schematic sketch of the proposed procedure is illustrated in Figure 3.1. The traffic prediction model developed in this research consists of three main modules:

Module A - The online simulation module that represents the surveillance system database that feeds the system with observation data from different sources such as CAVs and detectors. As explained in more detail in section 3.2.1, the online simulation module receives floating car data including location and speed and, thus, travel time. This module also simulates vehicle movements and estimates the traffic parameters for the required time interval for the location of downstream detectors. This module's outputs are noisy measurements of traffic state parameters, including flow, average speed, and link travel times on a lane by lane basis.

Module B - The data fusion, and traffic parameter prediction module receives the noisy measurements of traffic state parameters for the detector location from Module A. An AKF is used to fuse the received data with detector measurements. The outputs are the predicted traffic state parameters for the following time interval.

Module C **-** Spatial-temporal traffic pattern tracking, and prediction module uses the predicted traffic state parameters in FOTO and ASDA models to detect synchronized flow and wide moving jams. Moreover, this module dynamically identifies the predicted location of jam fronts and their resulting propagation on freeways. The mathematical models for each module are discussed below:



link i
link i+1
link n-1
link n
A
Probe data
Stochastic Microscopic model
Noisy measurements of traffic state (Flow/Speed/Travel time)
B
Detector data
Adaptive Kalman Filter
Predicted traffic state (Flow/Speed/Travel time)
C
FOTO & ASDA models
Predicted traffic patterns

Figure 3.1. Schematic sketch of the proposed traffic state and travel time prediction model

3.2.1 Module A: Online simulation

Module A in Figure 3.1 uses the Kerner-Kelnov stochastic microscopic model (KK model) (Kerner & Klenov, 2003) to simulate traffic conditions on a freeway segment in real time. KK is a stochastic microscopic three-phase traffic model that represents stochastic driver behaviors and their impact on traffic flow behavior, especially in freeway bottlenecks. The reader is referred to Kerner and Klenov (2003) for more detailed information regarding the KK stochastic microscopic model. In this module, vehicles' speed and location information are received from CAVs and other probe vehicles. Thus, vehicles' movements for each simulation time step (τ) are simulated based on the car-following rules below:

$$v_{n+1} = \max\,(0, \min\,(v_{free}, v_{c,n}, v_{s,n}) \tag{3.1}$$

$$x_{n+1} = x_n + v_{n+1}\tau \tag{3.2}$$

$$v_{c,n} = \begin{cases} v_n + \Delta_n & at \quad g_n \leq D_n \\ v_n + a_n\tau & at \quad g_n > D_n \end{cases} \tag{3.3}$$

$$\Delta_n = \max\,(-b_n\tau, \min\left(a_n\tau, v_{l,n} - v_n\right)) \tag{3.4}$$

$$D_n = d + G(v_n, v_{l,n}) \tag{3.5}$$

$$G(v_n, v_{l,n}) = \max\,(0, c\tau v_n + \beta a^{-1} v_n\left(v_n - v_{l,n}\right)) \tag{3.6}$$

Where x_n and v_n are the vehicle's location and speed, $a_n \geq 0$ and $b_n \geq 0$ are acceleration and deceleration, and $v_{free}, v_{c,n}, v_{s,n}$ are free flow speed, desirable speed, and safe speed, respectively. g_n is the space gap from the leading vehicle, and D_n is the synchronization distance, which is the distance at which a vehicle tends to adjust its speed in accordance with the preceding vehicle's speed. $k \geq 1$, β, and a are constants, and safe speed ($v_{s,n}$) is calculated based on (3.7) to (3.12):

$$v_{s,n} = \min\,(v_n^{(safe)}, \frac{g_n}{\tau} + v_l^{(a)}) \tag{3.7}$$

$$v_n^{(safe)} = v^{(safe)}\left(g_n, v_{l,n}\right) = b\tau(\alpha_{safe} + \beta_{safe}) \tag{3.8}$$

$$\alpha_{safe} = \left\lfloor \sqrt{2\frac{d_{p,n} + g_n}{b\tau^2} + \frac{1}{4}} - \frac{1}{2} \right\rfloor \tag{3.9}$$

$$d_{p,n} = b\tau^2(\alpha_p\beta_p + \frac{\alpha_p(\alpha_p - 1)}{2}) \tag{3.10}$$

$$\beta_{safe} = \frac{d_{p,n} + g_n}{(\alpha_{safe} + 1)b\tau^2} - \frac{\alpha_{safe}}{2} \tag{3.11}$$

$$v_l^{(a)} = \max(0, \min(v_{l,n}^{(safe)} - a\tau, v_{l,n} - a\tau, \frac{g_{l,n}}{\tau} - a\tau) \tag{3.12}$$

Where α_p is the integer part of $\frac{v_{l,n}}{b\tau}$, β_p is the fractional part of $\frac{v_{l,n}}{b\tau}$, and $v_l^{(a)}$ is the anticipation speed for the leading vehicle (Krauss et al., 1997). Stochastic driver behaviors and random deceleration and acceleration decisions and their impacts on vehicular motions are modeled based on the following rules:

$$S_{n+1} = \begin{cases} -1 & if\ \tilde{v}_{n+1} < v_n - \delta \\ 1 & if\ \tilde{v}_{n+1} > v_n + \delta \\ 0 & otherwise \end{cases} \tag{3.13}$$

Where S_{n+1} shows whether the vehicle decelerates ($S_{n+1} = -1$), accelerates ($S_{n+1} = 1$), or maintains its speed ($S_{n+1} = 0$) by comparing the expected speed ($\tilde{v}_{n+1}$) to the current speed (v_n). δ is a constant ($\delta << a\tau$).

Thereafter, random acceleration or deceleration is modeled as follows:

$$\xi_n = \begin{cases} -\xi_b & if\ S_{n+1} = -1 \\ \xi_a & if\ S_{n+1} = 1 \\ 0 & otherwise \end{cases} \tag{3.14}$$

Where ξ_b, ξ_a are impulsive random variables for deceleration and acceleration calculated by$\xi_b = a\tau\theta(\mathrm{p_b} - r)$ and $\xi_a = a\tau\theta(\mathrm{p_a} - r)$ where $\mathrm{p_b}$and $\mathrm{p_a}$are probabilities of random deceleration and acceleration; r =rand(0,1), $\theta(z) = 0\ at\ z < 0\ and\ \theta(z) = 1\ at\ z \geq 0$.

Random time delays in acceleration and deceleration are calculated based on the following stochastic functions:

$$a_n = a\theta(\mathrm{P_0} - \mathrm{r_1}) , b_n = a\theta(\mathrm{P_1} - \mathrm{r_1}) \tag{3.15}$$

$$\mathrm{P}_0 = \begin{cases} \mathrm{p}_0 & if\ S_n \neq 1 \\ 1 & if\ S_n = 1 \end{cases}, \mathrm{P}_1 = \begin{cases} \mathrm{p}_1 & if\ S_n \neq -1 \\ \mathrm{p}_2 & if\ S_n = -1 \end{cases} \tag{3.16}$$

Whenever a CAV's simulated coordinate is past the nearest downstream detector location, the vehicle is counted, and its speed is recorded. The average travel time from a previous upstream detector for CAVs is also recorded for each time interval. The macroscopic parameters are aggregated and estimated based on simulating the vehicles trajectories and thus, the vector *(y_k)*, representing the flow, speed, and travel time estimations, is obtained for multiple time intervals ahead.

3.2.2 *Module B: Data fusion and traffic parameter prediction*

A KF, which is an efficient recursive process that estimates and updates the state of dynamic systems, is a superior approach in traffic state estimation and prediction (Ojeda, Kibangou, & de Wit, 2013). This module develops an AKF process that receives information on the estimated aggregated traffic flow, speed, and travel time from the online simulator. Since the CAVs randomly enter the simulation according to their designated penetration rate, their actual observed rate is not available at each time step; thus, detector data is used to dynamically update the "observed" rate of CAVs in the AKF process. Once the state measurements become available, the estimations are updated, and the traffic state is predicted for the next time intervals. The model dynamics for the evolution of the macroscopic parameters are assumed to be following a random walk model. Such assumption might be a simplifying one but has been shown by previous researchers (Ojeda, 2014; Ojeda et al., 2013; Xia et al., 2011) to be effective for short-term traffic prediction purposes. Thereafter, the AKF is used for the three following processes: 1) filtering, 2) smoothing the impact of missing data by fusing detector information, and 3) multiple-step ahead prediction. The state transition as a linear discrete time stochastic process, as shown in equation (3.17).

$$x_{k+1} = x_k + W_k \tag{3.17}$$

Where x_k is the process's state vector at time step k. The process noise (W_k) is assumed to be Gaussian white noise with a covariance of Q. This assumption was tested using Kolmogorov–Smirnov test and based on state measurements of various simulation runs. Once the next measurement is observed, the AKF updates the estimates using the following measurement equation:

$$y_k = H_k x_k + V_k \tag{3.18}$$

Where y_k is the noisy measurement of the stae x at time k, and v_k is the measurement Gaussian noise with a covariance of R. The hypothesis of Gaussian distribution for the observation noise (V_k) was not rejected in the statistic test. The observation model (H_k) handles the impact of CAVs' penetration rate by dynamically updating the percentage of CAVs based on the detector measurements.

The a posteriori estimate of flow/speed $\hat{x}_{(k|k)}$ is calculated as follows:

$$\hat{x}_{(k|k)} = \hat{x}_{(k|k-1)} + K_k(y_k - H_k^T \hat{x}_{(k|k-1)}) \tag{3.19}$$

Where K_t is the updated Kalman gain equal to:

$$K_k = P_{(k|k-1)} H_k^T (H_k P_{(k|k-1)} H_k^T + \hat{R}_k)^{-1} \tag{3.20}$$

And the error covariance is updated to:

$$P_k = (1 - K_k H_k)\, P_{(k|k-1)} \tag{3.21}$$

The process is projected to the next time step with the following a priori estimates of error covariance and predicted parameters as follows:

$$P_{(k+1|k)} = P_k + \hat{Q}_k \tag{3.22}$$

$$\hat{x}_{(k+1|k)} = \hat{x}_{(k|k)} + \hat{q}_k \tag{3.23}$$

The AKF process is initialized based on the assumptions of $\hat{x}_{(k0|k-1)} = x_{k0}$, where $\hat{x}_{(k0|k-1)}$ is an a priori estimate of flow/speed, and x_{k0} is the actual detector measurement for the time interval *k₀*. The a priori estimate of the error covariance is assumed to be $P_{(k0|k-1)} = 1$.

The a priori statistics for stochastic errors are adaptively estimated based on the empirical estimators proposed by Myers & Tapley (1976). This method is developed based on a limited memory algorithm that adaptively estimates the a priori statistics of errors based on noise samples as intuitive approximations of true errors. The simplicity of this method and its efficiency in producing estimations with a limited number of samples makes it a promising candidate for online applications, and it has been used in many traffic state prediction studies in the literature to adaptively estimate noise statistics (Aljamal et al., 2020; Chu et al., 2005; Huang et al., 2018; Ojeda, Kibangou, & de Wit, 2013; Xia et al., 2011; Zhou et al., 2017).

Thus, measurement noise covariance R_k and process noise covariance Q_k for traffic flow and speed are updated based on equations (3.24) to (3.29). Choosing the number N of past observations is based on identifying the time frame in which the structural changes in traffic state parameters are considered. Review of the literature shows a range of sample sizes from N=4 (Ojeda etal., 2013) to N=20 (Zhou et al., 2017) of covariance matching methods. After testing sample sizes from N=5, increasing by 5, to N=30 and calculating the corresponding minimum mean prediction errors, N = 5 and N=15 was selected for the online noise statistics estimations for traffic flow and speed parameters respectively.

$$r_k = y_k - H_k\hat{x}_{(k|k-1)} \tag{3.24}$$

$$\hat{r} = \frac{1}{N} \sum_{k=k-N+1}^{k} r_k \tag{3.25}$$

$$\hat{R}_k = \frac{1}{N-1} \sum_{k=k-N+1}^{k} ((r_k - \hat{r})(r_k - \hat{r})^T - \frac{(N-1)}{N} H_k P_{(k|k-1)} H_k^T) \tag{3.26}$$

$$q_k = \hat{x}_{(k|k)} - \hat{x}_{(k-1|k-1)} \tag{3.27}$$

$$\hat{q} = \frac{1}{N} \sum_{k=k-N+1}^{k} q_k \tag{3.28}$$

$$\hat{Q}_k = \frac{1}{N-1} \sum_{k=k-N+1}^{k} ((q_k - \hat{q})(q_k - \hat{q})^T - \frac{(N-1)}{N} (P_{(k-1|k-1)} - P_{(k|k)})) \tag{3.29}$$

Module B’s outputs are predicted traffic flow and speed for a specified time horizon. In addition, total link travel time is predicted based on the predicted average travel time for CAVs multiplied by predicted traffic flow for each time interval.

For the multiple step ahead prediction purpose, where the detector measurements are not available, the simulated measurements from module A, for m time steps ahead, are considered as future observations of the traffic state parameters. While for each $\hat{x}_{(k+m|k)}$, $\hat{R} = \hat{R}_k$, and $\hat{Q}_k$ is calculated from (3.29).

3.2.3 Module C: Spatial-temporal traffic pattern tracking and prediction

Forecasting of traffic objects (FOTO) and automatic tracking of moving traffic jams (ASDA) models effectively reconstruct the spatiotemporal traffic patterns on real-world highway corridors without calibrating model parameters. The accuracy of the reconstructed patterns is highly dependent on the distance between detectors and the relative position of the detectors to the bottlenecks (B. S. Kerner et al., 2005; Rehborn & Palmer, 2008). In this module, predicted traffic flow and mean speed for two groups of detectors are utilized in FOTO and ASDA models to dynamically predict the traffic patterns. The first group consists of two loop detectors located at the upstream and downstream bottlenecks on a freeway link; theses detectors collect traffic parameters from all vehicles. The second group includes virtual data aggregation points [virtual spot detectors (VSDs)] that can be considered at desired distances along a freeway link. At these data collection and aggregation points, trajectory data from CAVs are collected and converted to predicted traffic state parameters, as described in Module A and Module B. The AKF adapts an estimation of noise covariances from the closest detector to project to a future time interval. This modification allows the model to utilize data from CAVs rather than from stationary measurements captured from infrastructure on the link. Thereafter, FOTO and ASDA models utilize the predicted traffic state information as follows:

First the traffic phase at each detector location is identified as follows:

FOTO detects free flow if $v_k > v_{syn}$.

FOTO detects synchronized flow if $v_k \leq v_{syn}$.

FOTO detects wide moving jam if $v_k < v_{jam}$ and $q_k < q_{jam}$.

Where v_k and q_k are predicted speed and flow, respectively, at the detector. If synchronized flow is detected, the upstream front of the synchronized flow $x_k^{S(up)}$ is determined as follows:

$$x_k^{S(up)} = \partial \Delta M(k) \tag{3.30}$$

$$\Delta M(k) = \Delta M(k)_{total}/n \tag{3.31}$$

$$\Delta M(k)_{total} = \sum_{t_{syn,B}}^{t} q_B(k) - \sum_{t_{syn,B}}^{t} q_A^*(k) \quad \text{at} \quad k > k_{syn,B} \tag{3.32}$$

$$q_A^*(k) = q_A(k) + q_{on}(k) - q_{off}(k) \tag{3.33}$$

Where ∂ is a model parameter representing the occupied segment of a freeway lane by each vehicle, including the space gap, in synchronized flow conditions, $q_A(k)$ is the predicted flow for the upstream detector, $q_{on}(k)$ is the on-ramp flow, and $q_{off}(k)$ is the upstream exiting flow, if located between detectors.

If a wide moving jam is detected, the upstream front ($x_k^{J(up)}$) and downstream front ($x_k^{J(down)}$) of the wide moving jam are detected as follows:

$$x_k^{J(up)} = -\sum_{k_{j,B}}^{k} \frac{q_A(k)-q_{min}}{\rho_{max}-(\frac{q_A(k)}{v_A(k)})} \quad for\ k \geq k_{j,B} \tag{3.34}$$

$$x_k^{J(down)} = -\sum_{k_{j1,B}}^{k} \frac{q_B(k)-q_{min}}{\rho_{max}-(\frac{q_B(k)}{v_B(k)})} \quad for\ k \geq k_{j1,B} \tag{3.35}$$

Where, $q_A(k)$ and $v_A(k)$ and, $q_B(k)$and $v_B(k)$ are the predicted flow rates and average speed at the upstream and downstream detectors, respectively; $k_{j,B}$ is the time instant when the moving jam is detected at the downstream detector; the time $k_{j1,B}$ determines the appearance of the downstream jam front at this detector; q_{min} and ρ_{max} are the minimum flow rate, and the maximum vehicle density within the jam, respectively.

3.3 Simulation results

The proposed traffic state and pattern prediction model is applied to a hypothetical freeway segment with a total length of 13 km, including two on-ramps located at the 5 km and 13 km marks and two off-ramps at 1.5 km upstream of the on-ramps. As shown in Figure 3.2, the main detectors (A and B) are located at the on-ramp bottlenecks, which are the desired locations to deploy traffic control devices (e.g., ramp meters), and thus, reliable prediction of traffic parameters is required. There are also two detectors that count the merging and exiting vehicles. Similar to Bekiaris-Liberis et al. (2016), virtual spot detectors (VSD1-VSD7) are located on the segment to collect CAVs' trajectory data for pattern tracking purposes. The distance between VSDs in this study is 1000 m; however, it can be adapted according to desired computation time, prediction accuracy, and homogeneity of the link, while the VSDs only attributed to the computational burden of the model and not to monetary costs. Vehicle motion, lane changing, and merging behaviors are simulated based on the Kerner-Kelnov (2003) stochastic three-phase model as described in Module A.

With the expected near future developments in CAVs, frequent updates of speed and positioning data will be available. Trajectory data from CAVs is assumed to be updated every 10 seconds. Other resolutions of data availability, including 5 and 30 seconds, were also tested. As expected, the results using a 5 second frequency showed higher accuracy at the expense of higher computation time, and the 30 second discrete time steps did not provide reliable predictions due to the microscopic characteristics of the proposed model. In this research, as a trade-off between computational time and prediction performance, a 10 second update interval is chosen. Whenever, the new trajectory data becomes available it is updated in module A, which helps to avoid error accumulation in the simulation process.

Model parameters for the stochastic microscopic model are illustrated in Table 3.1. Most of the parameters are adopted from the original model in Kerner-Kelnov (2003). Parameters such as probabilities of random acceleration and deceleration, and lane changing, are based on driver's behavioral analysis and classification studies for moderate risk-taker drivers by Li et al. (2017) for the base scenario. These parameters are later examined by way of sensitivity analysis to assess their impact on the model performance. For real-world applications, local driver behavioral models and vehicle compositions can be used to calibrate the model parameters.

The case study is designed to replicate peak period traffic conditions, and vehicles enter lanes according to a random log-normal distribution function, which has been suggested for congested traffic conditions in previous studies (X. Chen et al., 2010; Kong & Guo, 2016). The mean value of the distribution function changes every 20 minutes to reflect the increase and decrease in traffic flow at the beginning and ending of the peak period. CAVs are randomly tagged based on the specified penetration rates. Merging vehicles enter the segment at a uniform average rate with the rate changing every 20 minutes. Exiting vehicles are randomly tagged to exit from the off-ramps with exit probabilities of 10% and 40% for the first and second exits, respectively.

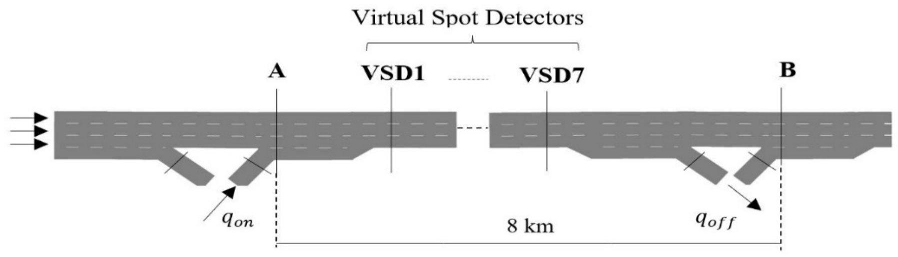


Figure 3.2. Freeway sketch and detector layout

3.3.1 Traffic state prediction results

Figures 3.3 to 3.7 show the traffic state prediction results for the first and second bottleneck locations, representing an 8 km segment. These figures include traffic flow, speed, total travel time, and average travel time flow per lane and the total predicted traffic flow for all lanes in 1-minute intervals for a total duration of 90 minutes for locations A and B, respectively. Traffic flow is reported as an hourly flow, which is calculated based on 5-minute traffic counts, and then converted to the hourly flow rate. Figures 3.4 and 3.6 illustrate the average predicted speed per lane and overall average speed of all lanes for the same time period at locations A and B, respectively. Total travel time and average travel time for the studied segment between two bottlenecks for 1-minute intervals are shown in Figure 3.7.

Table 3.1. Model parameters for the three-phase stochastic microscopic model

Parameter	Description	Value
τ	Simulation interval	1 s
v_{free}	Free flow speed	30 m/s
d	Vehicle length including minimum space gap	7.5 m
c	constant	1.5
β	constant	0.3
a	Maximum acceleration	1 m/s^2
b	Maximum deceleration	0.5 m/s^2
δ	Constant<< $a\tau$	0.01
P_a	Probability of random acceleration	0.3
P_b	Probability of random deceleration	0.2
p_0	Probability of random delay in acceleration	0.575+0.125*minimum $(1, v_n/10)$
p_1	Probability of random delay in deceleration	0.3
p_2	Probability of random delay in deceleration	0.48+0.32$\theta(v-15)$

The online simulator in Module A tracks all vehicle trajectories, and thus, collects traffic information from all vehicles, which is used as the ground truth data to evaluate the performance of the prediction model. The mean absolute percentage error (MAPE) index, one of the most common measures for evaluating the accuracy of prediction methods, is calculated based on the following equation:

$$MAPE = \sum_{t=1}^{n} \frac{|A_t - P_t|}{A_t} \tag{3.36}$$

Where A_t is the actual value of the traffic state parameter at time step t based on the information from all vehicles traveling the segment during the simulation period, and P_t is the predicted value of the same parameter.

The results show that the developed approach predicts traffic flow at 1-minute time intervals for the more congested bottleneck (location B) with accuracies of 7.3%, 4.6%, 3.8%, and

3.0% for the right lane, middle lane, left lane, and all lanes, respectively. MAPE is higher for the right lane, which is attributed to the expected higher traffic disturbances due to merging and existing vehicles. The results also show that prediction errors for per lane traffic flows are slightly higher compared to those in the case where all lanes are considered together. The higher error is due to vehicles that change lanes within the prediction horizon, which affects the per lane predictions, while the total flow for all lanes at the detector location stays unchanged. Owing to less disturbances at location A, which is the less congested bottleneck, higher accuracy in flow predictions is observed with the MAPEs of 3% to 3.8% throughout the lanes.

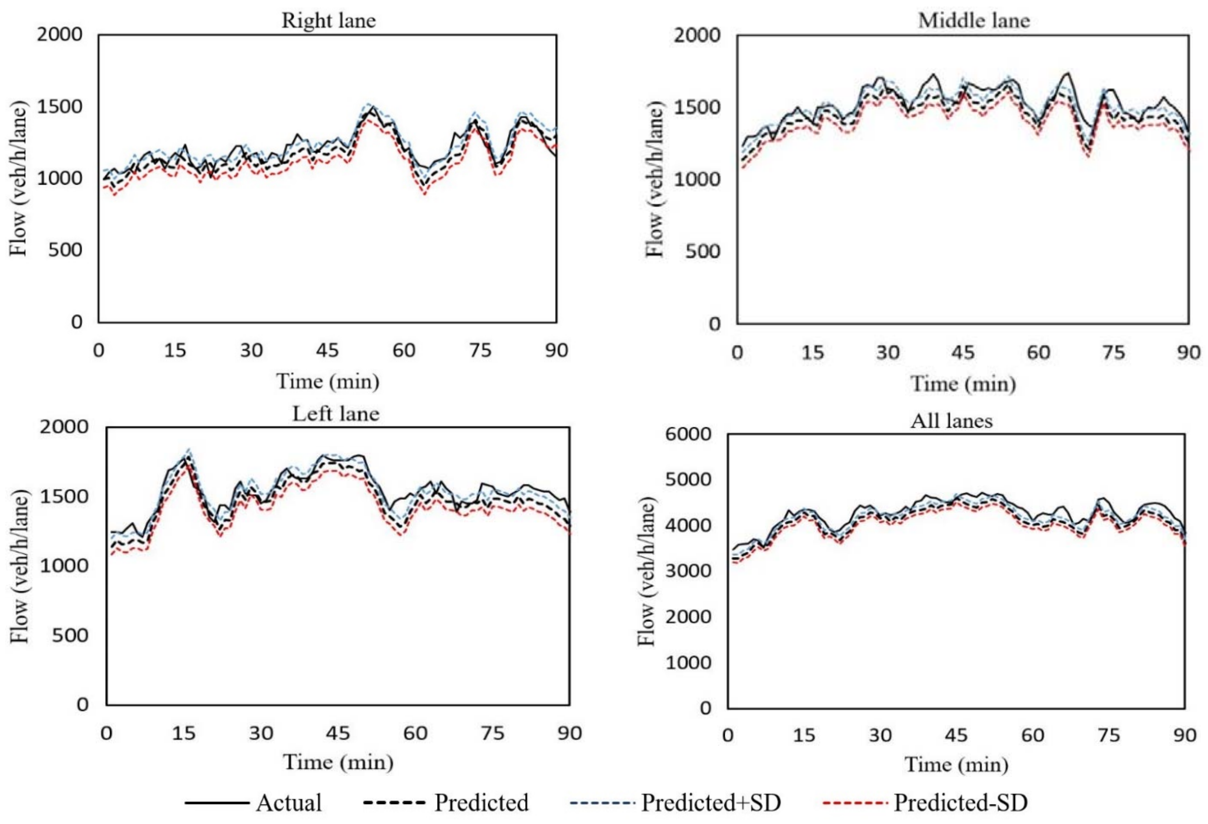


Figure 3.3. Traffic flow prediction for 1-minute intervals per lane and overall (Location A)

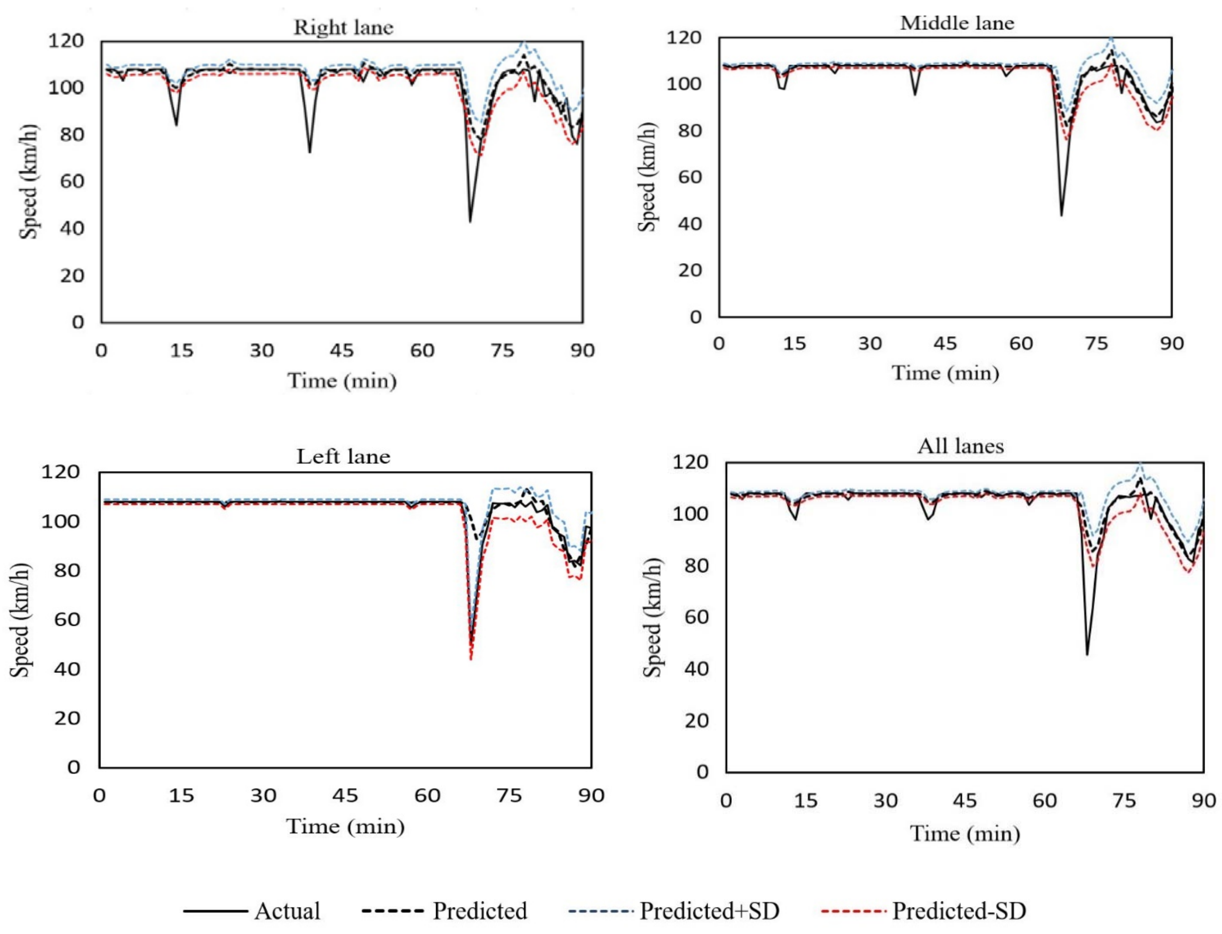
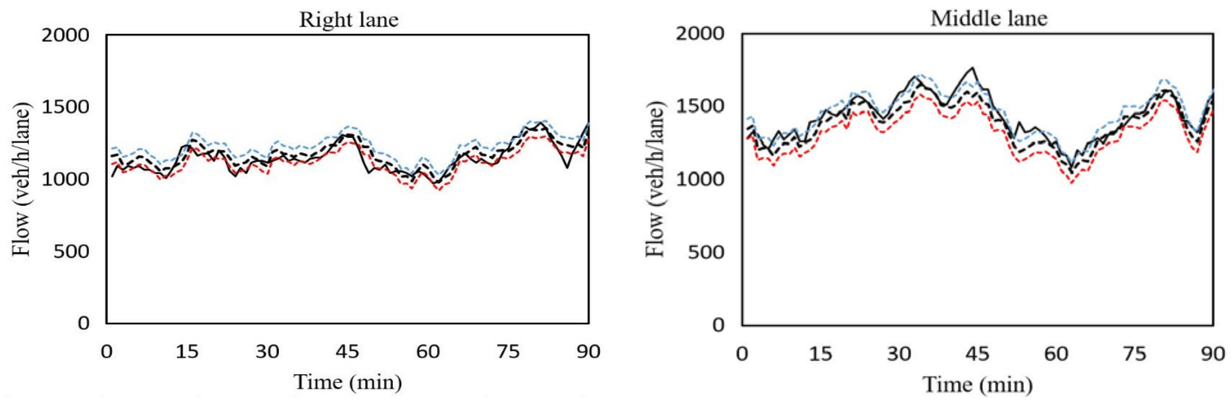


Figure 3.4. Average speed prediction for 1-minute intervals per lane and overall (Location A)



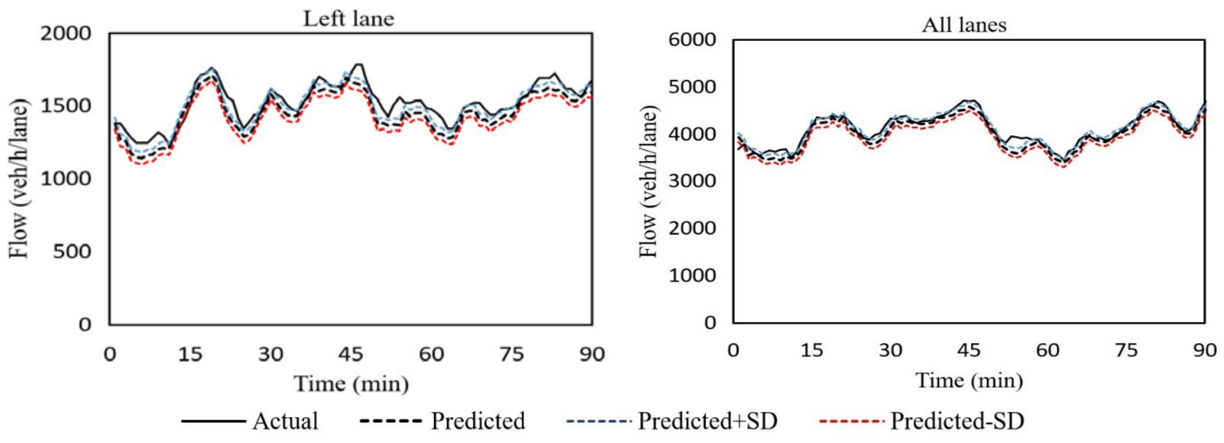


Figure 3.5. Traffic flow prediction for 1-minute intervals per lane and overall (Location B)

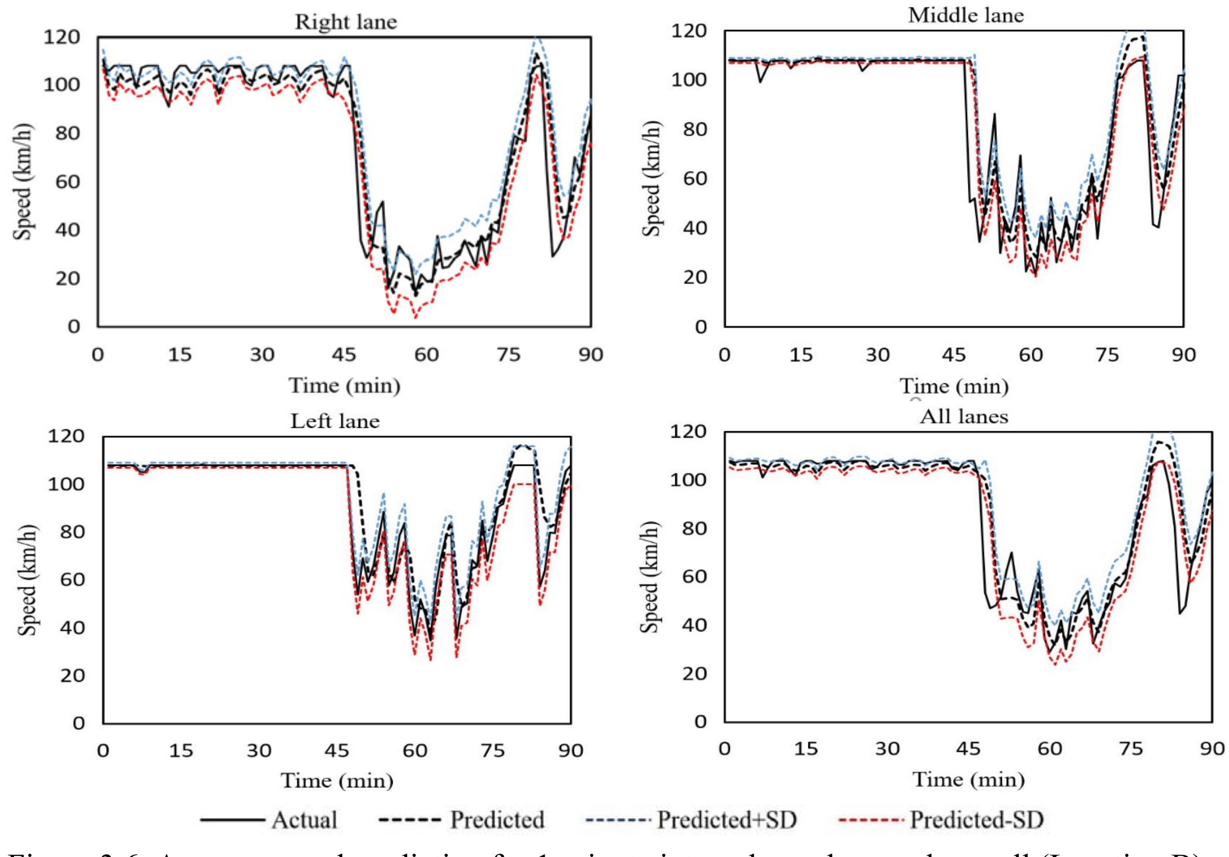


Figure 3.6. Average speed prediction for 1-minute intervals per lane and overall (Location B)

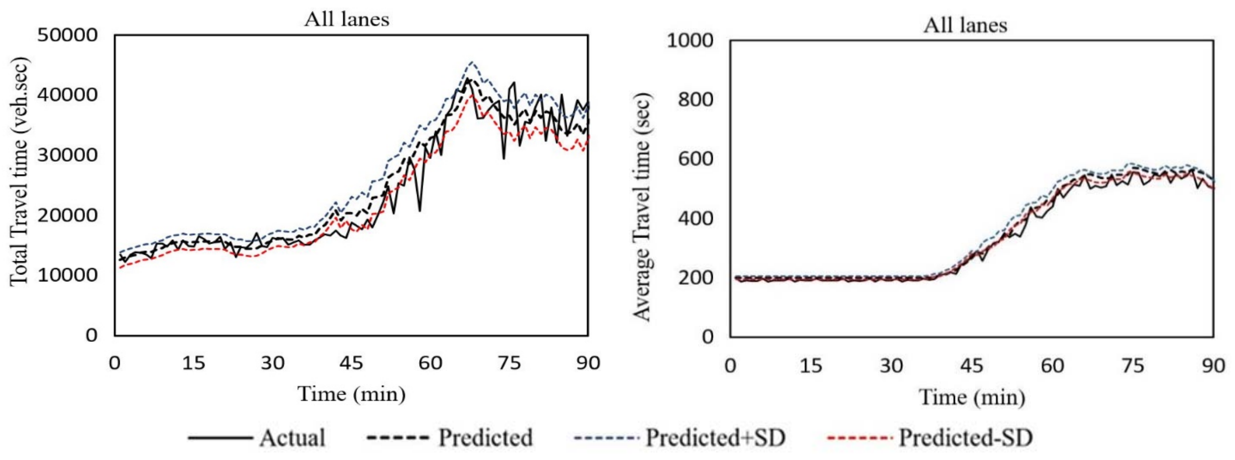


Figure 3.7. Total and average travel time predictions for 1-minute intervals for segment A-B

Figure 3.6 shows the "actual" and predicted average speed for the right, middle, and left lanes, as well as the average estimated speed of all lanes for location B. Average speed and its fluctuations vary among the lanes, resulting in varying levels of accuracy of 10.0%, 8.2%, 6.6%, and 6.6% for the right lane, middle lane, left lane, and all lanes, respectively. This result is expected due to the higher congestion level in the right lane. Here, merging vehicles from the on-ramp decrease the average speed of vehicles at the detector location to around 10 km/hr. However, the lowest average speed for all lanes remains around 30 km/hr. Moreover, a breakdown in the right lane happens at t= 44 min and propagates to the left lane at t=47 min, while the average speed for all lanes identifies the breakdown at t= 48 min. One of the important findings of this analysis is that despite the higher prediction performance, the average speed for all lanes is not an adequate or appropriate measure for identifying the severity and time of a speed breakdown, especially for the purpose of taking proactive control actions. In this case, for instance, relying on overall speed prediction results in a 5-minute time lag in identifying the onset of a breakdown. In contrast, right lane speed prediction is shown to be highly effective in identifying the onset of a breakdown; it has an accuracy within 1 minute.

Location A, shown in Figure 3.4, is configured at a lower congestion level to evaluate the model performance for such conditions. While the MAPEs for the speed predictions at location A stay below 4%, the sharp decrease in speed resulting from slow-moving vehicles, which are merging into the freeway, is not always fully captured by the prediction model. Such discrepancy between the "observed" and predicted speed that might be a limitation of the proposed model is mainly reported when the freeway is at free flow condition (i.e. density is lower than the critical density). Thereby, the stochasticity of driver behavior results is some vehicles merging at significantly low speed that cannot be fully predicted by the model. However, as shown in Figure 3.4, this low speed is not sustained for long time, as the vehicles accelerate to merge with the freeway at free flow speed.

The proposed model is also used to predict the total and average travel time for the segment between two bottlenecks (8 km). Total travel time for all vehicles that traverse the link and pass the detector location (in vehicle-second) is predicted within an accuracy of 8.6%, which is a higher error compared to the predicted average travel time (in seconds) with a MAPE of 4.7%. Total travel time predictions include an error component related to the vehicle count, which justifies the error difference compared to average travel times.

3.3.2 Sensitivity analysis

The impacts of the penetration rates of CAVs on the proposed model's performance are evaluated for 50%, 30%, 10% penetration rates and for different time step horizons. To gain a better understanding of the prediction errors for various scenarios, Root Mean Square Error (RMSE) indexes are also analyzed. A RMSE index provides a complementary understanding of the prediction error magnitude by illustrating the standard deviation of prediction errors. It is calculated as follows:

$$RMSE = \sqrt{\sum_{t=1}^{n}(A_t - P_t)^2/n} \tag{3.37}$$

However, increasing the prediction interval to 2 minutes increases MAPEs to around 10% for all penetration rates. Statistical analysis was conducted to evaluate the impact of the penetration rate of CAVs on MAPEs for overall flow predictions within a specific time interval (Table 3.2). These results show that in most cases, decreasing the penetration rate of CAVs does not significantly affect the accuracy of the total flow predictions, with the exception of when the penetration rate decreases from 50% to 30%. However, longer prediction intervals increase noisy measurements and ultimately impact the accuracy of predictions.

Per lane flow prediction results indicate that MAPEs fluctuate from 7.3% to 9.8% for the right lane, and the errors decrease for the middle and left lanes, which is expected because there are fewer traffic fluctuations on those lanes compared to the right lane where all merging and exiting maneuvers take place. Analyzing 60 simulation runs shows that, in most runs, noisy measurements of traffic counts are underestimated for the middle and the left lanes. This issue can be resolved by adding a parameter to the AKF equations that requires historical data and calibration of the noise function to increase the accuracy of per lane and total flow predictions.

The results for the speed predictions show that the proposed model is able to predict the average speed for all lanes with a MAPE of below 9% for all scenarios. A slight reduction in accuracy is observed by increasing the prediction time interval with MAPEs, which corresponds to around a 2% increase in MAPE in the worst-case scenario. According to the t-test results, shown in Table 3.3, penetration rates does not have a significant impact on the mean MAPE for all lane speed predictions. In contrast, per lane speed prediction results show a wide range of MAPEs, spanning between 6.6% for the left lane up to 16.0% for the right lane. Random fluctuations in demand, and more importantly, stochastic driver behaviors in speed adaption and merging maneuvers result in different traffic patterns affecting speed fluctuations and causing the wide range of per lane speed prediction errors.

Figure 3.8 shows that the accuracy of speed prediction is lower for the right lane in all scenarios. The MAPE for the right lane for 1-minute prediction intervals ranges from 10.0% to 13.9%. This issue is caused by drastic speed fluctuations at the merging region of the studied segment; these fluctuations strongly affect the right lane's speed profile. Another potential explanation behind the reduced prediction performance is the possibility of not observing any CAVs in cases with low penetration rates (10%). In these cases, the average speed is considered equal to the average speed of the previous time interval. This assumption results in higher error, especially if a drastic change has happened to the speed profile in the meantime due to a low-speed merging vehicle. Although the prediction error is relatively high for speed in the right lane, the breakdown phenomena are predicted on time for almost all scenarios.

Figure 3.9 shows the time that breakdowns happened in 30 simulation runs. All runs were based on 1-minute prediction intervals; however, the penetration rate of CAVs was 50% for runs 1 to 10, 30% for runs 11 to 20, and 10% for runs 21 to 30. In this research, the onset of a breakdown was when the average speed fell below 70 km/hr. Mean and variance of the probability distributions for traffic demand were the same for all runs. However, speed breakdowns happened at different times due to the stochastic driver behaviors (responding to traffic conditions). The times of the onsets of breakdowns were also different on three lanes for the same simulation run. The number at the left of each breakdown point in Figure 3.9 shows the time difference between the actual and predicted breakdown. For the right lane, in 19 out of 30 simulation runs, speed breakdown occurrence is correctly predicted at the same minute it occurred. For 8 of the simulation runs, the onset of breakdown is predicted one minute behind the actual occurrence, and in 3 cases, the prediction is one minute earlier.

For the middle and left lanes, the time differences were spread over a wider range from 0 to 5 minutes because when a breakdown occurred in the right lane, speed usually stayed low and congestion propagated upstream, until the traffic condition changed or a control measure was applied. However, in other lanes, when the average speed fluctuated around the speed threshold and a speed breakdown happened, sometimes there was an increase in the average speed and a fast return to the free flow condition. Thus, the prediction model may not capture these fluctuations, even if the MAPE is low. In general, despite the relatively low speed prediction performance of the right lane in terms of MAPEs (10.0% to 13.9%), the developed model is still able to identify and predict breakdown occurrence (i.e. when speed falls below 70 km/hr and this speed is sustained for 5 minutes) in the right lane in a timely manner. Such level of accuracy in the prediction is acceptable given the stochastic nature of traffic breakdown at bottlenecks. Consequently, the model can effectively aid in implementing proactive control strategies to prevent the propagation of a breakdown to the other lanes by primitively balancing lane flow distribution among the other lanes. This solution cannot be achieved by predicting the average speed on all lanes.

Prediction results for multiple time steps ahead considering 50% CAVs are shown in Table 3.4. The accuracy of predictions decreases for farther time steps ahead as expected, due to the increased noise in observations, and the lack of detector measurements to help in tuning the noise coefficient. According to the results, total traffic flow for the first 1-minute time interval ahead is predicted with relatively high accuracies, with MAPEs of 3.0% to 3.8% for different penetration rates of CAVs. Further time steps ahead still show mean prediction errors below 10% and 15% for flow and speed parameters, respectively. The rolling horizon process will take care of continuously updating and reducing these longer interval predictions errors.

A sensitivity analysis is performed to investigate the impact of imperfect calibration of stochastic parameters of the microscopic model on the performance of the approach. The results reveal that, compared to the base simulation parameters, a variation of 10% in the probability of random acceleration, deceleration, and lane changing causes an increase of up to 1% in mean prediction errors for both per lane and overall flow parameters, as well as for the left lane and all lane speeds. In addition, an increase of up to 4% in mean errors is observed for right lane and middle lane speed predictions.

The performance of the travel time predictions decreased as prediction intervals increased from 1 minute to 2 minutes and where MAPE, for the total travel time prediction, increased from 8.6% to 13.3%; the penetration rate of CAVs remained unchanged at 50%. Similar to the overall flow and speed predictions, total travel time predictions showed a slight increase in MAPE as the penetration of CAVs decreased. Due to the small difference among the average errors, t-test analysis was conducted to compare the means of MAPEs for each pair of scenarios, which confirmed that the mean errors were not significantly different, except for the average travel times in 2-minute prediction intervals (Table 3.3).

Table 3.2. Error Indexes for 1-minute and 2-minute prediction intervals

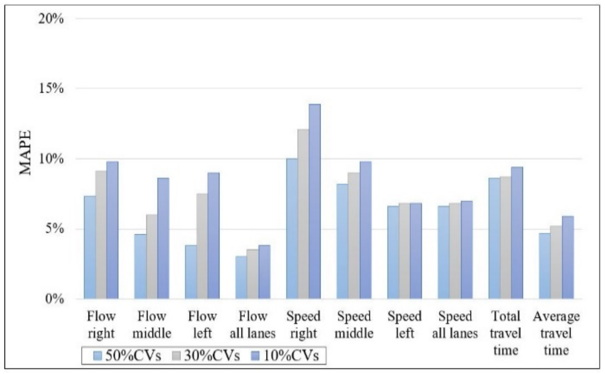
Error index	Traffic state parameter	1-min prediction interval			2-min prediction interval		
		50% CAVs	30% CAVs	10% CAVs	50% CAVs	30% CAVs	10% CAVs
MAPE	Flow-right lane	7.3%	9.1%	9.8%	8.1%	8.8%	11.2%
	Flow-middle lane	4.6%	6.0%	8.6%	8.0%	9.1%	11.6%
	Flow-left lane	3.8%	7.5%	9.0%	9.5%	9.9%	11.2%
	Flow-all lanes	3.2%	3.5%	3.8%	7.2%	8.2%	10.5%
	Speed-right lane	10.0%	12.1%	13.9%	14.1%	15.1%	16.0%
	Speed -middle lane	8.2%	9.0%	9.8%	10.6%	11.4%	13.0%
	Speed -left lane	6.6%	6.8%	6.8%	7.3%	7.9%	8.0%
	Speed -all lanes	6.6%	6.8%	7.0%	8.1%	8.6%	8.9%
	Total travel time	8.6%	8.7%	9.4%	13.3%	13.4%	13.7%
	Average travel time	4.3%	5.2%	5.9%	4.3%	4.6%	5.1%
RMSE (veh/h/lane)	Flow-right lane	99	115	125	101	121	144
	Flow-middle lane	80	103	110	152	169	170
	Flow-left lane	71	105	112	136	140	168
RMSE (veh/h)	Flow-all lanes	145	162	174	305	351	405
RMSE (km/h)	Speed -right lane	6	7	8	9	9	10
	Speed -middle lane	7	9	10	10	11	11
	Speed -left lane	7	7	8	8	9	9
	Speed -all lanes	5	5	6	7	7	8
RMSE (veh-sec)	Total travel time	2723	2815	2708	8159	8451	9082
RMSE (sec)	Average travel time	20	24	26	19	21	24

Table 3.3. t-statistics test results for means of MAPE errors in overall flow and speed predictions

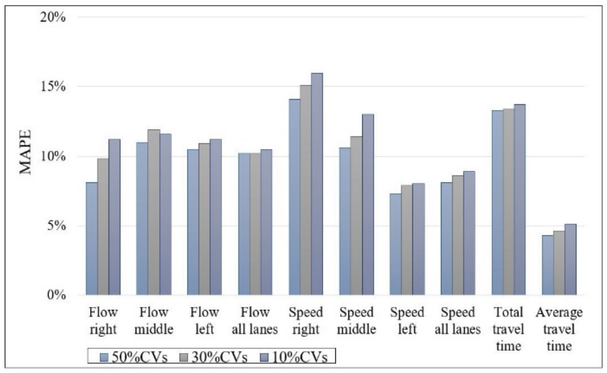
Parameter	Prediction intervals	Scenario Pairs	Alpha	N	t-Critical	t-stat	P-value
MAPE for all-lanes Flow predictions	1 minute	**50% CAVs-30% CAVs**	0.05	10	2.26	**-4.05**	**0.003**
		30% CAVs-10% CAVs				-1.21	0.25
	2 minutes	50% CAVs-30% CAVs				-0.14	0.89
		30% CAVs-10% CAVs				1.08	0.31
MAPE for all-lanes Speed predictions	1 minute	50% CAVs-30% CAVs				0.43	0.68
		30% CAVs-10% CAVs				-1.14	0.28
	2 minutes	50% CAVs-30% CAVs				-0.38	0.71
		30% CAVs-10% CAVs				-0.43	0.68
MAPE for Total Travel time	1 minute	50% CAVs-30% CAVs				-0.43	0.68
		30% CAVs-10% CAVs				-2.02	0.07
	2 minutes	50% CAVs-30% CAVs				-0.32	0.75
		30% CAVs-10% CAVs				-0.79	0.45
MAPE for Average Travel time	1 minute	50% CAVs-30% CAVs				-1.77	0.11
		30% CAVs-10% CAVs				-1.5	0.16
	2 minutes	**50% CAVs-30% CAVs**				**-4.5**	**0.002**
		30% CAVs-10% CAVs				**-2.6**	**0.03**

Table 3.4. MAPEs for multiple step ahead predictions with 1-minute prediction intervals

Traffic state parameter	1-min ahead	2-min ahead	3-min ahead	4-min ahead	5-min ahead
Flow-right lane	7.3%	9.5%	11%	12.2%	14%
Flow-middle lane	4.6%	6.5%	8.1%	9.5%	10.9%
Flow-left lane	3.8%	5.4%	7%	8.2%	9.1%
Flow-all lanes	3.2%	4.5%	6.1%	7.2%	9.2%
Speed-right lane	10.0%	14.2%	15.9%	17%	18.1%
Speed -middle lane	8.2%	12.1%	13.7%	14.5%	15.7%
Speed -left lane	6.6%	10.2%	11%	12.1%	12.9%
Speed -all lanes	6.6%	9%	10.1%	10.9%	13%
Total travel time	8.6%	11.9%	12.8%	13.5%	14.5%
Average travel time	4.3%	6.5%	7.8%	8.9%	10%

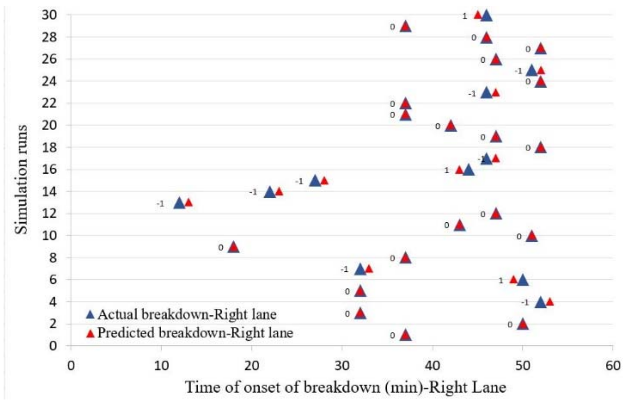


(a) 1-minute intervals

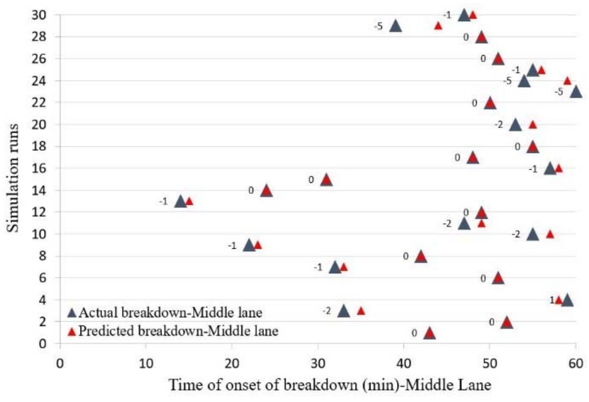


(b) 2-minute intervals

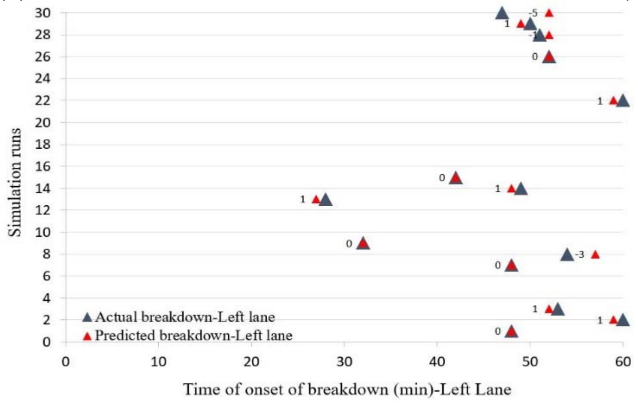
Figure 3.8. Mean Absolute Percentage Error for: (a) 1-minute intervals, (b) 2-minute intervals



(a)



(b)



(c)

Figure 3.9. Prediction of breakdown occurrence time for over 30 simulation runs; (a) Right lane, (b) Middle lane, (c) Left lane

3.3.3 Comparison to other methods

In order to further evaluate the performance of the proposed traffic state prediction model, two state of the art models are considered as benchmarks. Autoregressive integrated moving average (ARIMA) model has gradually become a standard method to compare with newly developed forecasting models because of its well-defined theoretical foundations and promising ability to predict traffic parameters (Lin et al., 2018). Thus, a seasonal ARIMA (SARIMA) model combined with a Kalman filter is chosen as the first benchmark model. On the other hand, traffic parameters display time-dependent volatilities that can not be explained by ARIMA models. Therefore, recent studies combined the generalized autoregressive conditional heteroskedasticity (GARCH) process with ARIMA to explain and predict such volatilities (Guo et al., 2014b; Y. Zhang et al., 2014). As a second benchmark, a SARIMA+GARCH structure is implemented and, an AKF is utilized to adaptively smooth the noise parameters. Interested readers can refer to Guo et al. (2014) for detailed explanation of this model. Traffic data from the same simulation runs and for the same aggregation and prediction intervals are used in this comparative analysis. Data sets including 1800 data points are used to calibrate the SARIMA (0,1,1) (0,1,1) and GARCH (1,1) structures for overall flow, average speed and average travel time predictions. The AKF process is similar to the proposed model while, the future observations of the state parameters are produced by the time series structures.

Figure 3.10 demonstrates a sample of the actual and predicted traffic flow, speed, and average travel time for location (B) for all examined models. Flow and average travel time prediction results show that both the proposed microsimulation-based model and SARIMA+GARCH+AKF model can accurately predict the evolution of these parameters over 1-minute prediction intervals compared to SARIMA+KF. In terms of the speed predictions, it can be observed that the proposed model can better capture the sudden changes in traffic speed, while the SARIMA+KF model has a tendency of underestimating the future speed and, SARIMA+GARCH+AKF shows rather large fluctuations around the actual speed values.

To further examine the prediction performance, the results for multiple step ahead predictions are summarized in Table 3.5. The MAPE results show that the proposed model consistently outperforms both benchmark models for all prediction steps, especially in the case of speed predictions. The accuracy of predictions reduces for farther time steps ahead for all methods. This can be attributed to the increased uncertainty associated with future traffic behavior. In terms of the flow predictions, the corresponding MAPEs results of the proposed model show only a slight improvement compared to those of the SARIMA+GARCH+AKF model with MAPEs of 3% to 10% for 1 to 5 prediction steps ahead for both models. The flow prediction results for SARIMA+KF shows a lower performance compared to the other models due to the lack of adaptability of the KF and also the lack of noise modeling properties of the GARCH model. The MAPEs for the average travel time predictions spread in a range of 4.3% to 10% for the proposed model, while ranging from 7.3% to 11.5% for the benchmark models which shows the advantage of the proposed model in this regard.

Speed prediction results demonstrate that the time series-based benchmark models are consistently inferior to the proposed model as the MAPEs are above 13% even for the first prediction step. While, the proposed model exhibits MAPEs of 6.6% to 13% for 1 to 5 prediction steps ahead. This higher performance can be attributed to the adaptability of the microscopic simulation model to the prevailing traffic conditions by capturing the CAVs data and simulating the drivers' behaviors accordingly. Moreover, the microscopic model considers stochastic driver behaviors such as lane changing, over acceleration, and speed adaption effects, which later form the macroscopic traffic parameters. These attributes enable the model to predict the speed fluctuations, especially the timely prediction of the onset of breakdown which is critical information for traffic control purposes. Using more advanced time series modeling and artificial intelligence approaches on larger data sets might lead to improve prediction performance. However, the training and computation burden of such models limits their applicability to the real-time traffic control.

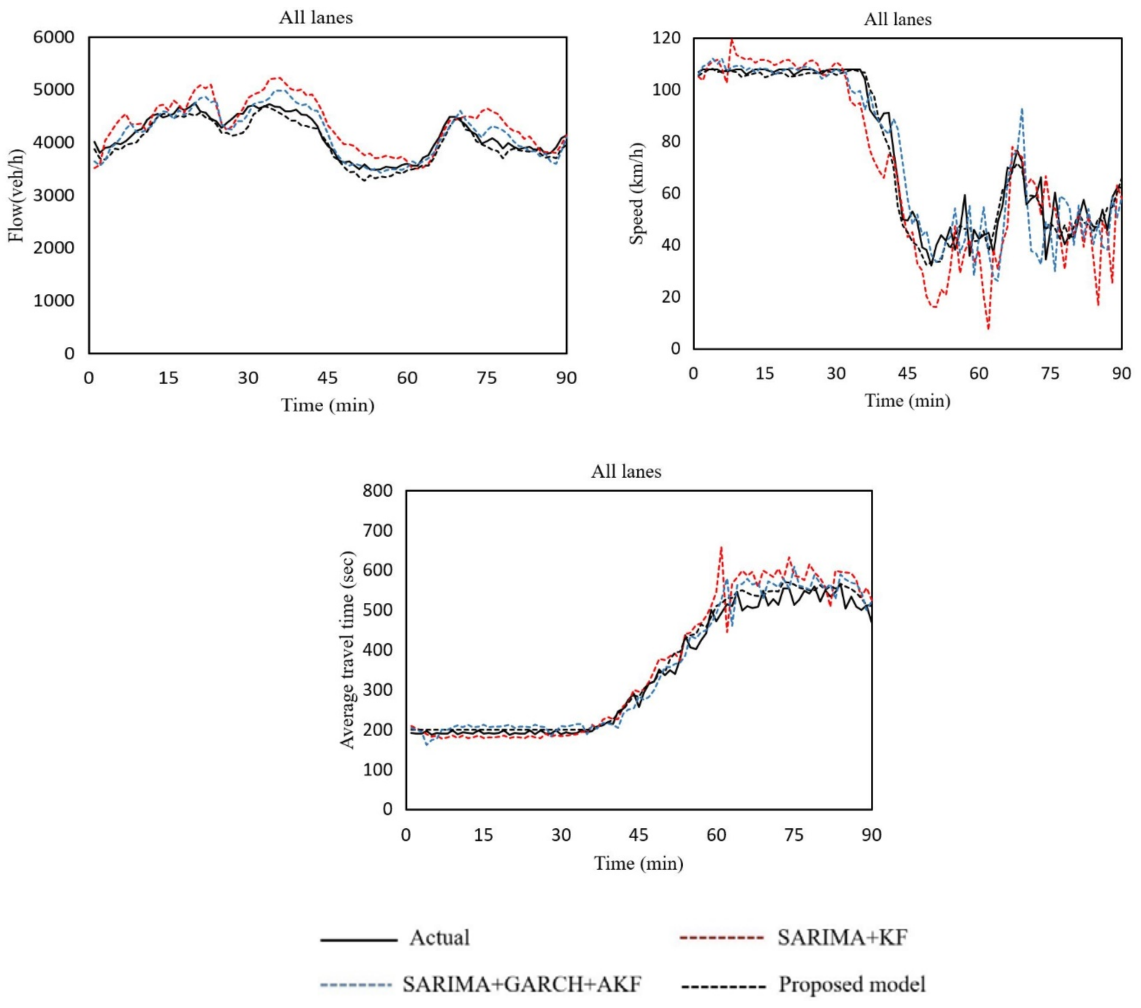


Figure 3.10. Traffic flow, speed, and average travel time prediction for 1-minute intervals (Location B)

Table 3.5. MAPEs for predictions based on the proposed method, SARIMA+KF method, and, SARIMA+GARCH+AKF (%)

	Proposed model					SARIMA+KF					SARIMA+GARCH+AKF				
Number of prediction steps ahead	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Flow	3.2	4.5	6.1	7.2	9.2	6.3	7.9	8.8	10.2	11.5	3.3	4.9	6.5	7.6	10.1
Speed	6.6	9.0	10.1	10.9	13.0	16.6	18.9	20.0	22.1	24.0	13.0	14.9	15.8	16.5	17.9
Average travel time	4.3	6.5	7.8	8.9	10	7.8	9.6	10.3	10.9	11.5	7.3	8.2	9.1	9.9	10.2

3.3.4 Spatial-temporal pattern prediction results

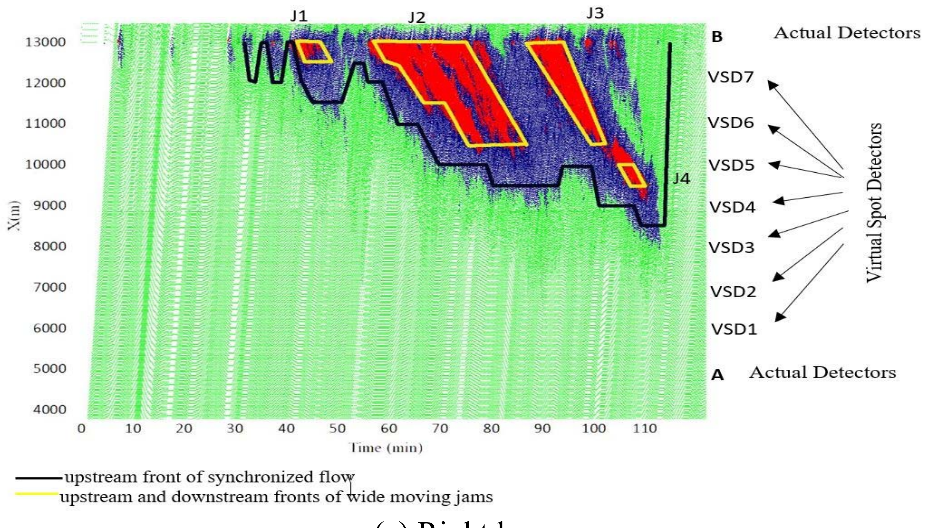
Module C of the developed model utilized the predicted traffic state information and FOTO and ASDA models to anticipate dynamically the formation and propagation of synchronized flow and wide moving jams in space and time, based on equations (3.30) to (3.35).

Figure 3.11 shows the time-space diagram with the expected traffic patterns for the right lane in space and time. Actual detectors are located at the fixed bottleneck locations, upstream and downstream of the link at x=5000 and x= 13000. Virtual spot detectors are assumed to be distanced 1000 m apart to collect average speed and traffic flow information. The background image in Figure 3.11 shows the actual vehicle trajectories on the link for the entire time period; the green trajectories correspond to vehicles with free flow speed, the blue ones reflect vehicles with a speed between 40 km/hr and 70 km/ hr, and the red ones represent vehicles with speeds below 40 km/ hr, which corresponds to an average wide jam speed (B. S. Kerner et al., 2005). Predicted jam fronts are shown with black and yellow lines for synchronized flow and wide moving jams, respectively. The model parameters used in the FOTO and ASDA models are v_{syn}=70 km/hr, v_{jam}=40 km/hr, $q_{jam} = 900\frac{veh}{hr}/lane$, and ρ_{max}=80 veh/km/lane.

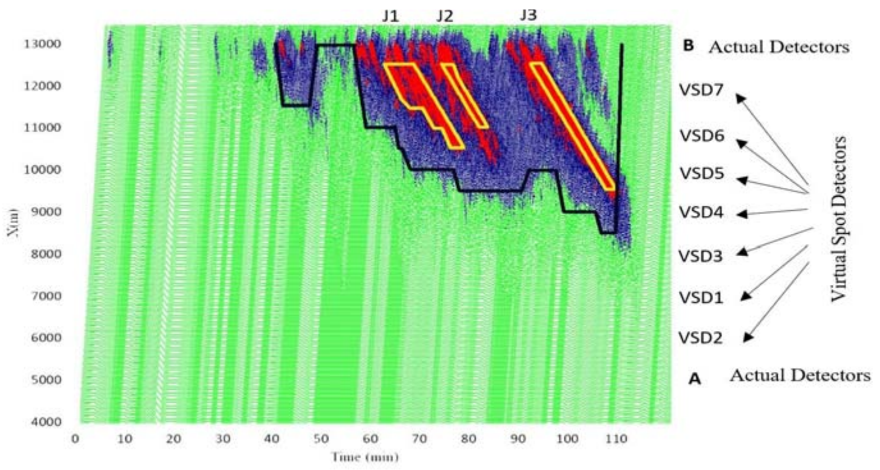
The black line in Figure 3.11 shows the upstream front of a synchronized flow (XSup), while the downstream front is fixed at the bottleneck location x=13300 m (the merging region of the on-ramp is from x=13000 m to x=13500 m). As illustrated in Figure 3.11, a few F to S to F transitions occur from t=5 min to t=32 min; these transitions are not detected by the tracking model because of a specific model setting that considers an F to S transition only if it lasts for at least 5 minutes. An F to S to J transition is detected at t=42 min forming the first wide moving jam (J1), which takes 8 minutes to dissipate (at t=50). While the second wide moving jam (J2) is predicted to occur at t=60 min, the real trajectory data indicates that it actually occurs at t = 59 min; consequently, there is a 1-minute delay in the prediction of J2. Based on the actual vehicle trajectory plot, lighter traffic conditions corresponding to the synchronized flow are observed from t= 75 min to t = 85 min, which results in local dissipation of the wide moving jam; however, the developed model is not able to reproduce these transitions. In fact, these synchronized flow patterns are observed at VSD6 and VSD7 (Figure 3.11(a)) for short periods of time, which causes the model to consider J2 as one pattern rather than three adjacent moving jams. The estimated speed for the downstream front of the wide moving jams is around 10 km/hr for all jams, which is consistent with the trajectory data.

Due to the complex spatial and temporal characteristics of traffic patterns, calculating an index to evaluate the accuracy of the predicted jam fronts is challenging. However, a visual examination of the predicted spatiotemporal patterns shows a good accordance compared to the trajectory data, especially the data that represents the occurrence and dissipation of synchronized flow and wide moving jams. In the case of J4 in Figure 3.11(a), the wide moving jam emerges between detectors VSD5 and VSD6; thus, the ASDA model is unable to track the S to J transition and the jam fronts until J4 reaches VSD5.

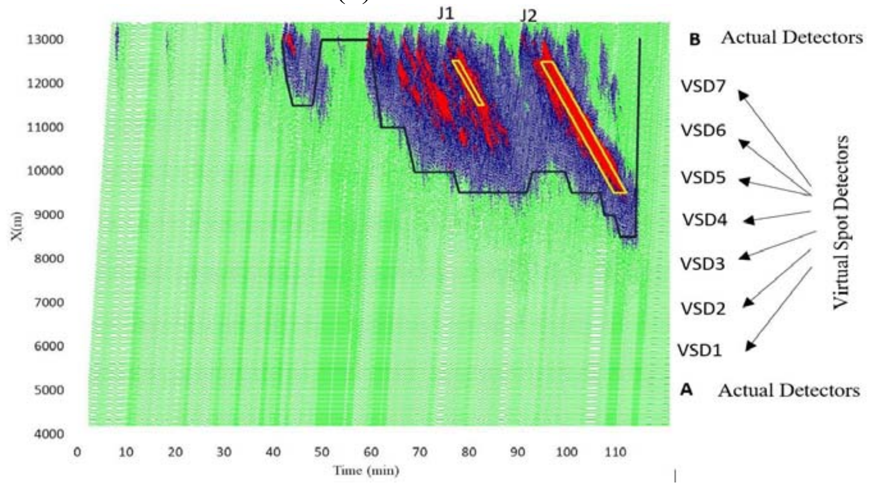
The developed model was tested to reproduce and predict traffic patterns in the middle and left lanes of the studied freeway segment. The upstream front of the synchronized flow was reproduced with satisfactory results in the middle and left lanes. However, the results were not as accurate for the wide moving jam predictions. As illustrated in Figure 3.11(b), the time and location of the wide moving jams were not efficiently predicted. This result may be explained by the fact that a breakdown in the right lane was triggered by the fixed bottleneck located at the merging region, where vehicles merged into the mainstream. However, a speed breakdown in other lanes was caused by stochastic drivers' behavior in over acceleration and lane changing and their attributed stochastic delays. These stochastic behaviors caused multiple stochastic transitions between traffic phases over short time periods. Another potential reason was that the onset of a breakdown in the middle and left lanes occurred upstream of the bottleneck detectors (usually between detector B and VSD7); therefore, the breakdown would not have been tracked by detector B.



(a) Right lane



(b) Middle lane



(c) Left lane

Figure 3.11. Predicted synchronized flow and wide moving jam patterns in space and time on the (a) right lane, (b) middle lane, and (c) left lane

3.3.5 Online simulation and computation time

A critical requirement for short-term and real-time traffic state prediction models is to produce timely predictions of the traffic state parameters while allowing for sufficient slack time to compute and inject proactive control measures. The proposed model is implemented using Go 1.14 programming language on a 64-bit Windows PC with 1.8 GHZ Inel Core-i7 and 16 GB of RAM (Appendix II). The online simulation and prediction processes take around 40 seconds for modeling a 2-hours traffic operation period on the network. This computation time is equivalent to around 0.67 second for 1-minute prediction interval, which shows the potential applicability of the model for real-time implementation. In a real-world application of online microscopic simulation models, (Y. Liu et al., 2006) reported a computation time of 120 seconds over a 2-hours period for online simulation and prediction of travel times on a network with 10 detectors.

3.4 Findings and discussions

There were several interesting facts revealed by the results of this study in the following areas:

Overall traffic state predictions: According to the results, overall traffic flow, average speed, and total travel time for 1-minute intervals and a 10% penetration rate of CAVs were predicted with MAPEs of 3.8%, 7.0%, and 5.9%, respectively. Most of the studies on short-term data-driven traffic parameters reported MAPEs of 6% to 12% for traffic flow, 4% to 15% for average speed, and 4% to 14% for travel time predictions (Comert et al., 2016; Fei et al., 2011; Vlahogianni et al., 2004). Most of the data-driven approaches in the literature adapted applicable prediction horizons of 1 to 25 minutes and reported increased errors with larger prediction horizons. Model-based approaches did not suggest an appropriate prediction interval duration, but most studies reported higher accuracy for longer periods of 30 to 120 minutes (Oh et al., 2015, 2018). The accuracy of different prediction models could not be simply compared due to several factors that contributed to the efficiency and accuracy of the models. However, in general, the prediction time horizon and level of accuracy for the model proposed in this research is comparable with data-based techniques; the proposed model has the advantages of model-based methods regarding adaptability and explanatory powers.

The accuracy of predictions for aggregated traffic state parameters, including overall traffic flow, overall speed, and total and average travel times, is not significantly affected by the penetration rate of CAVs within the range of 10% to 50%. Thus, with 10% of CAVs, satisfactory predictions can be achieved, with MAPEs below 10% for all aggregated parameters and 1-minute prediction intervals. At this rate and for 2-minute intervals, errors stay below 10.5%, 8.9%, and 13.7% for traffic flow, speed, and total travel time, respectively. These findings are in line with the results of Allström (2016), who achieved accuracies of 5% to 12% with 15% of probe vehicles. However, Seo & Kusakabe (2015) observed a 40% error in flow prediction when the penetration rate was as low as 3.5%. Thus, a minimum rate of CAVs should be available to achieve reliable predictions, even though, after a certain rate, the reliability may not necessarily improve by further increasing the penetration rate. Toppen & Wunderlich (2003) came to the same conclusion: at the highest levels of accuracy, little may be gained by making further improvements, and consequently, it makes little sense to invest in improvements if the gain in accuracy is minimal. In contrast, Tarnoff et al. (2008) concluded that if the error exceeds 20%, it undermines the effectiveness of the system and the public will lose confidence in the information source.

Lane by lane predictions: In this research, traffic state parameters are predicted on a lane by lane basis. This information is crucial to produce effective lane management strategies and driver assistant systems that promote a more balanced lane-flow distribution. Scenario analysis of different prediction intervals and penetration rates of CAVs shows a wider range of prediction errors on different lanes. In general, the prediction results for 1-minute intervals are superior compared to 2-minute intervals, and errors slightly increase by reducing the penetration rates, which is expected. Per lane flow predictions are shown to be reliable even with 10% of market penetration rates of CAVs. However, per lane speed prediction in the right lane, result in a MAPE of 10% to 16%. Although this is a limitation of the proposed model, it falls in the recommended error range in the literature, which is 5-20% (Allström, 2016). However, one of the important contributions of this research is that the onset time of the occurrence of a breakdown in the right lane, which is an important indicator used to activate control devices, can still be predicted with an error in the range of 1 minute even with 10% CAVs. Speed prediction errors for speed in the left lane are lower, with an error of around 6.6% and a slight decrease in MAPEs (0.2% decrease) by reducing the penetration rate from 50% to 30%. Thus, statistical tests were conducted to compare the mean MAPEs of all simulation runs for each scenario. The t-test results showed that changing the penetration rates does not significantly impact the mean MAPEs for these parameters. The same argument regarding an optimum reliability of predictions, which is achieved by a certain minimum penetration rate, may apply here.

While no analytical model-based traffic prediction methods have been developed for lane-level traffic state, the data-based lane-level fusion deep learning (FDL) speed prediction approach developed by (Gu et al., 2019) resulted in an average MAPE of 6.20%, 6.16% and 6.43% for the speed predictions at the right, middle and left lanes respectively. The prediction accuracy of FDL approach is shown to be similar to that obtained for our proposed method. Compared to the 90 minutes peak-period prediction span in our study, the MAPEs in the FDL approach are calculated over 24 hours of total prediction period which result in reduced MAPEs due to the higher accuracy during off-peak hours. A few recent studies revealed significant differences in operational characteristics of traffic lanes on freeways, and consequently, driving and lane changing behaviors should be incorporated to develop accurate multilane macroscopic traffic flow models (Duret et al., 2012; Nagalur Subraveti et al., 2019; Pan et al., 2019; Shiomi et al., 2015). These models should be further investigated so they can be used in model-based lane by lane traffic state predictions, especially when only stationary traffic data is available. In this present study, utilizing a combination of stationary and floating traffic data as input to the stochastic microscopic model and AKF reduces the extensive effort of calibrating multiple lane changing incentive parameters that are necessary requirements for macroscopic models.

Onset of breakdown: One of the main outcomes of this research is predicting the time of a breakdown occurrence on freeway lanes. As shown in Figure 3.9, the proposed model predicts the occurrence of a breakdown in the right lane a maximum of 1 minute later than the actual occurrence. The time difference between the actual and predicted onset of a breakdown is in the range of 0 to 5 minutes in the middle and left lanes because the stochastic lane changing behaviors are the main triggers of breakdowns in these lanes. Another promising finding is the apparent time difference between the onset of a breakdown when it is identified in the right lane and when the overall speed breakdown occurs. In other words, in most cases, breakdown in the right lane occurs at least 6 minutes before the overall breakdown. This result is in line with the results of previous studies that investigated lane-based breakdown and found breakdown occurred at different times in different lanes (Ma et al., 2013; Goto et al., 2018). Thus, utilizing average speed for all lanes as an indicator to activate control measures on freeway bottlenecks may cause inappropriate or insufficient control decisions. Future research should examine and compare the effectiveness of potential control strategies using both overall and right lane speed thresholds as indications of breakdown.

A further interesting finding is that in 16 out of 30 simulation runs, breakdown does not happen in the left lane at all. This further illuminates the need to develop lane management systems to optimize traffic flow efficiency (recently investigated by Yao et al. (2017). Although this is a partial limitation of the method in lane management applications, it does not affect future applications in on-ramp control, where breakdown in the right lane is the main concern.

Traffic pattern prediction: The spatial and temporal evolution of synchronized flow and wide moving jam patterns is predicted in this research based on the FOTO and ASDA models proposed by Kerner et al. (2005). However, some extensions are suggested. First, mutating the trajectory data to supplementary macroscopic traffic parameters at the VSD locations improves the accuracy of the tracked patterns with no need to add measurement infrastructure. Second, pattern tracking, and prediction is conducted on a lane by lane basis to count for different traffic behavior in the freeway lanes. The results revealed that the model may not be able to accurately reproduce congested patterns in the middle and left lanes. This result warrants further investigation to develop lane by lane congestion tracking approaches. Despite the discussed limitations, the detected and predicted congested patterns provide valuable information to activate freeway control and management measures. A novel application is to place a variable speed limit (VSL) sign upstream of the congested area to prevent jam propagation further upstream. This issue has not been investigated in VSL studies and applications. Another potential application is to produce accurate information about traffic conditions that can be used in driver assistance systems and variable message signs.

According to the above results and discussions the main advantages of the proposed traffic state prediction model can be summarized as follows: 1) given the short prediction interval, the accuracies of the overall traffic state parameters' predictions are similar or superior than other data-driven methods and yet without requiring intensive data; 2) the theoretical microscopic modeling approach combined with a data fusion model is shown to properly capture, reconstruct, and predict non-recurrent traffic conditions and spatiotemporal patterns on a lane by lane basis;

and 3) the model is able to predict the breakdown occurrence for the right lane at an early stage and with minimal time lag, which is critical to proactively inject proper remedy actions (e.g. RM or VSL) to prevent congestion propagation.

CHAPTER 4: A DYNAMIC BARGAINING GAME THEORY APPROACH TO RAMP METERING UNDER STOCHASTIC CAPACITY IN A CONNECTED VEHICLE ENVIRONMENT

4.1 Introduction

4.1.1 Background and motivations

Analysis of empirical traffic data shows that breakdowns in freeway bottlenecks do not occur at the same time for all lanes (D. Ma et al., 2013; Pan et al., 2019). For instance, next to a lane drop section, congestion often starts first in the right lane because of the lane changing activity of merging/diverging vehicles and then later propagates to the other lanes. However, modeling the stochastic breakdown behavior as a function of the traffic state in the right lane has been often overlooked in bottleneck capacity analysis, and more importantly, in ramp control approaches. Modeling the stochastic breakdown behavior and predicting the onset of a breakdown in the right lane of a merging bottleneck can be used as a proactive trigger in freeway controls (e.g., smart management of lateral flows) to take timely actions to prevent breakdown, thereby avoiding the propagation of breakdown to other lanes.

Despite the confirmed system-wide benefits of responsive and predictive ramp metering approaches, only a few studies in the literature raised concerns regarding the equity of such systems. Equity concerns may result in public and political opposition when it comes to implementing ramp metering strategies in the field (Benouar, 2004; Yin et al., 2004; Zhang & Levinson, 2005). Equity is a complex and broad concept that can be defined and measured in several ways. In general, equity is defined as a fair distribution of positive and negative outcomes from changing a policy or implementing a new policy (Cook & Messik, 1983). Current approaches mainly consider achieving a fair ramp metering solution in regard to waiting times on the metered ramps. However, fairness, in general, is difficult to define, and it depends on various factors such as policies, road user combinations, location of the network, and land use at the vicinity of the network. Thus, identifying and optimizing an inclusive equity index that accounts for such diverse definitions is a complex and perhaps an even impossible task, especially in dynamic and predictive ramp control settings. Tackling this issue using the concept of utility for road users and/or controllers might be helpful to develop a more adjustable and flexible control method that is able to consider and quantify available information and authorities' and users' perspectives of equity.

The proposed ramp metering approach in this study considers the need for flexibility in the solution to the ramp metering problem through a bargaining framework. Queue constraints are introduced in the majority of ramp metering models to prevent queue spillback and also to render the control approach more equitable in regard to waiting times in ramp queues. However, queue constraints decrease the effectiveness of the models: the ramp metering models fail to prevent or dissolve congested conditions completely (Papamichail et al., 2010). Thus, there are two critical parameters that need to be examined more closely: the frequency of the failure events that indicate the effectiveness of the capacity constraints and the distribution of the failure events among various ramps in a network, the latter being a critical equity concern. The number of breakdown events over a long period of operation has not been evaluated as an equity measure in previous studies.

In addition, despite decades of effort to address equity concerns of ramp metering implementations, developing flexible and computationally efficient control models that consider local equity while achieving system-wide efficiency is still in its infancy. I2I and V2I communication of future CAVs can be explored to address many of these challenges. I2I communications can play a unique role in enabling real-time controllers to work cooperatively to achieve the overall objective of improved freeway efficiency and equity while increasing safety and reliability. In addition, through their V2I capabilities, CAVs can disseminate vital information on lane-by-lane vehicle trajectories to road infrastructure (e.g., RM), leading to optimized traffic management schemes. Using CAV information in stochastic microscopic models to develop ramp metering approaches needs to be further investigated. This chapter takes many steps in this direction.

This research develops a stochastic model predictive control approach that optimizes ramp metering rates while accounting for the stochastic system dynamics and stochastic bottleneck capacity. To rectify the problem of a long computation time in the model predictive control (MPC) approach for real-time applications, a bargaining game theory approach has been adopted that allows for dynamic communication among controllers. Thus, the freeway control strategy developed in this study is formulated as a stochastic distributed model predictive control (SDMPC) framework based on a bargaining game approach to process the available information. This configuration allows the controllers to communicate their state and decision information while reducing the computational burden. Consequently, this configuration is a compromise between local and global performance (Portilla et al., 2012; Valencia, 2012; Valencia et al., 2015).

4.1.2 Review of the previous studies

The literature review section in this chapter is explained in three categories according to the main issues discussed in the studies (Figure 4.1).

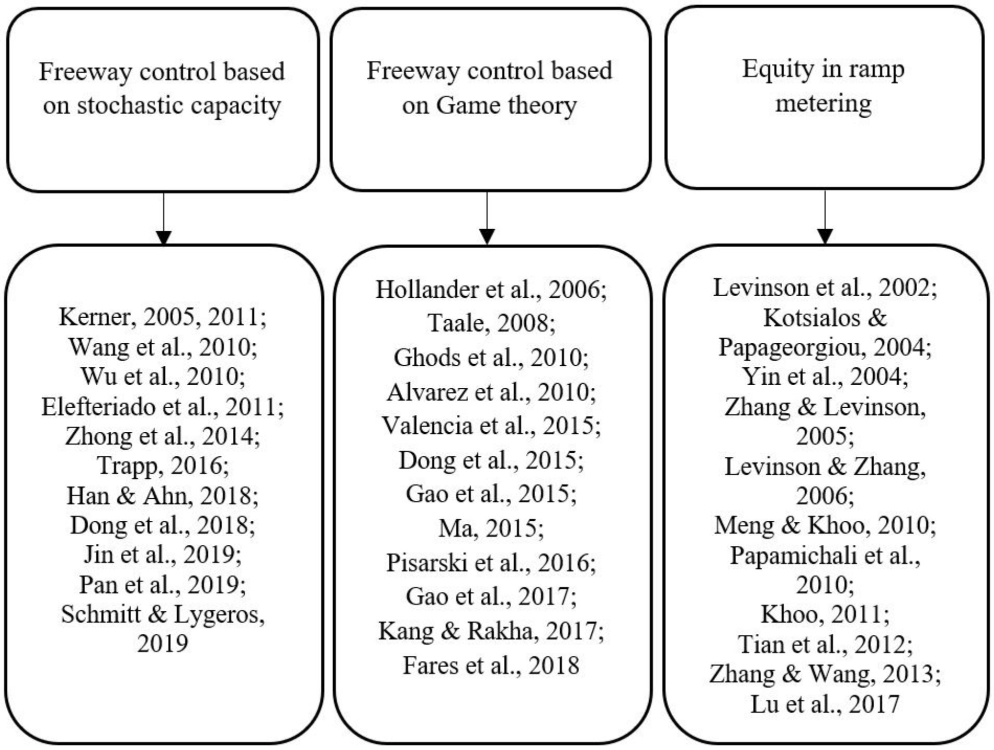

Freeway control based on stochastic capacity
Freeway control based on Game theory
Equity in ramp metering
Kerner, 2005, 2011; Wang et al., 2010; Wu et al., 2010; Elefteriado et al., 2011; Zhong et al., 2014; Trapp, 2016; Han & Ahn, 2018; Dong et al., 2018; Jin et al., 2019; Pan et al., 2019; Schmitt & Lygeros, 2019
Hollander et al., 2006; Taale, 2008; Ghods et al., 2010; Alvarez et al., 2010; Valencia et al., 2015; Dong et al., 2015; Gao et al., 2015; Ma, 2015; Pisarski et al., 2016; Gao et al., 2017; Kang & Rakha, 2017; Fares et al., 2018
Levinson et al., 2002; Kotsialos & Papageorgiou, 2004; Yin et al., 2004; Zhang & Levinson, 2005; Levinson & Zhang, 2006; Meng & Khoo, 2010; Papamichali et al., 2010; Khoo, 2011; Tian et al., 2012; Zhang & Wang, 2013; Lu et al., 2017

Figure 4.1. Literature review classification

A) Freeway control based on stochastic capacity

In recent years, there has been a growing number of appeals to model probabilistic breakdowns on freeway bottlenecks, and more importantly, to develop freeway control measures under such uncertainties (Dong et al., 2018; Han & Ahn, 2018; Pan et al., 2019; Schmitt & Lygeros, 2020; H. Wang et al., 2010; Zhong et al., 2014). Wu et al. (2010) used a pre-determined risk level based on the cumulative probability distribution of a breakdown to find the operational capacity. They also applied a chance-constrained zone algorithm approach to ramp metering with the objective of optimizing freeway throughput. In another study, Elefteriadou et al. (2011) used a product limit method to model the numerical values of the probability of a breakdown, and ramp metering rates were determined based on a 15% to 20% probability of a breakdown at each bottleneck. In a local ramp metering strategy, the metered ramp flows were calculated considering short-term and mid-term mainline flows. The developed algorithm required data measurements and control actuation on a second by second basis (Trapp, 2016). The above stochastic-capacity-based control approaches significantly improved the throughput of freeways, delayed breakdowns, and reduced average travel time and congestion duration.

Kerner (2002) proposed a three-phase theory, and according to the empirical findings, Kerner identified two different phases, in addition to free flow, of synchronized flow and wide moving jams with different microscopic and macroscopic characteristics. Transitions between the traffic phases are stochastic phenomena that are caused by microscopic driver behaviors such as over-acceleration and speed adoption; therefore, the stochastic transitions result in stochastic capacity on freeway bottlenecks. Based on this concept, Kerner (2007) developed a local congested traffic control approach (ANCONA) in which synchronized flow patterns were allowed to occur at the merging bottlenecks. Thereafter, the ramp metering control was triggered to localize the congested pattern in the vicinity of the bottleneck and to prevent its propagation. Despite the difficulty of applying ANCONA in a coordinated way at the network level, comparing its outcomes to free flow control approaches showed higher throughputs on the main road and on-ramps, and lower vehicle waiting times on the on-ramps (Kerner, 2007).

At the network level, a breakdown minimization (BM) principle was introduced to assign link flow rates in a way that the probability of a breakdown occurrence in at least one of the bottlenecks is minimized. In other words, BM is equivalent to maximizing the probability that a breakdown occurs at none of the bottlenecks. This approach to traffic assignment showed considerably greater inflow rates compared to user equilibrium and system optimized approaches; however, a spatial control measure must be implemented along with BM to localize congestion if a breakdown has occurred (Kerner, 2011).

B) Freeway control based on Game theory

Game theory is a mathematical optimization framework that includes a number of decision-makers (players) with conflicting interests. The game theoretical approaches allow for modeling multiple players that can be adversaries with various concerns regarding performance or reliability, which may or may not be consistent with other players or with the system-wide performance. In regard to transportation networks, various groups of users, authorities, agencies, controllers, etc. can be considered as players with contradictory utilities. Thus, the application of game theory in traffic modeling and control has been expanded over recent decades (Alvarez & Poznyak, 2010; Hollander & Prashker, 2006; Kang & Rakha, 2017; Pisarski & Canudas-De-Wit, 2016; Taale, 2009; Talebpour et al., 2015).

Chen & Ben-Akiva (1998) first explored the integration of dynamic traffic control and dynamic traffic assignment using three non-cooperative games, and the results showed the superiority of Stackelberg equilibrium compared to Cournot and Monopoly games (Chen & Ben-Akiva, 1998). The Stackelberg game, in which one or more players can anticipate the reactions of other players, was later used to develop integrated anticipatory road network controls (Taale, 2008). Integrated and coordinated control problems include several variables that need to be considered in the formulation and optimization process. Game theory models provide a computationally efficient framework to solve the dynamic control problems in real time (Ghods et al., 2010; Hollander & Prashker, 2006). However, most traffic control strategies in the framework of game theory are formulated based on non-cooperative games in which the players (i.e., controllers) do not have information about the traffic conditions, actions of other players, and respective outcomes (Alvarez & Poznyak, 2010; Ghods et al., 2010). In real traffic networks, this information can be available and transferable among controllers through traffic control and management infrastructure. The transfer of information is expected to become seamlessly sharable in the near future.

C) Equity in ramp metering

In ramp metering, equity is usually considered as a fair distribution of the reduction in travel time by ramp metering among on-ramp and freeway users. For instance, local ramp metering independently applied to multiple ramps may result in very long queues on some ramps compared to others, which is unfair (Papamichail et al., 2010). Another empirical study by Levinson et al. (2002) reported that ramp meters were more beneficial for long trips compared to short trips on Route 169 in the Twin Cities, Minnesota. This study showed that by implementing ramp metering, network users who travel three exits or fewer experience less reduction in travel time, while longer distance travelers benefit more, which caused public resistance to ramp metering and raised substantial doubts on the overall system effectiveness for users who were experiencing long delays on the ramp (Levinson et al., 2002; Levinson & Zhang, 2006; Lei Zhang & Levinson, 2005). Thus, it is important to develop ramp metering methods that are able to integrate both system-wide and local benefits in determining a ramp metering rate solution.

A few practical measures in the literature that considered equity factors or RM include a lower threshold for ramp metering rates, maximum allowable waiting times, ramp queue overrides, and weighted penalties for queue constraints. These practical solutions are shown to provide implicitly more balanced solutions to the ramp metering problem in terms of equity considerations (Kotsialos & Papageorgiou, 2004; G. Zhang & Wang, 2013). More explicit approaches considered weighted travel time optimization, incorporated dynamic penalties, or quantified and optimized spatial and temporal equity indexes to achieve some degree of fairness. Most of these studies showed measurable success in decreasing inequalities in ramp metering solutions, while the results were highly sensitive to the correct tuning of weight and penalty coefficients or required complex grouping and heuristic approaches to solve the ramp metering problem (Khoo, 2011; Meng & Ling, 2010; Q. Tian et al., 2012; Lei Zhang & Levinson, 2005).

4.2 Dynamic and cooperative ramp metering approach

In this study, the problem of cooperative ramp control is formulated to minimize system-wide travel time while considering the stochasticity of local bottleneck capacity. The freeway network is assumed to be continuously monitored and, thus, receives real time data. In addition to aggregate speed, density and flow data from point detectors and microscopic trajectory data are assumed to be available via vehicle to infrastructure communication (V2I). Thus, connected vehicles that are equipped with advanced sensing and communication capabilities will enable constant monitoring and exchange of their individual speed and position information with other roadside units (RSU) via 5G or DCRC.

The proposed method is formulated in a model predictive framework. Model predictive control (MPC), also known as receding-horizon control, is an online control approach that uses the dynamic information provided by infrastructure to optimize proactively a predefined cost function while considering the operational (e.g., maximum acceptable waiting time) and physical constraints of the system (e.g., queue storage capacity of a ramp).

MPC-based freeway control approaches are mainly focused on minimizing the total travel time on a freeway segment by controlling one or more control variables such as ramp metering rates and variable speed limits. Each MPC problem includes three main components: 1) a state-space model that describes and predicts the dynamic evolution of the system as a function of the current state of the system and control actions; 2) a system performance measure, commonly represented by a quadratic cost function; and 3) a set of constraints determined by the operational and physical limitations of the system.

A general centralized MPC problem can be formulated as follows:

$$\min_{u(k)} \sum_{h=k}^{k+N_p-1}[x^T(h+1|k)Wx(h+1|k)] + \sum_{h=k}^{k+N_c-1}[u^T\,(h)Vu(h)]$$

s.t:

$$x(h+1) = f(x(h), u(h)) \tag{4.1}$$

$$x(h+1) \in X\,, u(h) \in U$$

where, N_c and N_p are control and prediction horizons (with $N_c \leq N_p$). To limit the computational complexity, a control horizon (Nc<Np) is applied, after which the control variable in the MPC does not change. $x(h)$ denotes the vector for the state of the system at time step (h), and $u(h)$ shows the control input vector. Diagonal matrices, W and V with positive diagonal elements, are penalty matrices for the rate of change in the state and input parameters, which are responsible for preventing abrupt changes in those parameters. $f(x(h), u(h))$ is a function describing the time evolution of the dynamic system. The centralized MPC problem in (4.1) needs to be solved in real time to produce proactive control measures, which makes it impractical for real-life, large-scale networks, especially when multiple parameters are being optimized. Distributed model predictive control (DMPC) is an effective way to overcome the issue of computational burden in MPC. Thus, the ramp metering problem in this research is formulated as a stochastic distributed model predictive control (SDMPC).

As shown in Figure 4.2, once real-time data is collected and entered into the SDMPC, the model runs two modules: 1) a traffic state estimation/prediction model so traffic disturbances and bottleneck formation are anticipated for a future horizon (Np) before they even occur and 2) a game theory-based RM solution algorithm where RM actions are determined through a bargaining game, and remedial and proactive RM control strategies are injected into the system in a rolling horizon fashion. In this rolling horizon scheme, only the first optimized RM values are implemented. The horizon is then shifted one sample time with new information becoming available from the system and fed back to the optimization function. The control time step used in this study is 1 minute, meaning that the RM system is able to adjust its rates every minute if required. Thus, the whole process is repeated continuously until the end of the simulation.

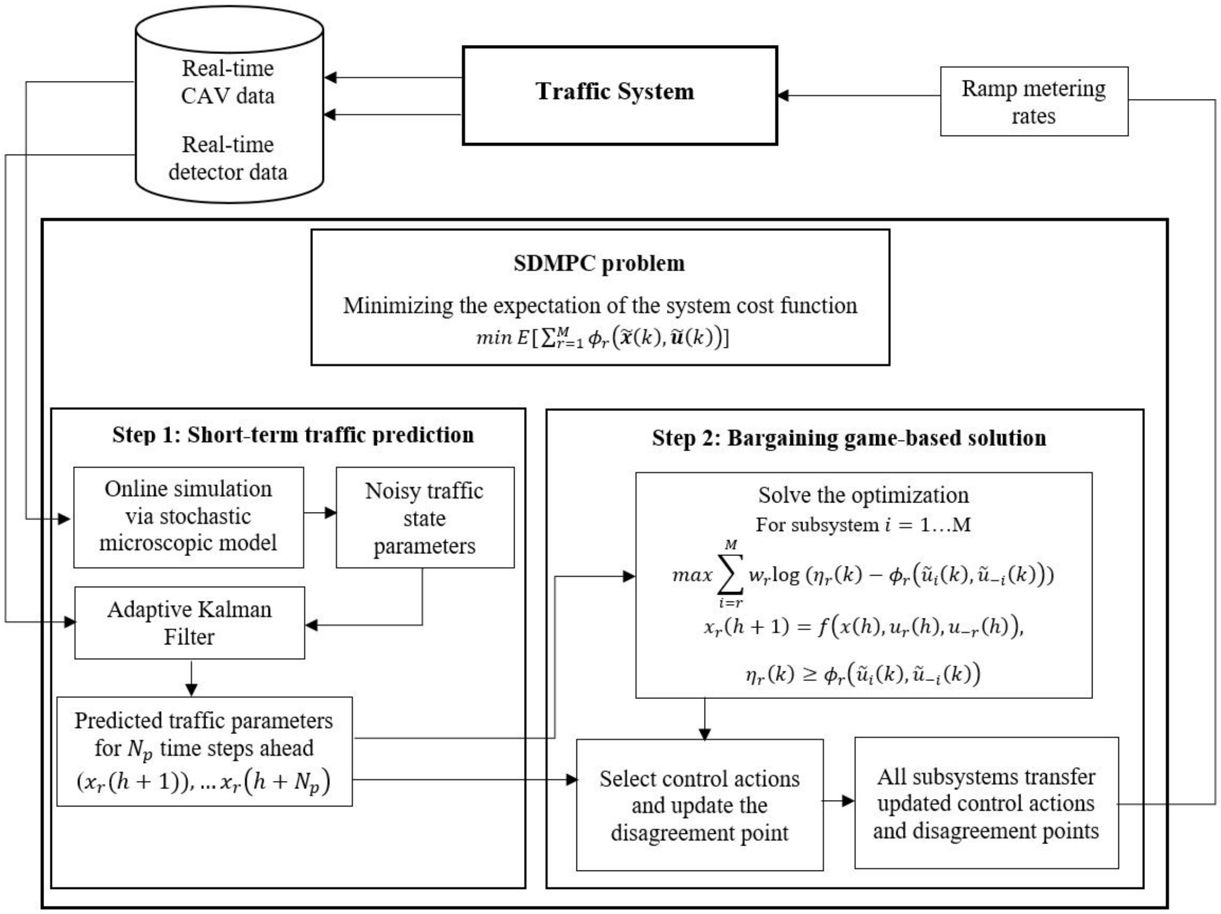

Real-time CAV data
Real-time detector data
Traffic System
Ramp metering rates
SDMPC problem
Minimizing the expectation of the system cost function
$min\, E[\sum_{r=1}^{M} \phi_r(\tilde{x}(k), \tilde{u}(k))]$
Step 1: Short-term traffic prediction
Online simulation via stochastic microscopic model
Noisy traffic state parameters
Adaptive Kalman Filter
Predicted traffic parameters for N_p time steps ahead $(x_r(h+1)), \dots x_r(h+N_p)$
Step 2: Bargaining game-based solution
Solve the optimization
For subsystem $i = 1 \dots M$
$max \sum_{i=r}^{M} w_r \log(\eta_r(k) - \phi_r(\tilde{u}_i(k), \tilde{u}_{-i}(k)))$
$x_r(h+1) = f(x(h), u_r(h), u_{-r}(h)),$
$\eta_r(k) \geq \phi_r(\tilde{u}_i(k), \tilde{u}_{-i}(k))$
Select control actions and update the disagreement point
All subsystems transfer updated control actions and disagreement points

Figure 4.2. Schematic diagram of the proposed SDMPC model and bargaining game solution to solve the predictive ramp metering problem

In SDMPC, the system is divided into several subsystems that communicate and share information to locally solve the MPC control actions, while trying to achieve some degree of coordination (Negenborn et al., 2008). As shown in Figure 4.3, the freeway network in this study is decomposed into M segments where each segment includes a local ramp controller.

The state-space model for each segment is formulated as a subsystem with a discrete-time stochastic linear state-space model:

$$x_r(k+1) = Ax_r(k) + Bu_r(k) + w_r(k) \qquad \text{for } r = 1, 2, \dots M \qquad (4.2a)$$

$$y_r(k) = C_k x_r(k) + v_k(k) \qquad \text{for } r = 1, 2, \dots M \qquad (4.2b)$$

For each subsystem, the future state of the system is a function of the current local state $(x_r(k))$ and the current local control input $(u_r(k))$. In the ramp metering context, $(x_r(k))$ is the vector of traffic parameters including flow, speed, and average travel time for the subsystem (r) at time step (k), and $(u_r(k))$ is the vector of ramp metering rates produced by the respective controller. The exact measurements of the state parameters are not available, but they can be estimated using the measurement equation $y_r(k)$, and A, B, and C are the state-space system matrices. The measurement noises are represented by $v_r(k)$, and the system disturbances, $w_r(k)$, are assumed to capture the combined impact of model uncertainty and exogenous disturbances (e.g., fluctuation in demand or collisions) on the state evolutions. $w_r(k)$ and $v_r(k)$ are assumed to be Gaussian white noises with covariances of Q_r and R_r. This assumption was tested using a Kolmogorov–Smirnov test and based on state measurements of various simulation runs as detailed Chapter 3.

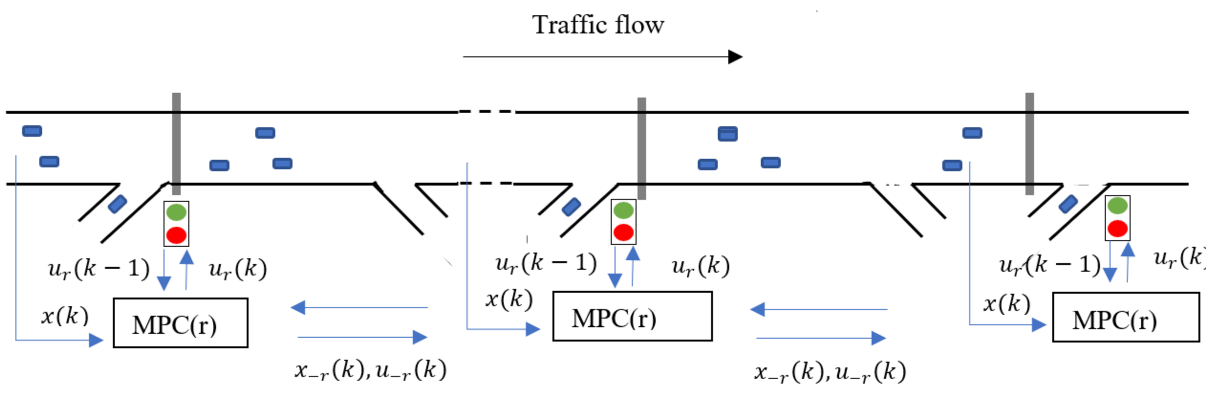


Figure 4.3. Schematic diagram of DMPC for a freeway segment with the ability to share information among controllers

The objective function of the decomposed system (i.e., each ramp meter in the case of this research) can be stated as equation (4.3) according to (Portilla et al., 2012; Valencia Arroyave, 2012; Venkat et al., 2005). The SDMPC problem in (4.3) can be solved by minimizing the expected cost of the decomposed system according to Heirung et al. (2018).

$$\min E[\sum_{r=1}^{M}\sum_{h=k}^{k+N_p-1} x_r^T(h+1|k)W_r x_r(h+1|k) + \sum_{h=k}^{k+N_c-1} u_r^T(h)V_r u_r(h)]$$

s.t:

$$x_r(h+1) = f(x(h), u_r(h), u_{-r}(h)) \tag{4.3}$$

$$E[x_r(h+1)] \in X_r \, , u_r(h) \in U_r$$

The term $\sum_{h=k}^{k+N_p-1} x_r^T(h+1|k)W_r x_r(h+1|k) + \sum_{h=k}^{k+N_c-1} u_r^T(h)V_r u_r(h)$ indicates the local cost function and is referred to as $\phi_r(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k))$ for simplification in the rest of this chapter. $x_r(h)$ and $u_r(h)$ are the local state and control input vectors; X_r and U_r are the local feasible sets for $x_r(h)$ and $u_r(h)$, respectively, with $X = \prod_r^M X_r$ and $U = \prod_r^M U_r$, which are the Cartesian products of the local feasible sets as expressed in Vanket et al. (2006). The control information as disseminated by other controllers is denoted by $u_{-r}(h)$. The state constraint is modified to reflect the fact that the state is a stochastic variable. According to Heirung et al. (2018), (Zhou et al., 2017), the SDMPC problem in (4.3) can be solved for the expected value of the predicted state instead of the actual value if the system disturbances are assumed to be Gaussian white noise. Thus, the optimal control that minimizes the expected cost can be converted to a deterministic optimization problem that treats the state estimate as its actual value, while an AKF that is developed to correct the noisy measurements produces the a priori estimate of the state. This research uses an extended version of the model based on a stochastic microscopic three-phase model, explained in Chapter 3, to estimate and predict traffic flow, speed, and travel time for short prediction horizons and for multiple time steps ahead. A brief mathematical formulation of the model is presented here.

4.2.1 Short-term microscopic-based traffic prediction model

The traffic prediction model in this research uses the Kerner-Kelnov (KK) stochastic microscopic model to simulate traffic conditions on a freeway segment in real time. KK is a stochastic microscopic three-phase theory-based traffic model that represents stochastic driver behaviors and their impacts on traffic behavior, especially in the context of freeway bottlenecks (Boris S. Kerner & Klenov, 2003). The details of the KK microscopic model are explained in Chapter 3. The following summary is provided for the reader’s convenience. Briefly, vehicles’ speed and location information are expected to be obtained from CAVs and other probe vehicles, and traffic state parameters, including flow, average speed, and link travel times, are estimated on a lane by lane basis as noisy measurements. Stochastic driver behaviors and random deceleration and acceleration decisions and their impacts on vehicular motions are considered in this model. Whenever a CAV’s simulated coordinate is past the nearest downstream detector location, the vehicle is counted, and its speed is recorded. The number of counted vehicles is corrected based on the average penetration rate of CAVs in the previous time intervals and is recorded as a noisy measurement of traffic flow. The average travel time from a previous upstream detector for CAVs is also recorded for each time interval.

An AKF is used to combine the received data from the online simulator with detector measurements. When detector measurements become available, estimations are updated, and the traffic state is predicted for multiple time intervals ahead. The AKF in this study is based on a series of sequential estimators that simultaneously calculate suboptimal adaptive estimations of the unknown a priori state and observation noise statistics of the system state based on N past observations. The details of The AKF predictor are explained in Chapter 3.

Choosing the number N of past observations is based on identifying the time frame in which the structural changes in traffic state parameters are considered. Review of the literature shows a range of sample sizes from N=4 (Ojeda, Kibangou, & de Wit, 2013) to N=20 (Zhou et al., 2017) of past observations. Considering 10-second observation intervals and after testing sample sizes increasing by 10 from N=10 to N=60 and calculating the corresponding minimum mean prediction errors, N = 20 and N=30 were selected for the online noise statistics estimations for traffic flow and speed parameters, respectively.

4.2.2 Bargaining game solution to the SDMPC problem with stochastic capacity

In the SDMPC optimization problem in (4.3), each controller finds its local optimized action ($\tilde{\mathbf{u}}_r(k)$) based on its local cost function $\phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)$ and local constraints, while also considering the decisions of other controllers. This situation can be formulated as a multi-player game with a set of “rules” defined by the local players and global system model; it also considers both operational and physical constraints. The “players” of the game are the controllers, and their desired “strategy” is to minimize the local system-wide control cost function; thus, their “desired choices” are the local control actions.

According to (Nash, 1953; Valencia Arroyave, 2012; Valencia et al., 2015), the bargaining and negotiation process of the game needs to be defined by the disagreement point ($\eta_r(k)$), which is defined as the added benefit (i.e., less decrease in local performance) perceived by a player if the player decides not to cooperate with the other players; consequently, the player acts solely based on their own local benefits. In other words, the disagreement point can be defined as the cost or disutility associated with a local alternative action when agreement is not a plausible alternative. For instance, in a ramp metering context, the coordinated ramp metering rate that is calculated by the SDMPC with the objective of optimizing the total system performance could result in a higher link travel time or breakdown occurrence in the vicinity of the local controller compared to what is expected by the localized control. In this case, the controller may decide not to cooperate and choose a ramp metering rate that better satisfies its local requirements.

In this framework, the utility of the controller r at each time step is defined based on the difference between the expected maximum cost of a non-cooperative decision (i.e., disagreement point) $\eta_r(k)$) and that of the local cost function in the event of a cooperative decision $\phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)$. Thus, each controller tries to find a sequence of actions ($\tilde{\mathbf{u}}_r(k)$) that maximizes its utility (($\eta_r(k)$)- $\phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)$), which is equivalent to a set of actions that minimizes $\phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)$ where, $\eta_r(k) \geq \phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)$.

The solution to the optimization problem in (4.3) is formulated based on maximizing the Nash products of the utility functions $\prod_{r=1}^{M}(\eta_r(k) - \phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right))$ that can be transformed into (4.4) according to Valencia (2012) because $\log \max \prod g(k) = \max \log \prod g(k) = max \sum \log g(k)$ for any convex function $g(k)$.

$$max \sum_{r=1}^{M} W_r \log\left(\eta_r(k) - \phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)\right)$$

s.t:

$$x_r(h+1) = f(x(h), u_r(h), u_{-r}(h)) \tag{4.4}$$

$$\eta_r(k) \geq \phi_r\left(x(k), u(k)\right)$$

$$x_r(h) \in X_r\,, u_r(h) \in U_r$$

where W_r is the weight factor for each controller and $\sum W_r = 1$. The weight can be set to provide more priority for a given RM. The function $f\left(x(h), u_r(h), u_{-r}(h)\right)$ is the prediction model described in section 4.2.1 and Chapter 3, and it is a function of the current state, inputs from the current controller, and inputs from other controllers. The solution of the optimization problem in (4.4) is unique and Pareto optimal where no change could improve the satisfaction of one player without other players being negatively affected. If the solution to (4.4) is feasible, the first time step of the control sequence is applied to the system; otherwise, the non-cooperative alternative is chosen as the control action that is inserted into the freeway system. In both cases, the disagreement point is dynamically evolved for the next time step as explained in the following section.

According to the above framework, the freeway control system in this research is divided into M distributed parallel subsystems where each subsystem includes an on-ramp and its upstream link. Controllers are modeled to minimize the total expected system delays, and the local cost function is defined by equation (4.5).

$$\phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right) = \sum_{h=k}^{k+N_p-1} T_r(h) + T_s L_r(h) + V_r(d_r((h) - d_r(h-1))^2 \tag{4.5}$$

Where $T_r(h)$ is the average link travel time predicted by the model explained in section 4.2.1 and Chapter 3. T_s is the prediction step, and $L_r(h)$ is the expected queue length on the ramp obtained using the simple queuing model $L_r(h) = L_r(h-1) + T_s(D_r(h) - d_r(h))$ where $D_r(h)$ is the ramp arrival inflow, and $d_r(h)$ is the metered flow that is allowed to merge onto the freeway (i.e., ramp metering rate). V_r is a nonnegative weighting factor that enables the control strategy to penalize undesirable abrupt changes in the selected ramp metering rates in two consecutive time periods. Thus, the bargaining game for the cooperative ramp metering problem is as follows:

$$max \sum_{r=1}^{M} W_r \log\left(\eta_r(k) - \phi_r\left(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\right)\right)$$

s.t:

$$x_r(h+1) = f(T_r(h), F_r(h), d_r(h), d_{-r}(h))$$

$$\eta_r(k) \geq \phi_r\left(x(k), u(k)\right) \tag{4.6}$$

$$L_{min-r} \leq L_r(h) \leq L_{max-r}$$

$$d_{min-r} \leq d_r(h) \leq d_{max-r}$$

Where L_{min-r} and L_{max-r} are the minimum and maximum queue length on ramp r respectively, and d_{min-r} and d_{max-r} indicate the range of the metered flow.

4.2.3 Formulating the disagreement point as a function of the local probability of a breakdown occurrence

The disagreement point needs to incentivize cooperative behavior and enhance the local performance of all players. These features can be achieved by dynamically updating the disagreement point at each time step of the optimization process as shown in (4.7) (Nash, 1953; Valencia Arroyave, 2012; Valencia et al., 2015). Thus, if the solution to (4.6) is feasible (e.g., a cooperative decision has been made), the disagreement cost can be reduced to improve the performance of the controller at the next time step by increasing the expectation of cooperative behavior. Otherwise, the current local control action overrides the solution in (4.6), and the disagreement cost increases according to (4.7) to encourage cooperative behavior by expanding the solution space to improve the chance of finding a cooperative set of actions in future time steps. The coefficient (α_d) can be either fixed or dynamically changed according to a user's preference of the range of change in local performance and expectations.

$$\eta_r(k+1) = \begin{cases} \eta_r(k) - \alpha_d\left(\eta_r(k) - \phi_r\big(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\big)\right) & if\,\eta_r(k) \geq \phi_r\big(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\big) \\ \left(\phi_r\big(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\big)\right) & if\,\eta_r(k) < \phi_r\big(\tilde{\mathbf{x}}(k), \tilde{\mathbf{u}}(k)\big) \end{cases} \tag{4.7}$$

According to (Chun & Thomson, 1990), the bargaining game problem with uncertain disagreement points can be solved by replacing the uncertain disagreement point with the expected value that represents the loss of local performance, which is evaluated at the current time for an unknown future condition. For the ramp metering problem in this research, the disagreement point that is the controllers' expected maximum loss of performance is defined based on the expected probability of a breakdown occurrence; this definition considers the stochastic capacity of freeway bottlenecks. Further, each controller finds a ramp metering rate that maintains the local probability of a breakdown within a determined range considering the predicted traffic states and local physical constraints.

In this research, the dynamic evolution of the disagreement point is projected by increasing or decreasing the accepted risk of a breakdown within the determined threshold. In other words, if the controller decides to cooperate, the controller tries to improve its local performance in the next time step by decreasing its expected level of risk of a breakdown, which is equivalent to lowering the local probability of a breakdown. According to the stochastic nature of a breakdown, this evolution approach is equivalent to decreasing the disagreement point with a random (α_d) coefficient at each time step. In contrast, if the controller decides not to cooperate, the disagreement point needs to be increased to encourage cooperation in future time steps. Thus, the controller increases its expected risk of a breakdown by accepting a higher probability of a breakdown. This higher probability is determined based on the rejected cooperative ramp metering rate, the predicted traffic condition for the next time step, and the resulting risk level. The chosen ramp metering rate at the disagreement point ($d_{disgr-r}(h)$) is used to calculate the disagreement point cost function $\eta_r(k)$ by adding the expected average link travel time and ramp waiting time. The probability of a breakdown and the desired risk level are calculated by modeling the breakdown behavior at merging bottleneck locations.

In the metastable condition, traffic breakdowns occur due to speed disturbances that have amplitudes larger than the critical amplitude (Kerner, 2002). In other words, when traffic flow is high, a smaller speed disturbance can result in a higher probability of a breakdown. The critical amplitude is not deterministic and varies with changes in traffic volume; thus, the probability of a breakdown in a freeway bottleneck is an increasing function of traffic flow (Elefteriadou et al., 1995; Persaud et al., 1998). Furthermore, as shown in Nagalur Subraveti et al. (2019) and in Chapter 3 of this thesis, speed disturbances at merging bottlenecks initially occur in the right lane where merging vehicles enter the freeway and possibly trigger a breakdown, which may propagate to other lanes. However, most of the freeway capacity studies model breakdown probabilities based on total traffic flow in all lanes. In this study, the probability of a breakdown is calibrated and modeled based on both total traffic flow and right lane traffic flow to evaluate the performance of the control model. As thoroughly explained in Chapter 2, the Weibull distribution is one of the most popular life distributions with a flexible shape that enables it to model a wide range of failure events, and it can be theoretically derived as an extreme value distribution (Brilon et al., 2005; Reiss and Thomas, 2007; Chow et al., 2009; Heshami et al., 2019). A 2-parameter Weibull distribution function is calibrated to represent the stochastic capacity at bottlenecks. The probability density function and cumulative distribution function for the Weibull distribution are as follows:

$$P(F_R) = \frac{\lambda}{\mu}\left(\frac{F_R}{\mu}\right)^{\lambda-1} exp(-\left(\frac{F_R}{\mu}\right)^{\lambda}) \tag{4.8}$$

$$C(F_R) = 1 - exp(-\left(\frac{F_R}{\mu}\right)^{\lambda}) \tag{4.9}$$

Where λ and μ are shape and scale parameters, respectively, and $F_R = F_{right} + d_{disgr-r}$ where F_{right} is replaced by F_{total} if total traffic flow is considered to calibrate the distribution function. Further details are presented in Section 4.3.2.

4.3 Numerical analysis

4.3.1. Baseline scenario network and traffic demand

The developed cooperative ramp metering approach is applied to a hypothetical freeway network of a total length of 15 km. As shown in Figure 4.4, the network consists of a three-lane mainstream, three metered on-ramps, ramp 1, ramp 2, and ramp 3 located at the 6 km, 9 km, and 12 km marks, respectively, and three off-ramps 1 km upstream of each on-ramp. The three-lane configuration allows for regenerating the disturbances caused by lane changing activities as observed in real world freeway networks. As a trade-off between computational time and prediction performance, a 5 second update interval for trajectory data from CAVs is chosen, and the data is aggregated over 10-second prediction intervals. The base scenario is defined to replicate busy peak period traffic conditions; thus, vehicles enter the freeway according to a random log-normal distribution function with mean and standard deviations shown in Figure 4.5(a) (X. Chen et al., 2010; Kong & Guo, 2016). An intensive analysis to evaluate the sensitivity of the prediction results to the market penetration rate of CAVs is conducted earlier in Chapter 3. The results confirmed that the lane by lane and overall traffic state parameters can be reliably predicted with 10% penetration of CAVs, except for speed in the right lane, which needs a 30% rate. Thus, CAVs are randomly tagged based on a 30% market penetration rate for this base scenario. Merging vehicles are generated at a uniform average rate that changes over 20-minute time spans as shown in Figure 4.5(b). Off- ramps are also modeled in the benchmark model to consider the disturbances caused by lane changing maneuvers of exiting vehicles. Vehicles are randomly tagged to exit from the off-ramps with exit probabilities of 10% for the first exit, and 20% for the second and third exits.

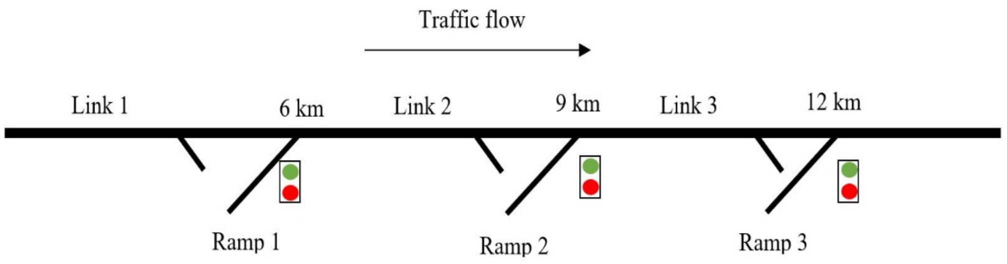
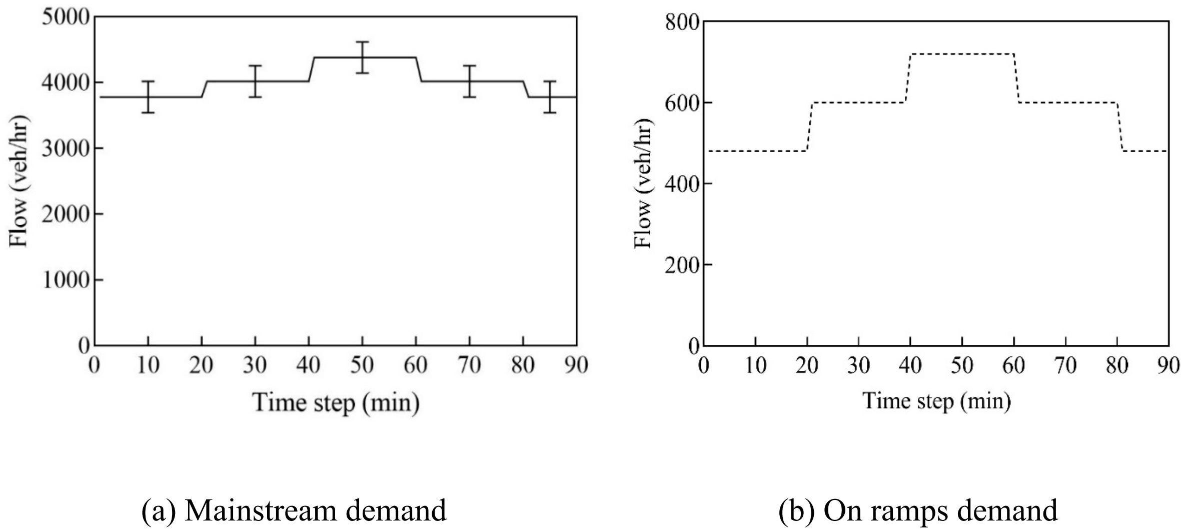


Figure 4.4. Benchmark network with three on-ramps



(a) Mainstream demand (b) On ramps demand

Figure 4.5. Demand scenarios considered in the experiment simulations

4.3.2. Stochastic breakdown model

The probability distribution function for traffic breakdown is analyzed based on 60 random simulation runs using the same traffic demand distribution on all runs. Breakdown events are identified based on the speed threshold of 70 km/hr, which is selected based on the free flow speed (FFS) curves for multi-lane highways and according to prior similar studies (Brilon et al., 2005; Elefteriadou & Lertworawanich, 2003; Ozguven & Ozbay, 2008; HCM, 2010) . Pre-breakdown flows for the right lane, all lanes, and on the on-ramps are recorded. The empirical cumulative distribution function for the summation of pre-breakdown flows and merging flows are plotted and smoothed. Based on the observation of the empirical cumulative distribution, smoothed density function, and the nature of the sample space of pre-breakdown flows, it is statistically reasonable to assume that the sample belongs to an extreme value distribution. The Weibull distribution is one of the most popular extreme value distributions with a flexible shape that enables it to model a wide range of failure events. Thus, the Weibull distribution is selected in this study to model the probabilistic characteristics of breakdown events (Reiss & Thomas, 2007; Heshami et al., 2019).

The Anderson-Darling and Kolmogorov-Smirnov statistical tests and graphical tools are used to fit the proper distribution function to the dataset of pre-breakdown flows and to calibrate the model parameters. The statistical tests measure how well data fits a specific distribution. The null hypothesis is that data follows a specific distribution, and the p-value, t-statistics, and critical values show if the null hypothesis is rejected or not for different significance levels. The analysis shows that pre-breakdown total flows and right-lane flows are distributed as two-parameter Weibull distribution functions. The results for the statistical analysis and the calibrated distribution parameters for both right-lane flows and total flows are shown in Table 4.1 and Table 4.2. The estimated probability density and cumulative probability distributions are shown in Figure 4.6.

Thereafter, the risk level, defined as the probability of a breakdown, can be determined using the cumulative distribution function. In the examined case study, a range of 15% to 50% was chosen as the risk level thresholds according to probabilistic ramp metering approaches in the literature and the calibrated distribution functions (Elefteriadou et al., 2011b; Trapp, 2016).

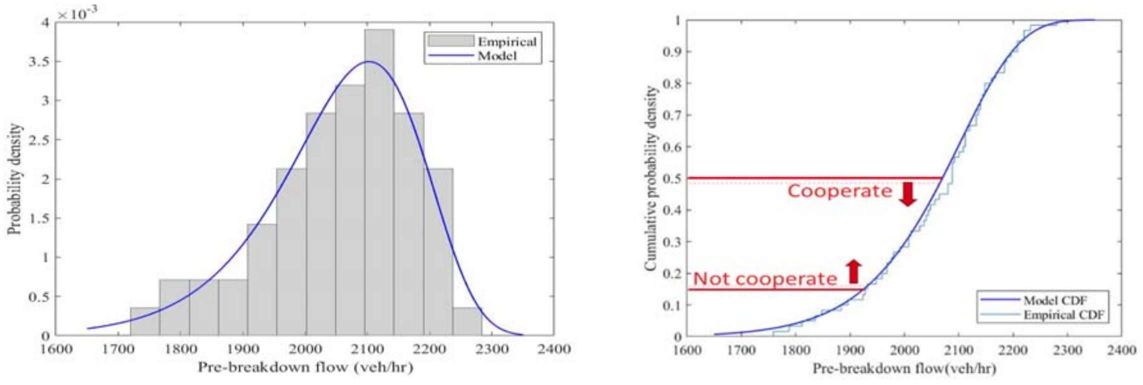
The disagreement point at the first time step is chosen based on an initial risk level of 30%, which is equivalent to a cumulative probability of a breakdown of 30%, and it dynamically evolves through the bargaining process. For instance, if the current accepted risk level results in achieving a cooperative decision, the controller should increase its local performance by accepting a lower risk level for the next time step. For the purpose of this case study, a 5% reduction in risk level to a minimum of a 15% probability of a breakdown is considered. Due to stochastic traffic behavior, this reduction is equivalent to choosing a random (α_d) coefficient in equation (4.7). In contrast, if the current risk level and associated disagreement cost do not result in a cooperative solution, the controller increases its disagreement point to be equal to the current optimal local cost according to (4.7). In this case, the controller rejects the optimum ramp metering rate because it causes a higher risk level. Thus, for the next time interval, the rejected ramp metering rate and the predicted flow is used to calculate the associated probability of a breakdown based on the probability distribution. The calculated probability (risk level) is considered to calculate the next disagreement point. If the increased risk level exceeds the maximum threshold of 50%, then the risk level is considered to be equal to 50%.

Table 4.1 . Kolmogorov-Smirnov (KS) test results for the 2P Weibull distribution of right lane pre-breakdown flows

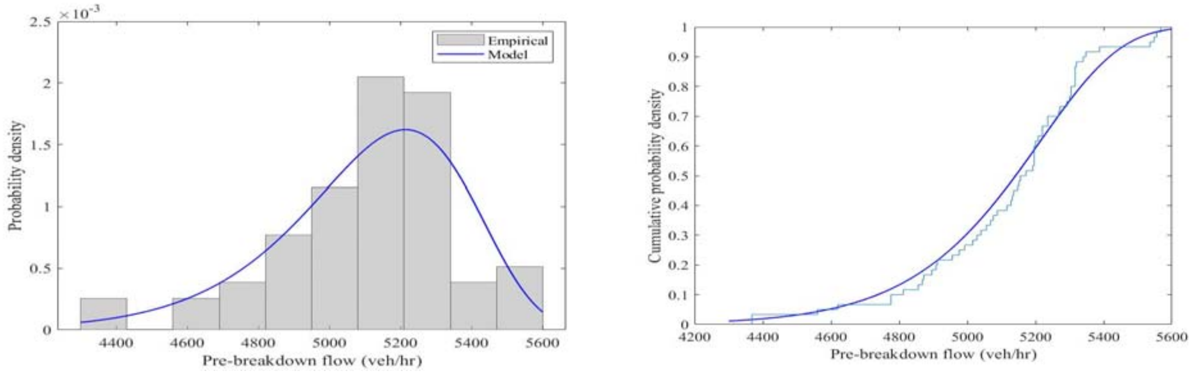
Pre-breakdown flow	KS P-value	Critical value			Shape parameter λ	Scale parameter μ
		α=0.05	α=0.02	α=0.01		
Right-lane flow	0.81	0.17	0.19	0.21	20	2108
Total flow	0.53	0.17	0.19	0.21	23	5224

Table 4.2. Anderson-Darling (AD) test results for the 2P Weibull distribution of total pre-breakdown flows

Pre-breakdown flow	AD Statistic	Critical value			Shape parameter λ	Scale parameter μ
		α=0.05	α=0.01	α=0.01		
Right-lane flow	0.19	2.50	3.29	3.91	20	2108
Total flow	0.68	2.50	3.29	3.91	23	5224



(a)



(b)

Figure 4.6. Probability distribution plots for pre-breakdown flows in a) the right lane and b) all lanes

4.3.3. Ramp metering results for the Base Scenario - Stochastic capacity based on the right lane pre-breakdown flow model

After calibrating the probability distributions for the stochastic capacity at merging bottlenecks, the developed game-theory based ramp metering approach was tested. In the DMPC problem, the simulation step (T_s) was considered to be 10 seconds. The optimal prediction horizon (N_p) and control horizon (N_c) were found to be approximately around 6 minutes and 3 minutes, respectively. These horizons allowed the whole response of the system to be considered, while not taking the farther future demand too much into account. A larger difference among N_p and N_c resulted in reduced performance because only the first sample of the optimized solution was applied to the system. The metered ramp flow is chosen form the range of 1 to 12 vehicles per minute. The benchmark network, online simulation, and traffic prediction and control processes were developed and implemented using the Go 1.14 programming language on a 64-bit Windows PC with 1.8 GHZ Intel Core-i7 and 16 GB of RAM. Go is a modern language that enables fast development and at the same time produces a highly efficient executable on multiple platforms (https://go.dev/). Moreover, Go's concurrency construct offers an efficient way of running a large number of simultaneous computations. Finally, a Go project can be easily changed to work on server or desktop applications, making future development easier.

An interior point algorithm was used to solve the local optimization problems in the DMPC approach. The optimal control was computed and applied to the traffic system every minute. The maximum allowable queue length at on ramps was set to 30 vehicles to prevent spillback onto the street. Initially, the developed ramp metering model was implemented to model the right-lane flow breakdown. The calculated ramp metering rates and the queue evolution on the ramps for the implemented control method are shown in Figure 4.7(a) and 4.7(b), respectively. As shown, as demand increased, ramp metering switches on, and the queue length on the ramps gradually increased to reach approximately the maximum length of 30 vehicles.

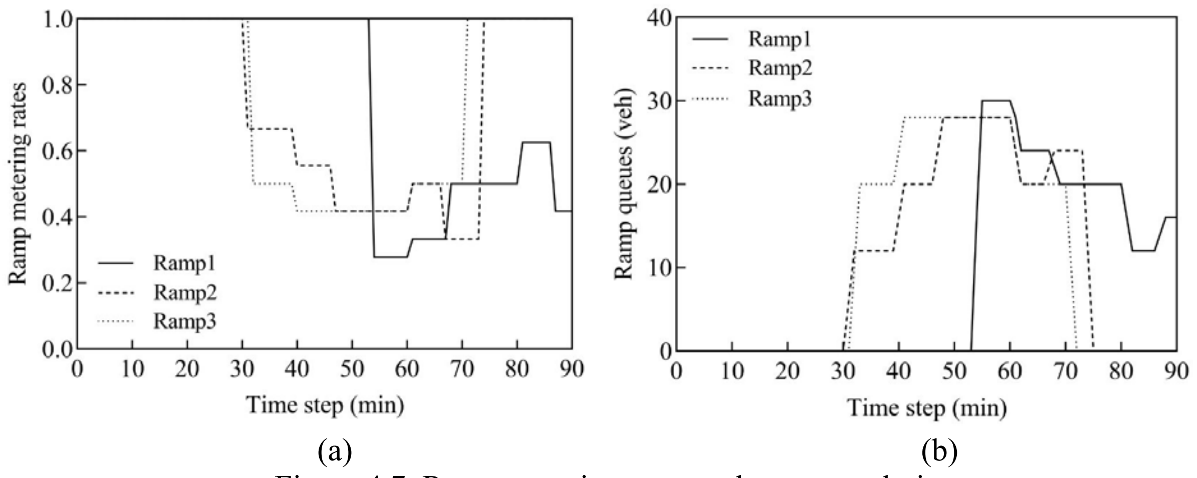


Figure 4.7. Ramp metering rates and queue evolutions

Figure 4.8 compares the evolution of speed, traffic flow, and link travel time for the cooperative ramp metering scenario with the no control case. The figure shows that ramp control actions are effective in proactively mitigating congested traffic conditions, which delays or even prevents the occurrence of traffic breakdowns. The speed evolution on the links in Figure 4.8 shows that, at the earlier time periods, average speed fluctuates between 60 km/hr and free flow speed; these fluctuations in speed indicate merging activities. As merging demand from on-ramps increases, traffic breakdown occurs as a bottleneck forms and congestion starts propagating upstream on links 2 and 1. Using the cooperative ramp metering measures, the congested patterns completely dissolve on link 3, smoothing out traffic patterns on upstream links. This improvement results in higher flow rates and lower link travel times compared to the no control case. The minimum observed speed is also shown to be higher in the controlled case: the average reported speeds are 32, 40, and 56 km/hr for links 1, 2, and 3, respectively, compared to the average speeds of 20, 23, and 23 km/hr, respectively, in the no control case. In general, the total time spent on the network is reduced by 24% compared to the no control case.

In the no control case, as illustrated in the time-space diagrams of vehicle trajectories in Figure 4.9 (a), the perturbation in the vicinity of ramp 1 and ramp 2, caused by vehicles slowing down or changing lanes, promptly results in the emergence of a wide moving jam propagating upstream and triggering traffic to slow down and almost stop. In addition, the propagation of the wide moving jam on link 2 results in the further deterioration of traffic conditions on link 1; consequently, another wide moving jam is formed on link 1. The congestion, more localized on link 3, is mainly due to the capacity drop downstream of the ramp 2 metering mainstream flow that approaches ramp 3; however, a considerable delay is still observed.

Figure 4.9 (b) demonstrates the effectiveness of the cooperative RM measures in damping down shockwave propagations over the entire network. For example, congestion on link 3 is completely prevented. The perturbance on the freeway in the vicinity of ramp 2 causes a localized synchronized flow on a small stretch of the freeway; however, this synchronized flow pattern is shown to quickly resolve, and its impacts are not propagated further upstream. Ramp 1 still exhibits some perturbation and shows wave formation, but the impact is significantly reduced. In addition, a shockwave forms on link 1 because link 1 has a higher mainstream demand compared to the other links as 20% of the initial demand exits from each downstream off-ramp. In addition, the limited queue storage on the ramp does not allow further increases in the ramp metering rate to prevent congestion. Despite this limitation, the time duration of congested conditions and the spatial propagation of wide moving jams are both significantly reduced.

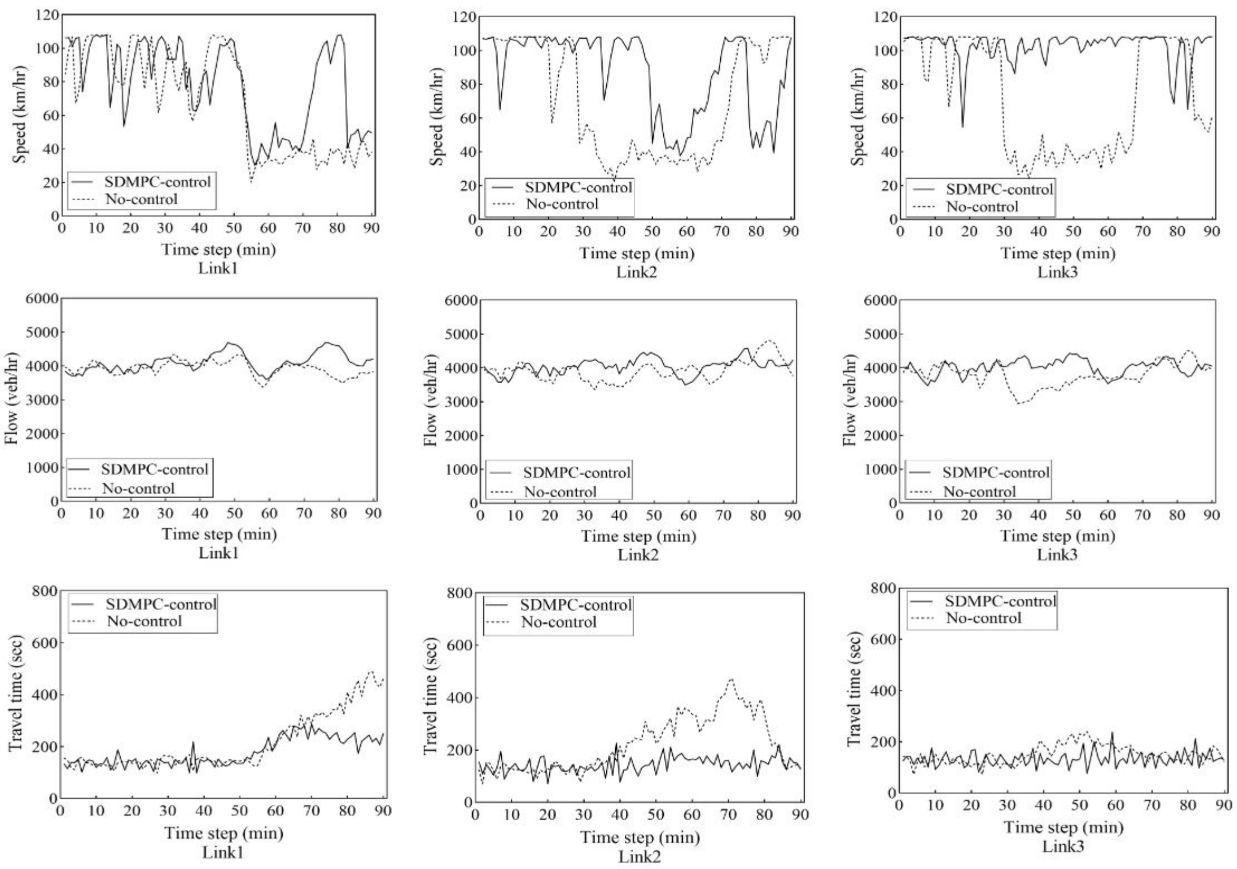
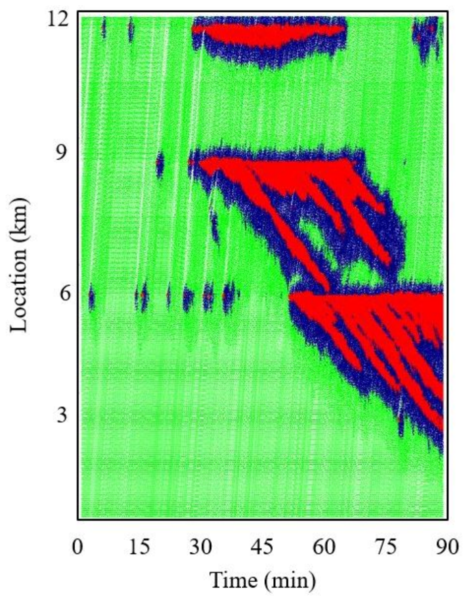
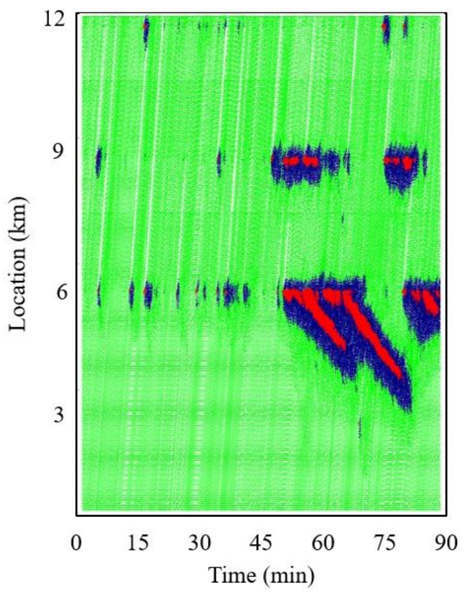


Figure 4.8. Speed, flow, and link travel time evolutions under the SDMPC ramp metering implementation vs. no control case



(a)



(b)

🟩 Speed above 70 km/hr 🟦 Speed 30 to70 km/hr 🟥 Speed below 30 km/hr

Figure 4.9. Breakdown behavior on the network under (a) no control and (b) the SDMPC ramp metering implementation

4.3.4. *Measures of performance: efficiency, effectiveness, and equity*

It has been noted in previous freeway control studies that ramp metering measures may not be effective if the ramp queue storage is limited, the capacity and/or density constraints are not properly chosen, or traffic flow in the mainstream is too high (Hegyi et al., 2005; Kerner, 2007; Papamichail et al., 2010). However, the performance of ramp metering methods has mainly been evaluated solely based on their efficiency in reducing delays and vehicle time spent on the entire network. To the authors knowledge, the long-term effectiveness of ramp metering methods in terms of the number of times that they fail to address congested conditions has not been evaluated as a measure of performance. In addition, the few studies that investigated ramp metering performance in terms of equity considerations are mainly focused on an equitable ramp waiting times (Meng & Ling, 2010; Papamichail et al., 2010). Therefore, in this research, the performance of the proposed model is evaluated based on three critical aspects: efficiency, effectiveness, and equity. The focus is on the impacts of stochastic capacity and the cooperative solution algorithm. For this purpose, the following scenarios are examined:

Scenario 1: No control

Scenario 2: Stochastic capacity based on right lane pre-breakdown flow model (proposed base model)

Scenario 3: Stochastic capacity based on total pre-breakdown flow model

Scenario 4: Deterministic capacity centralized MPC

Scenario 5: Deterministic capacity distributed MPC

Scenario 6: ALINEA

For the deterministic capacity-based approaches in Scenario 4 and Scenario 5, a capacity of 4700 vehicles per hour (veh/hr) is considered; this capacity is equivalent to 90% of the most frequent observed value of pre-breakdown flow. In Scenario 6, ALINEA (Asservissement Line´aired'Entre´e Autoroutie´re) is implemented, which is a widely used method in real on-ramp installations. It has been frequently used as a benchmark to compare to other ramp metering methods from all categories. ALINEA is a local and responsive feedback control approach that has been shown to successfully ameliorate traffic conditions at freeway on-ramps, especially when the admissible queue length on the ramp is unlimited or extremely large (Papamichail et al., 2010).

A) Efficiency evaluation

The efficiency of the implemented scenarios is evaluated in terms of average travel time (seconds) and the total time spent (TTS) (veh.hr). In Figure 4.10, the evolution of the average travel time for the implemented scenarios is plotted for comparison. All control scenarios show significant improvement in terms of average travel times compared to the no control case. While scenarios 2 and 3 are both stochastic capacity-based methods, scenario 2, which considers the right-lane breakdown model, performs slightly better. Scenarios 4 and 5 are both deterministic capacity-based approaches, but the centralized MPC approach in Scenario 4 reduces average travel time more than in scenario 5. ALINEA successfully performs: it has similar results to scenarios 2 and 3 until around t=60 min. Increasing traffic demand significantly reduces the efficiency of ALINEA because each independent ALINEA responds to the congested condition when it reaches its area and is detected by the feedback detectors, which may result in a late response; in the meantime, congestion starts to propagate upstream. In addition, the limited ramp queue storage prevents ALINEA from mitigating the congestion.

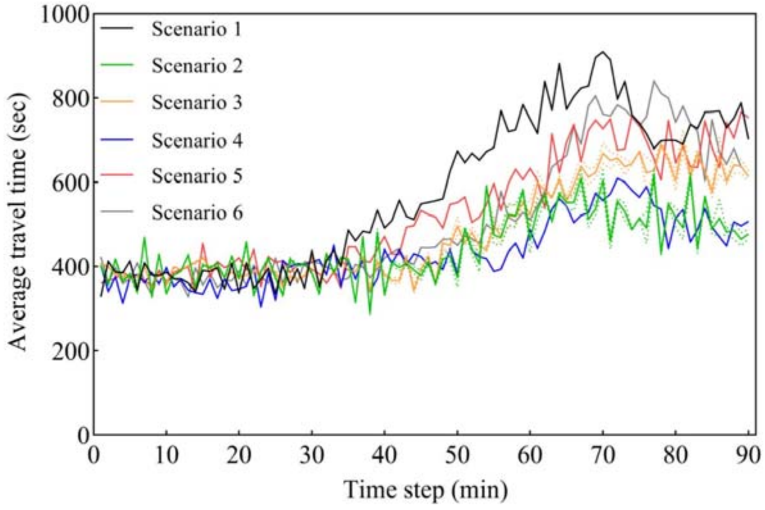


Figure 4.10. Average travel times for the examined control scenarios

To better compare the performance of the implemented scenarios, TTS on the network by mainstream and merging vehicles, including waiting times on the ramps, are calculated. Table 4.3 presents the TTS and the % improvement for each of the examined scenarios. According to the results, TTS is reported as 680 (veh.hr) for scenario 2 and 725 (veh.hr) for scenario 3; therefore, a 5% increase in efficiency when a right-lane breakdown probability is assumed. This outcome indicates the superior performance of the right-lane pre-breakdown model in proactively capturing stochastic breakdown behavior. Thus, the controllers in this scenario can make more timely and efficient ramp metering decisions. Overall, Table 4.3 reports that scenario 4 shows the best performance in terms of TTS because it improved by 27% compared to the no control case; this improvement is 3% higher than that obtained by the SDMPC approach. The higher efficiency in scenario 4 is mainly due to the centralized framework, which finds a system-wide optimal solution to the ramp metering problem. ALINEA shows less improvement in TTS because it tries to maintain downstream density at approximately the critical density by preventing merging vehicles on the ramp for as long as necessary. Consequently, ramp queues reach their maximum length most of the time, which increases ramp waiting times.

Table 4.3. Total time spent on the freeway network in the examined scenarios

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Scenario 6
TTS (Veh.hr)	891	680	725	653	772	789
SD of TTS	-	24	39	-	-	-
Improvement	Benchmark	24%	19%	27%	14%	12%

B) Effectiveness evaluation

Effectiveness is usually defined as producing the intended or expected outcomes consistently and sustainably over time (Drucker, 1963; Griffin & Moorhead, 2010; McDonough and Braungart, 2013). For traffic control methods, a certain method is considered effective if it successfully improves traffic conditions for the given assumptions and constraints over long-term operation; efficiency elaborates on the level of improvement with a short-run perspective. As mentioned earlier, ramp metering measures may not be always successful because of physical constraints or inadequate capacity. To be more specific, due to the uncertain behavior of freeway capacity at merging bottlenecks, assuming a deterministic value for capacity may result in the underutilization of infrastructure or, in extreme cases, failure of the control measure when larger capacities are considered. Thus, due to the stochastic nature of traffic demand and supply (e.g., capacity), the performance of control measures in responding to traffic conditions may not be consistent over different periods of freeway operation. In other words, despite control actions, breakdowns may still occur because of various uncertainties. For this research, effectiveness is defined as the number of breakdown events that occur during each scenario.

Sixty random simulation runs were conducted using the stochastic microscopic simulation model. The simulation runs represented 60 peak periods over the course of freeway operation. In the no control case (scenario 1), breakdowns occurred in all simulation runs on link 1 and in 52 out of 60 simulation runs on link 2. The number of breakdowns decreased for link 3 despite the same ramp demand as other locations. The absence of a breakdown on ramp 3 could be explained by the capacity drop phenomena at the upstream bottlenecks, which metered traffic flow and density on link 3. In addition, a portion of the initial demand exited the network from three upstream off-ramps, further reducing the mainstream demand at the location of the third bottleneck. Overall, considering 180 situations in which a breakdown could occur during the 60 simulation runs and 3 bottlenecks, 148 breakdown events were observed, which was equivalent to 82% of all possibilities.

When ramp metering measures were implemented, the number and duration of breakdown events were significantly reduced for all scenarios; however, a number of congested patterns were still observed. In these cases, the control actions were considered to be insufficient (i.e., they failed) if the improvement in the average link travel time compared to the no control case was less than 5%; in this case, the scenario was considered as a breakdown occurrence. Table 4.4 presents the number of breakdowns, according to the above definition, for each link during the 60 simulation runs. In general, the stochastic capacity-based approaches with bargaining game solution (scenarios 2 and 3) were shown to be more effective; these scenarios experienced a failure rate of 17% (scenario 2) and 21% (scenario 3) in the control actions out of the total breakdown possibilities because the approaches more realistically model actual breakdown behavior. As indicated in Table 4.4, scenario 2 showed the best performance in terms of stopping and dissolving congestion. The effectiveness of scenario 2 was attributed to three main characteristics of the proposed model: i) lane by lane predictions resulted in a more timely and precise anticipation of traffic state, ii) the bargaining game solution locally prevented breakdowns when necessary, and iii) the stochastic capacity considerations especially as a function of merging and right lane activities captured probabilistic breakdown behaviors. Interestingly, ALINEA had a lower breakdown ratio (24%) compared to scenarios 4 and 5 even though its efficiency was lower due to larger queueing times. The main reason for this result could be the less sensitivity of ALINEA to uncertain capacities because it targeted critical occupancy instead. Consequently, ALINEA was more successful than the approaches in scenarios 4 and 5 in preventing breakdowns over several periods of operation despite the approach lacking the predictive and coordinative features of the other approaches.

Table 4.4. Number of breakdown events that occurred during the 60 simulation runs for all scenarios

Control cases	Number of breakdown events				
	Link 1	Link 2	Link 3	Total number	Total percentage
Scenario 1	60	52	36	148	82%
Scenario 2	13	10	8	31	17%
Scenario 3	14	13	11	38	21%
Scenario 4	12	25	10	47	26%
Scenario 5	16	20	15	50	28%
Scenario 6	21	15	8	44	24%

C) Equity evaluation

Most studies in the literature focused on achieving equity over each control cycle and equitable waiting times in ramp queues. Papamichail et al. (2010) considered average time for queuing and traveling 6.5 km downstream as measures of equity over the control period; they also considered a more balanced travel time among all controlled ramps to be a more equitable solution. A few studies tried to achieve an optimal weighted travel time or optimal equity index over each control cycle (Khoo, 2011; Meng & Ling, 2010; Q. Tian et al., 2012; Lei Zhang & Levinson, 2005), but long-term equity impacts of ramp metering have not been evaluated in any of these studies. In this research, the equitability of the proposed ramp metering is evaluated from two aspects. First, short-run equity is evaluated in terms of the average value of the sum of queueing time on each ramp and travel time to the downstream ramp location. This evaluation measure of ramp equity is similar to the one used by Papamichali et al. (2010). The second aspect, which is unique to this research, is expressed in terms of the fair distribution of the impacts of RM strategies. This aspect can be expressed in terms of the frequency of occurrence of failed control actions among controllers. These measures are included because failures are unavoidable due to the stochastic nature of traffic demand and supply. In other words, with certain control measures, some ramps may become congested more frequently than others for the sake of system optimality. Thus, an RM scheme is fairer when it results in a fair distribution of failure occurrences.

Figure 4.11 presents the average times for queuing and traveling to the downstream ramp for all scenarios. In general, the evaluated travel time shows some degree of balance among all ramps for all MPC-based scenarios compared to the no-control and ALINEA scenarios. This result is achieved by distributing the burden of travel time reduction among controllers through coordinated ramp metering and the imposed queue constraints. The bargaining game-based methods (scenarios 2, 3, and 5) show more balanced results compared to the centralized MPC (scenario 4), this balance is gained by the incentive-based cooperation among controllers. This fairness, however, is achieved at the expense of a lower TTS improvement in scenarios 2, 3, and 5, compared to scenario 4, since equity and efficiency are partially competing properties.

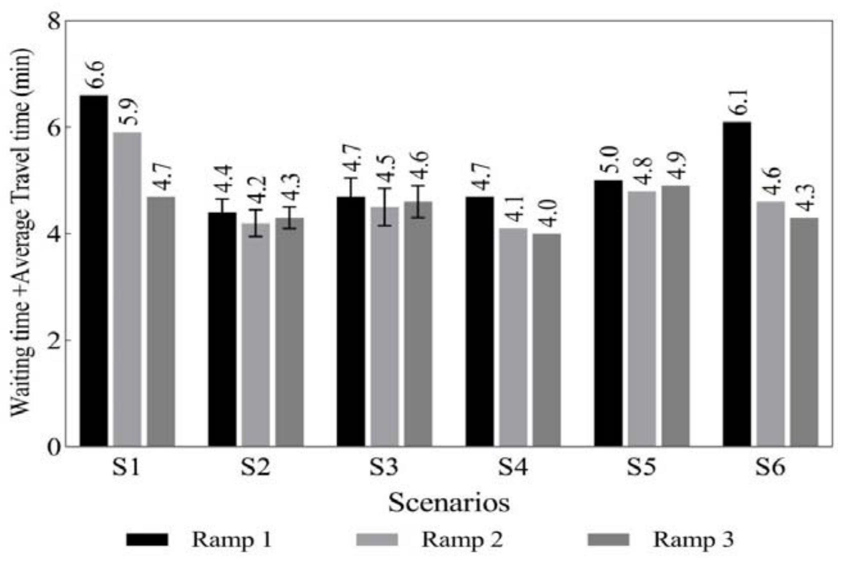


Figure 4.11. Average queuing waiting times plus traveling downstream link

Long-run equity properties of the implemented scenarios are evaluated in terms of the balanced distribution of the number of failed ramp metering solutions produced by the scenarios; failure is considered as a travel time reduction of less than 5% compared to the no control case. Stochastic characteristics of traffic demand and supply cause inconsistencies in the control outcomes in real world applications, while no studies to date have evaluated inadequacies as a measure of fairness. Table 4.4 indicates the number of breakdowns for each link over the 60 simulation runs. While the total number of breakdowns is considered as a measure of effectiveness of the implemented scenarios, the balance of these numbers among controllers can be considered as an equity measure. These measures can indicate equity for regular commuters that frequently enter the network from a specific ramp by identifying how often the commuters encounter congestion compared to commuters who enter from other ramps. As shown in Table 4.4, in scenarios 2 and 3, there is a balance among the number of breakdown events among controllers within a range of 8 to 13 and 10 to 14 breakdowns, respectively. Thus, these scenarios are more equitable solutions compared to scenario 4, which has a wide range (10 to 25) in the number of breakdowns for different ramps. For instance, in scenario 4, over the course of 60 periods (e.g., days), vehicles merging from ramp 1 are likely to experience congestion 25 times on link 2 due to the failure of the downstream controller; this number is around 10 times for vehicles entering from ramp 3 over the same period of operation. This inequality in scenario 4 is mainly caused by the imposed deterministic capacity, which results in the MPC model not adequately handling dynamic and stochastic breakdown behavior. ALINEA also fails to perform in terms of long-run equity.

In general, the simulation results show that the proposed DMPC with the bargaining game solution (Scenario 2) outperforms the centralized MPC and uncoordinated ALINEA in terms of short-run and long-run equity and effectiveness measures. However, this superiority is achieved by compromising system-wide efficiency in terms of total time spent on the network; in other words, there is an approximate 3% loss of performance. The cooperative properties of the bargaining game-based approach attain higher equity and effectiveness performance by facilitating communication among the controllers: each controller receives the traffic state and decision information from other controllers. In contrast, in the centralized MPC of scenario 4, the optimized solution is calculated based on a controller's assumption of the decisions of other controllers and not the actual data from the other controllers; consequently, the solution may be less effective. In addition, the ability of the controllers to determine their expected local costs based on the probabilistic breakdown model and their potential to make local decisions improves the effectiveness of the proposed approach.

CHAPTER 5: SUMMARY OF RESEARCH FINDINGS AND CONCLUSIONS

This chapter presents a summary of the research, concluding remarks and provides potential directions for novel research. Sections 5.1, 5.2 and 5.3 share an overall research summary and the key findings related to Chapters 2, 3 and 4. The areas that may be of interest for future study are suggested in Section 5.4.

5.1 Research contributions and findings on deterministic and stochastic freeway capacity modeling

The definition, estimation, and stochastic nature of freeway capacity were abundantly investigated in previous studies. However, the influence of weather conditions on the stochasticity of freeway jam density and capacity and its distribution were not thoroughly examined. The main contributions of Chapter 2 were as follows:

1) *Modeling and calibration of deterministic capacity under various weather conditions.* The calibrated segmented regression algorithm calculates a break-point in the data set based on an iterative calculation process, which is the point of slope change in the triangular regression (critical density). The advantage of using this algorithm compared to previously developed fundamental diagram regression models is that in this segmented regression approach the calibration is completely based on the statistical analysis of the whole data set ; thereby avoiding imposing statistically biased thresholds to divide the data set into the free flow and congested sections.

Fixed values of capacity were first derived from the fundamental diagrams; then the impact of weather conditions on the maximum hourly flow, the distribution of capacity and jam densities were investigated. The results showed that, even if the deterministic value is considered, the effect of weather conditions on capacity and jam density is statistically significant and may not be ignored due to their considerable fluctuations under different weather conditions.

2) *Modeling and calibration of probabilistic capacity under various weather conditions.* The stochastic capacity of the freeway was also explored based on analyzing the frequency of different pre-breakdown flows. A Weibull distribution function was calibrated for each type of weather conditions (i.e. snow, rain, clear, etc.). The statistical analysis showed that the difference among the parameters of the distributions are significant for different weather conditions.

5.2 Research contributions and findings on short-term traffic prediction in a connected vehicle environment

Chapter 3 presented a traffic state prediction approach based on a stochastic microscopic three-phase model. The model consists of three modules: 1) online simulation, 2) data fusion and prediction, and 3) spatial-temporal traffic pattern tracking and prediction. The developed model utilizes speed and location information from floating vehicle data (e.g., probes and CAVs) to estimate traffic state parameters based on a stochastic microscopic model. These estimated traffic parameters are then fused with fixed detector measurements using an AKF to obtain traffic state predictions on a lane by lane basis over a short time horizon. This predicted information is in turn used to dynamically track and predict the spatial-temporal congestion patterns.

The developed model contributes to the body of knowledge as follows:

1) Developing a short-term microscopic-based traffic prediction model that incorporates the stochasticity of driver behaviors. The developed microscopic model considers stochastic driver behaviors such as lane changing, over acceleration, and speed adaption effects.

These microscopic decisions are the main triggers of traffic breakdowns on freeways. Additionally, rather than using historical traffic data, the model uses information from previous time intervals to continuously self-adapt and calibrate its error parameters once predicted information becomes available as measurements; thereby increasing the model's efficiency for online applications.

2) *Predicting the traffic parameters on a lane by lane basis and predicting the lane by lane spatiotemporal congestion patterns.* The developed prediction model produces lane by lane predictions of traffic parameters which include traffic flow and speed on a lane by lane basis, predicted link travel time, and predicted spatiotemporal congested patterns. This information can be used as critical inputs to trigger proactive freeway and lane management control measures. The developed model relies on emerging data collection technologies (e.g. CAV) to reproduce and predict the desired microscopic/macroscopic traffic flow characteristics and spatiotemporal patterns lane by lane. This lane-based information can be transmitted to other CAVs or road infrastructure to activate control actions needed to proactively delay or even prevent the propagation of congested traffic patterns.

3) *Addressing the inconsistency of measurement problem generated by the multi-type sensors (i.e. detector and CAV data) by the utilization of the adaptive Kalman filtering (AKF) method.* A unique characteristic of the developed model is combining the complementary aspects of both the theoretical model-based and data-based approaches to synergize the levels of accuracy and efficiency of the outcomes. More specifically, the proposed model carries the explanatory power of the model-based microscopic traffic, which enables the model to reconstruct and predict recurrent and non-recurrent traffic through an online simulator. Meanwhile, its efficient recursive

and adaptive process enables lane by lane traffic parameters prediction for short horizons with parallel accuracy to that of data-driven models, yet without similar intensive data requirements.

The results of the developed approach are shown to outperform those of the benchmark time series-based prediction models. The predictions are tested under several scenarios with numerous simulation runs to determine their performance. The examined scenarios include 1-minute and 2-minute prediction intervals and different levels of market penetration rates of CAVs. The rigorous analysis carried out shows that the proposed model-based traffic prediction method predicts traffic state parameters for short time intervals of 1 minute with an accuracy comparable to that of data-driven models.

Lane by lane and overall traffic state parameters are shown to be reliably predicted when combined information from fixed sensor data with only 10% penetration of CAVs; however, a similar level of reliability of prediction could not be obtained for the resulting speed predictions for the right lane. These outcomes can be explained by the stochasticity of drivers' behavior when driving in the right lane and the need to take more frequent decisions pertaining to merging, deceleration, and speed adaption. Yet, the comparison of the results with SARIMA+KF and SARIMA+GARCH+AKF benchmark models showed the superiority of the proposed method especially in terms of speed predictions.

The key findings showed that it is important that lane by lane traffic state predictions are examined due to the various mechanisms in the emergence and propagation of congested patterns in different lanes of freeways. The results of the runs clearly show the considerable time lag between the onset of a breakdown in the right lane compared to the later overall speed breakdown at a bottleneck.

The other unique contribution is the spatial-temporal congested traffic patterns that are also predicted on a lane by lane basis. The results demonstrated that the proposed model reliably reproduces and predicts synchronized flow and wide moving jams, including jam fronts and wave speeds in the right lane. However, the outputs do not seem promising for other lanes, especially in identifying wide moving jams. More work needs to be conducted in the future to improve that.

5.3 Research contributions and findings on cooperative ramp metering in a connected vehicle environment

In Chapter 4, a dynamic predictive and cooperative ramp metering approach that considers the stochastic behavior of freeway capacity in terms of total flow and right lane flow breakdown probability models is developed. In this chapter, the ramp metering problem is modeled based on a distributed model control algorithm, which is solved based on a bargaining game approach. In the bargaining game, each controller, a player in the game, solves the local optimization problem simultaneously based on its own costs and constraints and the information received from the other controllers.

1) Developing a predictive and cooperative ramp metering model based on a distributed model predictive control approach. The developed distributed model facilitates the communication among controllers. Based on the received information, the local RM controller can choose not to cooperate based on the expected local probability of a breakdown because avoiding a local breakdown is considered a higher local priority compared to minimizing total system travel time. This unique property allows for a more equitable distribution of breakdown events, while seeking system-wide efficiency. In other words, the proposed approach maintains the effectiveness of local control decisions and achieves system-wide efficiency whenever it is possible.

The performance of the proposed model is evaluated and compared based on six scenarios: no control, stochastic capacity based on a right lane pre-breakdown flow model, stochastic capacity based on a total pre-breakdown flow model, deterministic capacity-based centralized MPC, deterministic capacity-based distributed MPC, and a local feedback control (ALINEA). The performances of the scenarios are evaluated and compared based on efficiency, effectiveness, and equity considerations.

The results showed that the proposed model consistently outperforms the deterministic capacity-based models in terms of effectiveness and equity of the ramp metering solutions. Effectiveness is defined in terms of the control method's ability to improve traffic conditions, consistently, under dynamic and changing traffic behavior over operation periods; whereas the efficiency measure elaborates on the level of improvement.

In this research, effectiveness is introduced as a novel measure of performance, and it is evaluated based on the number of failed control actions. Equity is measured based on a more balanced distribution of the failures and queuing and travel time among controllers. In respect to efficiency, defined as a reduction in total time spent on the network, the centralized approach is shown to perform slightly better than the proposed model (3% more reduction in TTS). Thus, the proposed bargaining game theory approach is capable of finding solutions with a balanced trade off between equity and efficiency. In other words, the developed approach successfully addresses the equity and effectiveness issues in RM without sacrificing efficiency.

5.4 Future extensions

Based on the outcomes of Chapter 2, it is suggested that the stochastic behavior of freeway capacity under various weather conditions be modeled, and the appropriate probabilistic model be selected based on the real-time weather condition reports. For future studies, the proposed approach can be further combined with reliability analysis of probabilistic capacity. Such analysis can help traffic engineers select the most appropriate quantile of the probabilistic model based on the desired level of service. Another potential research subject is using the same data set in hazard functions to predict the time of breakdown occurrence. In addition, previous studies have shown that the freeway capacity may change according to the vehicle mix and the proportion of heavy vehicles to standard vehicles; thus, evaluating their impacts on the probabilistic behavior and distribution of freeway capacity would be an interesting area to explore.

There are several research directions that can be recommended for future research according to the outcomes of Chapter 3. The next step of this research should include the validation of the model performance using a different microscopic traffic simulation model or real observed data (e.g. NGSIM). In this research, CAVs are assumed to have similar driving behavior as human driven vehicles. In heavy traffic, CAV driving behavior also influence and impact non-equipped vehicles travelling nearby which have to mirror their driving behavior. An extension of this work should incorporate distinct car-following and lane-changing models for CAV and human driven vehicles. In future studies, more advanced AKF methods, such as correlation techniques that can produce unbiased estimations of noise statistics can be investigated in the context of real-time traffic prediction problems. In addition, application of other advanced filtering approaches that are able to handle various error distributions (e.g. particle filters) is another promising approach to explore.

A potential extension to the ramp metering model in Chapter 4 is to consider other equity concerns that can be modeled using the definition of disagreement point or unequal weight factors of the utility function of the bargaining game. Integrating multiple control measures such as variable speed limit and ramp metering using the proposed bargaining game framework is another interesting addition that may have a synergetic effect reducing the number of failure events in freeway congestion control strategies.

REFERENCES

Alessandri, A., Febbraro, A. Di, Ferrara, A., & Punta, E. (1999). Nonlinear optimization for freeway control using variable-speed signaling. *IEEE Transactions on Vehicular Technology*, *48*(6), 2042–2052.

Aljamal, M. A., Abdelghaffar, H. M., & Rakha, H. A. (2020). Real-Time Estimation of Vehicle Counts on Signalized Intersection Approaches Using Probe Vehicle Data. *IEEE Transactions on Intelligent Transportation Systems*, *January*, 1–11.

Allström, A. (2016). Highway Traffic State Estimation and Short-term Prediction. In *Highway Traffic State Estimation and Short-term Prediction* (Issue 1749).

Alvarez, I., & Poznyak, A. (2010). *Game Theory Applied to Urban Traffic Control Problem*. *1*, 2164–2169.

Bekiaris-Liberis, N., Roncoli, C., & Papageorgiou, M. (2016a). Highway traffic state estimation with mixed connected and conventional vehicles. *IEEE Transactions on Intelligent Transportation Systems*, *17*(12), 3484–3497.

Bekiaris-Liberis, N., Roncoli, C., & Papageorgiou, M. (2016b). Highway Traffic State Estimation with Mixed Connected and Conventional Vehicles. *IFAC-PapersOnLine*, *49*(3), 309–314.

Bekiaris-Liberis, N., Roncoli, C., & Papageorgiou, M. (2017). Traffic State Estimation per Lane in Highways with Connected Vehicles. *Transportation Research Procedia*, *27*, 921–928.

Bellemans, T., De Schutter, B., & De Moor, B. (2003). An Integrated ControlModel for Freeway Corridor Under Nonrecurrent Congestion. *American Control Conference, 2003. Proceedings of the 2003*, *5*(4), 4077–4082 vol.5.

Benouar, H. (2004). *Equity analysis of freeway ramp metering*. University of California, Berkeley.

Brilon, W., & Geistefeldt, J. (2009). Implications of the Random Capacity Concept for Freeways. *International Symposium on Freeway and Tollway Operations*, 1–11.

Brilon, W., Geistefeldt, J., & Regler, M. (2005). Reliability of Freeway Traffic Flow. *Transportation and Traffic Theory*, *July*, 125–144.

Cai, P., Wang, Y., Lu, G., Chen, P., Ding, C., & Sun, J. (2016). A spatiotemporal correlative k-nearest neighbor model for short-term traffic multistep forecasting. *Transportation Research Part C: Emerging Technologies*, *62*, 21–34.

Cassidy, M. J., & Bertini, R. L. (1999). Bservations at a. *International Symposium of Transportation and Traffic Theory. Amsterdam, Netherlands*, 107–124.

Chen, D., & Ahn, S. (2018). Capacity-drop at extended bottlenecks: Merge, diverge, and weave. *Transportation Research Part B: Methodological*, *108*, 1–20.

Chen, O. J., & Ben-Akiva, M. E. (1998). Game-Theoretic Formulations of Interaction Between Dynamic Traffic Control and Dynamic Traffic Assignment. *Transportation Research Record: Journal of Transportation Research Board*, *1617*(1), 179–188.

Chen, X., Li, L., & Zhang, Y. (2010). A markov model for headway/spacing distribution of road traffic. *IEEE Transactions on Intelligent Transportation Systems*, *11*(4), 773–785.

Chen, Y. Y., Lv, Y., Li, Z., & Wang, F. Y. (2016). Long short-Term memory model for traffic congestion prediction with online open data. *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, 132–137.

Chow, A., Dadok, V., Dervisoglu, G., Gomes, G., Horowitz, R., Kurzhanskiy, A. A., Kwon, J., Lu, X. Y., Muralidharan, A., Norman, S., Sánchez, R. O., & Varaiya, P. (2009). TOPL: Tools for operational planning of transportation networks. *2008 Proceedings of the ASME Dynamic*

Systems and Control Conference, DSCC 2008, *PART B*, 1341–1348.

Chrobok, R., Pottmeier, A., ur Marinosson, S., & Schreckenberg, M. (2002). On-line simulation and traffic forecast: Applications and results. *Cell*, *3*(1), 2.

Chu, L., Oh, J.-S., & Recker, W. (2005). Adaptive Kalman Filter Based Freeway Travel time Estimation. *Transportation Research Board 2005 Annual Meeting*, 1–21.

Chun, Y., & Thomson, W. (1990). Nash solution and uncertain disagreement points. *Games and Economic Behavior*, *2*(3), 213–223.

Comert, G., Bezuglov, A., & Cetin, M. (2016). Adaptive traffic parameter prediction: Effect of number of states and transferability of models. *Transportation Research Part C: Emerging Technologies*, *72*, 202–224.

Deng, W., Lei, H., & Zhou, X. (2013). Traffic state estimation and uncertainty quantification based on heterogeneous data sources: A three detector approach. *Transportation Research Part B: Methodological*, *57*, 132–157.

Dervisoglu, G., Gomes, G., & Horowitz, R. (2009). Automatic Calibration of the Fundamental Diagram and Empirical Observations on Capacity. *Transportation Research Board*, *January*, 1–14.

Dhingra, S. L & Gull, I. (2008). Traffic Flow Theory Historical Research Perspectives. IIT Bombay, India Transportation *Research Circular E-C149: 75 Years of the Fundamental Diagram for Traffic Flow Theory.*

Di, X., Liu, H. X., & Davis, G. A. (2010). Hybrid extended Kalman filtering approach for traffic density estimation along signalized arterials. *Transportation Research Record*, *2188*, 165-173.

Dong, S., Mostafizi, A., Wang, H., & Li, J. (2018). A stochastic analysis of highway capacity: Empirical evidence and implications. *Journal of Intelligent Transportation Systems: Technology, Planning, and Operations*, *22*(4), 338–352.

Drucker, P. F. (1963). Managing for business effectiveness. *Harvard Business Review*, 41(3), 53–60.

Duret, A., Ahn, S., & Buisson, C. (2012). Lane flow distribution on a three-lane freeway: General features and the effects of traffic controls. *Transportation Research Part C: Emerging Technologies*, *24*, 157–167.

Duret, A., & Audin, R. M. (2009). Spatio-Temporal Analysis of Impacts of Lane Changing Consistent with Wave Propagation. *Word Journal Of The International Linguistic Association*, 1–16.

Duret, A., & Yuan, Y. (2017). Traffic state estimation based on Eulerian and Lagrangian observations in a mesoscopic modeling framework. *Transportation Research Part B: Methodological*, *101*, 51–71.

Elefteriadou, A. (1994). A probabilistic model of breakdown at freeway-merge junctions. Ph.D. Thesis, Northwestern University.

Elefteriadou, L., Kondyli, A., Washburn, S., Brilon, W., Lohoff, J., Jacobson, L., Hall, F., & Persaud, B. (2011a). Proactive ramp management under the threat of freeway-flow breakdown. *Procedia - Social and Behavioral Sciences*, *16*, 4–14.

Elefteriadou, L., Kondyli, A., Washburn, S., Brilon, W., Lohoff, J., Jacobson, L., Hall, F., & Persaud, B. (2011b). Proactive ramp management under the threat of freeway-flow breakdown. *Procedia - Social and Behavioral Sciences*, *16*, 4–14.

Elefteriadou, L., & Lertworawanich, P. (2003). Defining, Measuring and Estimating Freeway Capacity. *82nd Annual Transportation Research Board Meeting, 12-16 January, Washington D.C.*

Evans, J. L., Elefteriadou, L., & Gautam, N. (2001). Probability of breakdown at freeway merges using Markov chains. *Transportation Research Part B: Methodological*, *35*(3), 237–254.

Fei, X., Lu, C. C., & Liu, K. (2011). A bayesian dynamic linear model approach for real-time short-term freeway travel time prediction. *Transportation Research Part C: Emerging Technologies*, *19*(6), 1306–1318.

Ferrara, A., Nai Oleari, A., Sacone, S., & Siri, S. (2012). An event-triggered Model Predictive Control scheme for freeway systems. *Proceedings of the IEEE Conference on Decision and Control*, 6975–6982.

Geistefeldt, J. (2010). Consistency of stochastic capacity estimations. *Transportation Research Record*, *2173*, 89–95.

Ghods, A. H., Fu, L., & Rahimi-Kian, A. (2010). An efficient optimization approach to real-time coordinated and integrated freeway traffic control. *IEEE Transactions on Intelligent Transportation Systems*, *11*(4), 873–884.

Griffin, R. W., & Moorhead, G. (2010). *Organizational Behavior: Managing People and Organizations*. Issue May 2017.

Gu, Y., Lu, W., Qin, L., Li, M., & Shao, Z. (2019). Short-term prediction of lane-level traffic speeds : A fusion deep learning model ☆. *Transportation Research Part C*, *106*(July), 1–16.

Guo, J., Huang, W., & Williams, B. M. (2014a). Adaptive Kalman filter approach for stochastic short-term traffic flow rate prediction and uncertainty quantification. *Transportation*

Research Part C: Emerging Technologies, *43*, 50–64.

Guo, J., Huang, W., & Williams, B. M. (2014b). Adaptive Kalman filter approach for stochastic short-term traffic flow rate prediction and uncertainty quantification. *Transportation Research Part C: Emerging Technologies*, *43*, 50–64.

Hadj-Salem, H., Blosseville, J. M., & Papageorgiou, M. (1990). Alinea. A local feedback control law for on-ramp metering; a real-life study. *IEE Conference Publication*, *320*, 194–198.

Han, Y., & Ahn, S. (2018). Stochastic modeling of breakdown at freeway merge bottleneck and traffic control method using connected automated vehicle. *Transportation Research Part B: Methodological*, *107*, 146–166.

Hegyi, A. (2004). *Model predictive control for integrating traffic control measures*. PhD Thesis. Delft University of Technology, Neatherland.

Hegyi, Andreas, De Schutter, B., & Hellendoorn, H. (2005). Model predictive control for optimal coordination of ramp metering and variable speed limits. *Transportation Research Part C: Emerging Technologies*, *13*(3), 185–209.

Heirung, T. A. N., Paulson, J. A., Leary, J. O., & Mesbah, A. (2018). Stochastic model predictive control — how does it work ? *Computers and Chemical Engineering*, *114*, 158–170.

Heshami, S., Kattan, L., Gong, Z., & Aalami, S. (2019). Deterministic and Stochastic Freeway Capacity Analysis Based on Weather Conditions. *Journal of Transportation Engineering Part A: Systems*, *145*(5).

Highway Capacity Manual (HCM). (2010). Transportation Rsearch Board, Washington, DC: National Research Council.

Historical weather and climate data. (2016).

https://climate.weather.gc.ca/historical_data/search_historic_data_e.html

Hollander, Y., & Prashker, J. N. (2006). The applicability of non-cooperative game theory in transport analysis. *Transportation*, *33*(5), 481–496.

Hu, X., & Sun, J. (2019). Trajectory optimization of connected and autonomous vehicles at a multilane freeway merging area. *Transportation Research Part C: Emerging Technologies*, *101*(February), 111–125.

Huang, W., Jia, W., Guo, J., Williams, B. M., Shi, G., Wei, Y., & Cao, J. (2018). Real-time prediction of seasonal heteroscedasticity in vehicular traffic flow series. *IEEE Transactions on Intelligent Transportation Systems*, *19*(10), 3170–3180.

Jin, L., Kurzhanskiy, A. A., & Amin, S. (2018). Throughput-improving control of highways facing stochastic perturbations. *ArXiv*, 1–15.

Kachroo, P., Ozbay, K., & Hobeika, A. G. (2001). Real-time travel time estimation using macroscopic traffic flow models. *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, 132–137.

Kang, K., & Rakha, H. A. (2017). Game theoretical approach to model decision making for merging maneuvers at freeway on-ramps. *Transportation Research Record*, *2623*(1), 19–28.

Karimi, A., Hegyi, A., Schutter, B. De, Hellendoom, H., & Middelham, F. (2004). Integration of Dynamic Route Guidance and Freeway Ramp Metering Using Model Predictive Control. *Proceeding of the 2004 American control conference,* Boston, Massachusetts.

Kawasaki, Y., Hara, Y., & Kuwahara, M. (2019). Traffic state estimation on a two-dimensional network by a state-space model. *Transportation Research Part C: Emerging Technologies*, *March*, 1–17.

Ke, R., Li, W., Cui, Z., & Wang, Y. (2020). Two-Stream Multi-Channel Convolutional Neural Network for Multi-Lane Traffic Speed Prediction Considering Traffic Volume Impact. *Transportation Research Record: Journal of the Transportation Research Board*, *Im*, 036119812091105.

Kerner, B. S., Rehborn, H., Haug, A., & Maiwald-Hiller, I. (2005). Tracking of congested traffic patterns on freeways in California. *Traffic Engineering and Control*, *46*(10), 380–385.

Kerner, Boris S. (2002). Empirical macroscopic features of spatial-temporal traffic patterns at highway bottlenecks. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, *65*(4), 30.

Kerner, Boris S. (2007). Control of spatiotemporal congested traffic patterns at highway bottlenecks. *IEEE Transactions on Intelligent Transportation Systems*, *8*(2), 308–320.

Kerner, Boris S. (2009). *Introduction to Modern Traffic Flow Theory and Control, The Long Road to Three-Phase Traffic Theory*. Springer-Verlag Berlin Heidelberg.

Kerner, Boris S. (2011). Optimum principle for a vehicular traffic network: Minimum probability of congestion. *Journal of Physics A: Mathematical and Theoretical*, *44*(9).

Kerner, Boris S. (2017). Breakdown in traffic networks: Fundamentals of transportation science. In *Breakdown in Traffic Networks: Fundamentals of Transportation Science*.

Kerner, Boris S., & Klenov, S. L. (2003). Microscopic theory of spatial-temporal congested traffic patterns at highway bottlenecks. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, *68*(3), 20.

Khondaker, B., & Kattan, L. (2015). Variable speed limit: an overview. *Transportation Letters*, *7*(5), 264–278.

Khoo, H. L. (2011). Dynamic penalty function approach for ramp metering with equity constraints. *Journal of King Saud University-Science*, 23(3), 273–279.

Kittelson, W. K., & Roess, R. P. (2000). *AfterHCM2000*. *1950*(01), 10–16.

Kong, D., & Guo, X. (2016). Analysis of vehicle headway distribution on multi-lane freeway considering car-truck interaction. *Advances in Mechanical Engineering*, *8*(4), 1–12.

Kotsialos, A., & Papageorgiou, M. (2004). *Efficiency and equity properties of freeway network-wide ramp metering with AMOC*. *12*, 401–420.

Krauss, S., Wagner, P., & Gawron, C. (1997). Metastable states in a microscopic model of traffic flow. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, *55*(5), 5597–5602.

Kumar, K., Parida, M., & Katiyar, V. K. (2013). Short Term Traffic Flow Prediction for a Non Urban Highway Using Artificial Neural Network. *Procedia - Social and Behavioral Sciences*, *104*, 755–764.

Kuo, C.-W., & Tang, M.-L. (2011). Relationship among service quality, corporate image, customer satisfaction and behaviroal intention for the elderly in high speed rail service. *Journal of Advanced Transportation*, *47*(June 2010), 512–525.

Laflamme, E. M. (2013). *Extreme Value Theory: Applications To Estimation of Stochastic Traffic Capacity and Statistical Downscaling of Precipitation Extremes*. PhD Thesis, University of New Hampshire, Durham, New Hampshire.

Lam, W. H. K., Shao, H., & Sumalee, A. (2008). Modeling impacts of adverse weather conditions on a road network with uncertainties in demand and supply. *Transportation Research Part B: Methodological*, *42*(10), 890–910.

Leclercq, L., Knoop, V. L., Marczak, F., & Hoogendoorn, S. P. (2016). Capacity drops at merges: New analytical investigations. *Transportation Research Part C: Emerging Technologies*, *62*, 171–181.

Levinson, D., & Zhang, L. (2006). Ramp meters on trial: Evidence from the Twin Cities metering holiday. *Transportation Research Part A: Policy and Practice*, *40*(10), 810–828.

Levinson, D., Zhang, L., Das, S., & Sheikh, A. (2002). Ramp meters on trial: evidence from the Twin Cities ramp meters shut-off. *2002 TRB Annual Meeting*, *1*, 1–35.

Li, D., Ranjitkar, P., & Ceder, A. (2014). Integrated approach combining ramp metering and variable speed limits to improve motorway performance. *Transportation Research Record*, *2470*, 86–94.

Li, M., Zhou, X., & Rouphail, N. M. (2017). Quantifying travel time variability at a single bottleneck based on stochastic capacity and demand distributions. *Journal of Intelligent Transportation Systems: Technology, Planning, and Operations*, *21*(2), 79–93.

Lin, L., Handley, J. C., Gu, Y., Zhu, L., Wen, X., & Sadek, A. W. (2018). Quantifying uncertainty in short-term traffic prediction and its application to optimal staffing plan development. *Transportation Research Part C: Emerging Technologies*, *92*(February), 323–348.

Lipp L., Corcoran, L. and Hickman, G. (1991). Benefits of central computer control for the Denver ramp metering system. *Transportation Research Rec*ord, No. 1320, Research Board, Washington, D.C.

Liu, X., Zhang, G., Kwan, C., Wang, Y., & Kemper, B. (2013). Simulation-based, scenario-driven integrated corridor management strategy analysis. *Transportation Research Record*, *2396*, 38–44.

Liu, Y., Lin, P. W., Lai, X., Chang, G. L., & Marquess, A. (2006). Developments and applications of simulation-based online travel time prediction system: Traveling to Ocean City, Maryland. *Transportation Research Record*, *1959*, 92–104.

Lu, X. Y., Varaiya, P., Horowitz, R., Su, D., & Shladover, S. E. (2011). Novel freeway traffic control with variable speed limit and coordinated ramp metering. *Transportation Research Record*, *2229*, 55–65.

Ma, D., Nakamura, H., & Asano, M. (2013). Lane-based breakdown identification at diverge sections for breakdown probability modeling. *Transportation Research Record*, *2395*, 83–92.

Ma, M. H., Yang, Q. F., Liang, S. D., & Li, Z. L. (2015). Integrated Variable Speed Limits Control and Ramp Metering for Bottleneck Regions on Freeway. *Mathematical Problems in Engineering*, *2015*.

Management, I. (2018). A Lane-Based Analysis of Stochastic Breakdown Phenomena on an Urban Expressway Section. *Asian Transport Studies*, *5*(1), 64–80.

Masher, D.P., Ross, D.W., Wong, P.J., Tuan, P.L., Zeidler, H.M., Petracek, S. (1975). Guidelines for Design and Operation of Ramp Control Systems. Stanford Research Institute, Menlo Park, California.

Mazaré, P., Bayen, A. M., & Hall, S. D. (2012). Trade-offs between inductive loops and GPS probe vehicles for travel time estimation : A Mobile Century case study. *Transportation Research Board 91st Annual Meeting, Washington, DC*, *61801*(217), 20.

McDonough, W. & Braungart, M. (2013). The upcycle: *Beyondsustainability-Designing for abundance*. New York: Charles Melcher

Meeker, W. Q., & Escobar, L. A. (1994). Maximum likelihood methods for fitting parametric statistical models. *Statistical methods for physical science*, 28, 211–244.

Meng, Q., & Ling, H. (2010). A Pareto-optimization approach for a fair ramp metering. *Transportation Research Part C*, *18*(4), 489–506.

Messick, David M. & Cook, K. S. (1983), *Equity Theory: Psychological and Sociological Perspectives.* New York: Praeger.

Messmer, A., & Papageorgiou, M. (1994). Automatic control methods applied to freeway network traffic. *Automatica*, *30*(4), 691–702.

Miska, M. P. (2007). Microscopic online simulation for real time traffic management. In *Transport and Planning : Vol. Doctoral*.

Modi, V., Kondyli, A., Washburn, S. S., & McLeod, D. S. (2014). Freeway capacity estimation method for planning applications. *Journal of Transportation Engineering*, *140*(9), 1–9.

Moorhead, G., & Griffin, R. W. (1997). *Organizational behavior (5th ed.).* Boston: Houghton Mifflin.

Muggeo, V. M. R. (2003). Estimating regression models with unknown break-points. *Statistics in Medicine*, *22*(19), 3055–3071.

Myers, K. A., & Tapley, B. D. (1976). Adaptive Sequential Estimation with Unknown Noise Statistics. *IEEE Transactions on Automatic Control*, *21*(4), 520–523.

Nagalur Subraveti, H. H. S., Knoop, V. L., & van Arem, B. (2019). First order multi-lane traffic flow model–an incentive based macroscopic model to represent lane change dynamics. *Transportmetrica B*, *7*(1), 1758–1779.

Nantes, A., Ngoduy, D., Bhaskar, A., Miska, M., & Chung, E. (2016). Real-time traffic state estimation in urban corridors from heterogeneous data. *Transportation Research Part C: Emerging Technologies*, *66*, 99–118.

Nash, J. F. J. (1953). Two-Person Cooperative Games Author (s): John Nash Published by : The Econometric Society Stable URL : http://www.jstor.org/stable/1906951 REFERENCES Linked references are available on JSTOR for this article : You may need to log in to JSTOR to access t. *Journal of Econometric Society*, *21*(1), 128–140.

Negenborn, R. R., De Schutter, B., & Hellendoorn, J. (2008). Multi-agent model predictive control for transportation networks: Serial versus parallel schemes. *Engineering Applications of Artificial Intelligence*, *21*(3), 353–366.

Newell, G. F. (1993). A simplified theory of kinematic waves in highway traffic, part II: Queueing at freeway bottlenecks. *Transportation Research Part B*, *27*(4), 289–303.

Oh, S., Byon, Y. J., Jang, K., & Yeo, H. (2015). Short-term Travel-time Prediction on Highway: A Review of the Data-driven Approach. *Transport Reviews*, *35*(1), 4–32.

Oh, S., Byon, Y. J., Jang, K., & Yeo, H. (2018). Short-term travel-time prediction on highway: A review on model-based approach. *KSCE Journal of Civil Engineering*, *22*(1), 298–310.

Ojeda. (2014). *Short-term multi-step ahead traffic forecasting*. 134.

Ojeda, L. L., Kibangou, A. Y., & de Wit, C. C. (2013). Adaptive Kalman filtering for multi-step ahead traffic flow prediction. *2013 American Control Conference*, 4724–4729.

Ojeda, L. L., Kibangou, A. Y., & Wit, C. C. De. (2013). *Online Dynamic Travel Time Prediction using Speed and Flow Measurements*. 4045–4050.

Ossenbruggen, P. J. (2016). Assessing freeway breakdown and recovery: A stochastic model. *Journal of Transportation Engineering*, *142*(7), 1–9.

Ozguven, E. E., & Ozbay, K. (2008). Nonparametric bayesian estimation of freeway capacity distribution from censored observations. *Transportation Research Record*, *2061*, 20–29.

Paesani, G. , Kerr, J., Perovich, P., and Khosravi, E. (1997). System wide adaptive ramp metering in southern California. ITS America 7th Annual Meeting.

Pan, T., Lam, W. H. K., Sumalee, A., & Zhong, R. (2019). Multiclass multilane model for freeway traffic mixed with connected automated vehicles and regular human-piloted vehicles. *Transportmetrica A: Transport Science*, *0*(0), 1–29.

Papadopoulou, S., Roncoli, C., Bekiaris-Liberis, N., Papamichail, I., & Papageorgiou, M. (2018). Microscopic simulation-based validation of a per-lane traffic state estimation scheme for highways with connected vehicles. *Transportation Research Part C: Emerging Technologies*, *86*(April 2017), 441–452.

Papageorgiou, M. (1995). An integrated control approach for traffic corridors. *Transportation Research Part C*, *3*(1), 19–30.

Papageorgiou, M., Diakaki, C., Dinopoulou, V., Kotsialos, A., & Wang, Y. (2003). Review of Road Traffic Control Strategies. *Proc. IEEE*, *91*(12), 2043–2067.

Papageorgiou, M., & Kotsialos, A. (2002). Freeway Ramp Metering: An Overview. *IEEE Transactions on Intelligent Transportation Systems*, *3*(4), 271–281.

Papamichail, I., Kotsialos, A., Margonis, I., & Papageorgiou, M. (2010). Coordinated ramp metering for freeway networks - A model-predictive hierarchical control approach. *Transportation Research Part C: Emerging Technologies*, *18*(3), 311–331.

Persaud, B., Yagar, S., Brownlee, R. (1998). Exploration of the breakdown phenomenon in freeway traffic." *Transportation Research Record,* 1634: 64–69.

Persaud, B, & Univer-, R. P. (n.d.). *Exploration of the Breakdown*. *98*, 64–69.

Persaud, B., Yagar, S., Tsui, D., & Look, H. (2001). Breakdown-related capacity for freeway with ramp metering. *Transportation Research Record*, *1748*, 110–115.

Pisarski, D., & Canudas-De-Wit, C. (2016). Nash Game-Based Distributed Control Design for Balancing Traffic Density over Freeway Networks. *IEEE Transactions on Control of Network Systems*, *3*(2), 149–161.

Polson, N. G., & Sokolov, V. O. (2017). Deep learning for short-term traffic flow prediction. *Transportation Research Part C: Emerging Technologies*, *79*, 1–17.

Polus, A., & Pollatschek, M. A. (2002). Stochastic nature of freeway capacity and its estimation. *Canadian Journal of Civil Engineering*, *29*(6), 842–852.

Portilla, C., Valencia, F., Lopez, J. D., Espinosa, J., Nüñez, A., & De Schutter, B. (2012). Non-linear model predictive control based on game theory for traffic control on highways. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, *4*(PART 1), 436–441.

Rakha, H., & Zhang, Y. (2006). Analytical procedures for estimating capacity of freeway weaving, merge, and diverge sections. *Journal of Transportation Engineering*, *132*(8), 618–628.

Raza, A., & Zhong, M. (2017). Hybrid lane-based short-term urban traffic speed forecasting: A genetic approach. *2017 4th International Conference on Transportation Information and Safety, ICTIS 2017 - Proceedings*, 271–279.

Rehborn, H., & Palmer, J. (2008). ASDA/FOTO based on Kerner’s Three-Phase traffic theory in North Rhine-Westphalia and its integration into vehicles. *IEEE Intelligent Vehicles Symposium, Proceedings*, 186–191.

Rempe, F., Huber, G., & Bogenberger, K. (2016). Spatio-Temporal Congestion Patterns in Urban

Traffic Networks. *Transportation Research Procedia*, *15*, 513–524.

Reiss, R. D. & Thomas, M. (2007). *Statistical analysis of extreme values: With applications to insurance, finance, hydrology and other fields*. 3rd edition, New York: Springer.

Ruppe, S., Junghans, M., Haberjahn, M., & Troppenz, C. (2012). Augmenting the Floating Car Data Approach by Dynamic Indirect Traffic Detection. *Procedia - Social and Behavioral Sciences*, *48*, 1525–1534.

Schmitt, M., & Lygeros, J. (2020). On convexity of the robust freeway network control problem in the presence of prediction and model uncertainty. *Transportation Research Part B: Methodological*, *134*, 167–190.

Schreckenberg, M., & Wahle, J. (2001). Towards a multi-agent system for on-line simulations based on real-world traffic data. *At-Automatisierungstechnik*, *49*(11), 485–492.

Seo, T., & Kusakabe, T. (2015). Probe vehicle-based traffic state estimation method with spacing information and conservation law. *Transportation Research Part C: Emerging Technologies*, *59*, 391–403.

Shaaban, K., Khan, M. A., & Hamila, R. (2016). Literature Review of Advancements in Adaptive Ramp Metering. *Procedia Computer Science*, *83*(Ant), 203–211.

Shiomi, Y., Taniguchi, T., Uno, N., Shimamoto, H., & Nakamura, T. (2015). Multilane first-order traffic flow model with endogenous representation of lane-flow equilibrium. *Transportation Research Part C: Emerging Technologies*, *59*, 198–215.

Smith, B. L., Byrne, K. G., Copperman, R. B., Hennessy, S. M., & Goodall, N. J. (2003). an Investigation Into the Impact of Rainfall. *Review Literature And Arts Of The Americas*.

Smith, M., Huang, W., Viti, F., Tampère, C. M. J., Lo, H. K., Harks, T., Poschwatta, T., Gehlot, H., Honnappa, H., Ukkusuri, S. V., Tzeng, H. Y., Siu, K. Y., Sun, J., Como, G., Lovisari, E., Savla, K., Kerner, B. S., Rastgoftar, H., Atkins, E., … Mahmassani, H. S. (2019). Network Performance Under System Optimal and User Equilibrium Dynamic Assignments: Implications for ATIS. *Physica A: Statistical Mechanics and Its Applications*, *1408*(1), 83–93.

Smulders, S. (1990). Control of freeway traffic flow by variable speed signs. *Transportation Research Part B*, *24*(2), 111–132.

Srivastava, A., & Geroliminis, N. (2013). Empirical observations of capacity drop in freeway merges with ramp control and integration in a first-order model. *Transportation Research Part C: Emerging Technologies*, *30*, 161–177.

Suh, J., & Yeo, H. (2016). An empirical study on the traffic state evolution and stop-and-go traffic development on freeways. *Transportmetrica A: Transport Science*, *12*(1), 80–97.

Sumalee, A., Zhong, R. X., Pan, T. L., & Szeto, W. Y. (2011). Stochastic cell transmission model (SCTM): A stochastic dynamic traffic model for traffic state surveillance and assignment. *Transportation Research Part B: Methodological*, *45*(3), 507–533.

Sumalee, Agachai, Pan, T., Zhong, R., Uno, N., & Indra-Payoong, N. (2013). Dynamic stochastic journey time estimation and reliability analysis using stochastic cell transmission model: Algorithm and case studies. *Transportation Research Part C: Emerging Technologies*, *35*, 263–285.

Sunderrajan, A., Viswanathan, V., Cai, W., & Knoll, A. (2016). Traffic state estimation using floating car data. *Procedia Computer Science*, *80*, 2008–2018.

Taale, H. (2009). Integrated Anticipatory Control of Road Networks. In *Aenorm* (Vol. 17, Issue

64).

Talebpour, A., Mahmassani, H. S., & Hamdar, S. H. (2015). Modeling Lane-Changing Behavior in a Connected Environment: A Game Theory Approach. *Transportation Research Procedia*, *7*, 420–440.

Tarnoff, P. J., Young, S. E., Crunkleton, J., & Nezamuddin, N. (n.d.). *Guide to Benchmarking Operations Performance Measures Preiminary Daft, Prepared for NCHRP Transportation Research Board of The National Academies*. *January 2008*.

Tian, J., Treiber, M., Ma, S., Jia, B., & Zhang, W. (2015). Microscopic driving theory with oscillatory congested states: Model and empirical verification. *Transportation Research Part B: Methodological*, *71*, 138–157.

Tian, Q., Huang, H., Yang, H., & Gao, Z. (2012). Efficiency and equity of ramp control and capacity allocation mechanisms in a freeway corridor. *Transportation Research Part C*, *20*(1), 126–143.

Toppen, A., & Wunderlich, K. (2003). Travel Time Data Collection for Measurement of Advanced Traveler Information Systems Accuracy. *Time*, *June*.

Trapp, R. (2016). A Local Non-restrictive Ramp Metering Strategy Based on Stochasticity of Capacity. *Transportation Research Procedia*, *15*, 594–606.

Treiber, M., Hennecke, A., & Helbing, D. (2000). Congested traffic states in empirical observations and microsopic simulations. *Physical Review-E,* 62(2), 1805–1824.

Treiber, M., & Kesting, A. (2018). The Intelligent Driver Model with stochasticity – New insights into traffic flow oscillations. *Transportation Research Part B: Methodological*, *117*, 613–623.

Valencia Arroyave, F. (2012). *Game theory based distributed model predictive control: an approach to large-scale systems control*. 128.

Valencia, F., López, J. D., Núñez, A., Portilla, C., Cortes, L. G., Espinosa, J. & De Schutter, B. (2015). Congestion Management in Motorways and Urban Networks Through a Bargaining-Game-Based Coordination Mechanism. *Game Theoretic Analysis of Congestion, Safety and Security*, Springer Series in Reliability Engineering.

van Erp, P. B. C., Knoop, V. L., & Hoogendoorn, S. P. (2018). Macroscopic traffic state estimation using relative flows from stationary and moving observers. *Transportation Research Part B: Methodological*, *114*, 281–299.

Van Hinsbergen, C. P. I. J., Schreiter, T., Zuurbier, F. S., Van Lint, J. W. C., & Van Zuylen, H. J. (2010). Fast traffic state estimation with the localized extended kalman filter. *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, 917–922.

Venkat, A. N., Rawlings, J. B., & Wright, S. J. (2005). Stability and optimality of distributed model predictive control. *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference, CDC-ECC '05*, *2005*, 6680–6685.

Vlahogianni, E. I., Golias, J. C., & Karlaftis, M. G. (2004). Short-term traffic forecasting: Overview of objectives and methods. *Transport Reviews*, *24*(5), 533–557.

Vlahogianni, E. I., Karlaftis, M. G., & Golias, J. C. (2014). Short-term traffic forecasting: Where we are and where we're going. *Transportation Research Part C: Emerging Technologies*, *43*, 3–19.

Wang, H., Rudy, K., Li, J., & Ni, D. (2010). Calculation of traffic flow breakdown probability to optimize link throughput. *Applied Mathematical Modelling*, *34*(11), 3376–3389.

Wang, X., Yin, D., & Qiu, T. Z. (2018). Applicability analysis of an extended METANET model in traffic-state prediction for congested freeway corridors. *Journal of Transportation Engineering Part A: Systems*, *144*(9), 1–11.

Wang, Y., Papageorgiou, M., & Messmer, A. (2008). Real-time freeway traffic state estimation based on extended Kalman filter: Adaptive capabilities and real data testing. *Transportation Research Part A: Policy and Practice*, *42*(10), 1340–1358.

Wegerle, D., Kerner, B. S., Schreckenberg, M., & Klenov, S. L. (2019). Prediction of moving bottleneck through the use of probe vehicles: a simulation approach in the framework of three-phase traffic theory. *Journal of Intelligent Transportation Systems: Technology, Planning, and Operations*, *24*(6), 598–616.

Work, D. B., Member, S., Tossavainen, O., Blandin, S., Member, A. M. B., Iwuchukwu, T., & Tracton, K. (2008). *An Ensemble Kalman Filtering Approach to Highway Traffic Estimation Using GPS Enabled Mobile Devices*. 5062–5068.

Wu, X., Michalopoulos, P., & Liu, H. X. (2010). Stochasticity of freeway operational capacity and chance-constrained ramp metering. *Transportation Research Part C: Emerging Technologies*, *18*(5), 741–756.

Wu, Y., Tan, H., Qin, L., Ran, B., & Jiang, Z. (2018). A hybrid deep learning based traffic flow prediction method and its understanding. *Transportation Research Part C: Emerging Technologies*, *90*(April 2017), 166–180.

Xia, J., Chen, M., Huang, W., Xia, J., Chen, M. E. I., & Huang, W. E. I. (2011). *A Multistep Corridor Travel-Time Prediction Method Using Presence-Type Vehicle Detector Data A Multistep Corridor Travel-Time Prediction Method Using Presence-Type Vehicle Detector*

Data. *2450*(May).

Xie, Y., Zhang, Y., & Ye, Z. (2007). Short-term traffic volume forecasting using Kalman filter with discrete wavelet decomposition. *Computer-Aided Civil and Infrastructure Engineering*, *22*(5), 326–334.

Yao, J., Rakha, H., Teng, H., Kwigizile, V., & Kaseko, M. (2009). Estimating highway capacity considering two-regime models. *Journal of Transportation Engineering*, 135(9), 670–676.

Yao, S., Knoop, V. L., & van Arem, B. (2017). Optimizing traffic flow efficiency by controlling lane changes: Collective, group, and user optima. *Transportation Research Record*, *2622*(1), 96–104.

Yeon, J., Hernandez, S., & Elefteriadou, L. (2009). Differences in freeway capacity by day of the week, time of day, and segment type. *Journal of Transportation Engineering*, *135*(7), 416–426.

Yin, Y., Liu, H., & Benouar, H. (2004). *A Note on Equity of Ramp Metering*. 497–502.

Yuan, K., Knoop, V. L., & Hoogendoorn, S. P. (2015). Capacity drop: Relationship between speed in congestion and the queue discharge rate. *Transportation Research Record*, 2491, 72–80.

Yuan, K., Knoop, V. L., Leclercq, L., & Hoogendoorn, S. P. (2017). Capacity drop: a comparison between stop-and-go wave and standing queue at lane-drop bottleneck. *Transportmetrica B*, *5*(2), 149–162.

Yuan, Y., Lint, J. W. C. Van, Wilson, R. E., Wageningen-kessels, F. Van, & Hoogendoorn, S. P. (2012). *Real-Time Lagrangian Traffic State Estimator for Freeways*. *13*(1), 59–70.

Zegeye, S. K., Schutter, B. De, Hellendoorn, H., & Breunesse, E. (2009). Reduction of travel times and traffic emissions using model predictive control. *2009 American Control Conference*, 5392–5397.

Zhang, G., & Wang, Y. (2013). Optimizing Coordinated Ramp Metering : A Preemptive Hierarchical Control Approach. *Computer-Aided Civil and Infrastructure Engineering*, *28*, 22–37.

Zhang, Lei, & Levinson, D. (2005). *Balancing Efficiency and Equity of Ramp Meters*. *131*(June), 477–481.

Zhang, Li, Gou, J., & Jin, M. (2012). Model of Integrated Corridor Traffic Optimization. *Transportation Research Record: Journal of the Transportation Research Board*, *2311*, 108–116.

Zhang, Y., Zhang, Y., & Haghani, A. (2014). A hybrid short-term traffic flow forecasting method based on spectral analysis and statistical volatility model. *Transportation Research Part C: Emerging Technologies*, *43*, 65–78.

Zheng, Z., Ahn, S., Chen, D., & Laval, J. (2011). Freeway traffic oscillations: Microscopic analysis of formations and propagations using Wavelet Transform. *Transportation Research Part B: Methodological*, *45* `(9), 1378–1388.

Zhong, R. X., Sumalee, A., Pan, T. L., & Lam, W. H. K. (2014). Optimal and robust strategies for freeway traffic management under demand and supply uncertainties: An overview and general theory. *Transportmetrica A: Transport Science*, *10*(10), 849–877.

Zhou, Y., Ahn, S., Chitturi, M., & Noyce, D. A. (2017). Rolling horizon stochastic optimal control strategy for ACC and CACC under uncertainty. *Transportation Research Part C*, *83*, 61–76.

APPENDICES

Appendix I

A possible parameterization to model segmented relationship between the response and the variable Z is to fit the terms:

$$\alpha Z + \beta(Z - \varphi)_+$$

where φ is the break-point and $(Z - \varphi)_+$=$(Z - \varphi)_+ \times I(Z > \varphi)$ being $I(A) = 1$ if A is true. α is the slope of the left line segment (for $Z \leq \varphi$), and β is the difference in slopes; thus, $(\alpha + \beta)$ is the slope of the right segment an if the break-point exist, $|\beta| > 0$. Note that the loge-likelihood is not differentiable at $Z = \varphi$.

The key in fitting segmented regression by means of the linearization is that relevant first order Taylor's expansion around $\varphi^{(0)}$ holds exactly, provided that $\varphi^{(0)}$ is the break-point:

$$(Z - \varphi)_+ = (Z - \varphi^{(0)})_+ + \left(\varphi - \varphi^{(0)}\right)(-1)I(Z > \varphi^{(0)})$$

where $(-1)I(Z > \varphi^{(0)})$ is the first derivative of $(Z - \varphi)_+$ assessed in $\varphi^{(0)}$. There fore the algorithm at each step s is:

1. Fix $\varphi^{(s)}$ and calculate $(Z - \varphi^{(s)})_+$ and $(-1)I\left(Z > \varphi^{(s)}\right)$.
2. Fit the model as: $\alpha Z + \beta(Z - \varphi^{(s)})_+ + \delta(-1)I(Z > \varphi^{(s)})$. Coefficient δ measures the difference between the two fitted straight lines (before and after $\varphi^{(s)}$ (s)) at $Z = \varphi^{(s)}$.
3. Improve the break-point estimate by ($\varphi^{(s+1)} = \frac{\delta}{\beta} + \varphi^{(s)}$).
4. Repeat the process until convergence.

Appendix II

PERMISSIONS <permissions@asce.org>

Wed 12/16/2020 11:57 AM

Dear Seiran,
Thank you for your inquiry. As an original author of an ASCE journal article or proceedings paper, you are permitted to reuse your own content (including figures and tables) for another ASCE or non-ASCE publication (including your thesis), provided it does not account for more than 25% of the new work.

A full credit line must be added to the material being reprinted. For reuse in non-ASCE publications, add the words "With permission from ASCE" to your source citation. For Intranet posting, add the following additional notice: "This material may be downloaded for personal use only. Any other use requires prior permission of the American Society of Civil Engineers. This material may be found at https://doi.org/10.1061/JTEPBS.0000232."

Each license is unique, covering only the terms and conditions specified in it. Even if you have obtained a license for certain ASCE copyrighted content, you will need to obtain another license if you plan to reuse that content outside the terms of the existing license. For example: If you already have a license to reuse a figure in a journal, you still need a new license to use the same figure in a magazine. You need a separate license for each edition.

For more information on how an author may reuse their own material, please view: http://ascelibrary.org/page/informationforasceauthorsreusingyourownmaterial

Sincerely,

Leslie Connelly
Manager, Publications Marketing
American Society of Civil Engineers
1801 Alexander Bell Drive
Reston, VA 20191

PERMISSIONS@asce.org

703-295-6169
Internet: www.asce.org/pubs | www.ascelibrary.org | http://ascelibrary.org/page/rightsrequests

Appendix III

Simulation program:

```
#----------------------NEW FILE-------------------------- sim/cmd/csv/main.go
package main
import (
"encoding/csv"
"flag"
"fmt"
"io"
"io/ioutil"
"log"
"os"
"path/filepath"
"sim"
"sim/common"
"sim/controller"
"sim/placer"
"sim/simulator"
"sort"
"strconv"
"strings"
)
const (
top = "a_top"
mid = "b_mid"
bot = "c_bot"
ramp = "d_rmp"
exit = "e_exit"
)
var rampStarts = make([]float32, 0)
var exitStarts = make([]float32, 0)
var exitProbs = make([]float32, 0)
var detectorDelta = float32(0)
var confPath string
var outPath string
var adjustDelta float32
func main() {
os.Remove("./out.log")
w, _ := os.OpenFile("./out.log", os.O_CREATE|os.O_RDWR, 0755)
log.SetOutput(w)
p1 := flag.String("p", "./", "the path for config.csv and poisson_rate.csv")
p2 := flag.String("o", "./", "the path for outputs")
flag.Parse()
if len(*p1) == 0 || len(*p2) == 0 {
log.Println(*p1, *p2)
flag.Usage()
os.Exit(0)
}
confPath = *p1
outPath = *p2
f1, err := os.Open(filepath.Join(confPath, "config1.csv"))
if err != nil {
```

```
log.Fatalf("Could not open csv file: %v", err)
}
defer f1.Close()
f2, err := os.Open(filepath.Join(confPath, "config2.csv"))
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f2.Close()
f3, err := os.Open(filepath.Join(confPath, "ramps.csv"))
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f3.Close()
r1 := csv.NewReader(f1)
_, err = r1.Read()
r2 := csv.NewReader(f2)
_, err = r2.Read()
r3 := csv.NewReader(f3)
_, err = r2.Read()
if err != nil {
log.Fatalf("Could not read csv file: %v", err)
}
sim1 := &sim.Config{}
sim2 := &sim.Config{}
for {
record, err := r1.Read()
if err == io.EOF {
break
}
if err != nil {
log.Fatal(err)
}
parseProperty(record, sim1)
}
for {
record, err := r2.Read()
if err == io.EOF {
break
}
if err != nil {
log.Fatal(err)
}
parseProperty(record, sim2)
}
for {
record, err := r3.Read()
if err == io.EOF {
break
}
if err != nil {
log.Fatal(err)
}
parseRamp(record)
}
adjustExitProbs()
sim1.OutDir = filepath.Join(outPath, "run1_report1")
```

```
sim1.OptimizeDir = filepath.Join(outPath, "run1_report3")
sim1.ExitPositions = exitStarts
sim1.ExitProbabilities = exitProbs
sim2.OutDir = filepath.Join(outPath, "run1_report2")
sim2.OptimizeDir = filepath.Join(outPath, "run1_report3")
sim2.ExitPositions = exitStarts
sim2.ExitProbabilities = exitProbs
run1(*sim1, *sim2)
sim1.OutDir = filepath.Join(outPath, "run2_report1")
sim1.OptimizeDir = filepath.Join(outPath, "run2_report3")
sim2.OutDir = filepath.Join(outPath, "run2_report2")
sim2.OptimizeDir = filepath.Join(outPath, "run2_report3")
run2(*sim1, *sim2)
}
func run1(conf1 sim.Config, conf2 sim.Config) {
log.Println("------------> Running the simulation with optimized light.")
toRemove := make([]string, 0)
pattern, _ := filepath.Glob(conf1.OutDir + "*")
toRemove = append(toRemove, pattern...)
pattern, _ = filepath.Glob(conf2.OutDir + "*")
toRemove = append(toRemove, pattern...)
pattern, _ = filepath.Glob(conf2.OptimizeDir + "*")
toRemove = append(toRemove, pattern...)
for _, dir := range toRemove {
os.RemoveAll(dir)
}
conf1.UseAlwaysGreen = false
conf2.UseAlwaysGreen = false
reports := initialSim(conf1, conf2)
sort.Strings(reports)
}
func run2(conf1 sim.Config, conf2 sim.Config) {
log.Println("------------> Running the simulation with always green light and the same seed.")
toRemove := make([]string, 0)
pattern, _ := filepath.Glob(conf1.OutDir + "*")
toRemove = append(toRemove, pattern...)
pattern, _ = filepath.Glob(conf2.OutDir + "*")
toRemove = append(toRemove, pattern...)
pattern, _ = filepath.Glob(conf2.OptimizeDir + "*")
toRemove = append(toRemove, pattern...)
for _, dir := range toRemove {
os.RemoveAll(dir)
}
conf1.UseAlwaysGreen = true
conf2.UseAlwaysGreen = true
sim.HasPresetSeeds = true
reports := initialSim(conf1, conf2)
sort.Strings(reports)
}
func initialSim(conf sim.Config, conf2 sim.Config) []string {
os.RemoveAll(conf.OutDir)
os.MkdirAll(conf.OutDir, 0755)
lInfo := getLaneCopy(conf)
setGenRates(lInfo, conf)
dInfo := getNormalDetectorsCopy(conf)
linkInfo := getLinkCopy(conf)
```

```
iCP := placer.NewNormalPlacer(conf)
for _, l := range lInfo {
log.Println(l.ID, l.Name, "Start:", l.Start, "End:", l.End)
}
s := simulator.NewInitialSim(lInfo, dInfo, linkInfo, iCP, conf)
s.SetOtherSimulator(flowPredictSim, conf2)
s.Simulate()
return nil
}
func flowPredictSim(probeInfoPaths []string, conf sim.Config, outDir string, simulationDuration float32)
*sim.OptimizerDecision {
conf.SimulationDuration = simulationDuration
os.RemoveAll(outDir)
os.MkdirAll(outDir, 0755)
lInfo := getLaneCopy(conf)
setExactForLanes(lInfo, probeInfoPaths, conf, conf.SnapshotIncrement)
dInfo := getNormalDetectorsCopy(conf)
baseReport1Path := filepath.Dir(probeInfoPaths[0])
setPredictDetector(dInfo, baseReport1Path, conf)
linkInfo := getLinkCopy(conf)
iCP := placer.NewRespectfulPlacer(conf)
s := simulator.NewFlowSim(lInfo, dInfo, linkInfo, iCP, conf)
decision := s.Simulate()
s.Save(outDir)
return decision
}
func setGenRates(lInfo []simulator.LaneInfo, conf sim.Config) {
ratesDir := filepath.Join(confPath, "rates")
rates, err := ioutil.ReadDir(ratesDir)
if err != nil {
log.Fatalf("Could not read dir info: [%v]", err)
}
for _, r := range rates {
idx := -1
rType := ""
if strings.HasPrefix(r.Name(), top) {
idx = 0
rType = r.Name()[len(top)+1 : strings.Index(r.Name(), ".")]
} else if strings.HasPrefix(r.Name(), mid) {
idx = 1
rType = r.Name()[len(mid)+1 : strings.Index(r.Name(), ".")]
} else if strings.HasPrefix(r.Name(), bot) {
idx = 2
rType = r.Name()[len(bot)+1 : strings.Index(r.Name(), ".")]
} else if strings.HasPrefix(r.Name(), ramp) {
rIdxLen := strings.Index(r.Name()[len(ramp):], "_")
rIdx, err := strconv.Atoi(r.Name()[len(ramp) : len(ramp)+rIdxLen])
if err != nil {
log.Fatalf("Incorrect ramp id: [%v]", err)
}
idx = getIthRampID(rIdx)
rType = r.Name()[len(ramp)+1+rIdxLen : strings.Index(r.Name(), ".")]
} else {
continue
}
speed := fmt.Sprintf("%f", conf.MaxFreeFlowSpeed)
```

```
if idx >= 3 {
speed = fmt.Sprintf("%f", conf.MaxFreeFlowSpeedOnRamp)
}
lInfo[idx].Gen = map[string]string{"name": rType, "speed": speed, "path": filepath.Join(ratesDir, r.Name()), "col":
"1"}
}
}
func setExactForLanes(lInfo []simulator.LaneInfo, probeInfoPaths []string, conf sim.Config, interval float32) {
for i := 0; i < len(lInfo); i++ {
lInfo[i].Gen = map[string]string{"name": "exact", "path": probeInfoPaths[i], "interval": fmt.Sprintf("%.1f",
interval)}
}
}
func setPredictDetector(dInfo []simulator.DetectorInfo, baseReport1Path string, conf sim.Config) {
for i := 0; i < len(dInfo); i++ {
dInfo[i].DType = "predict"
dInfo[i].Path = filepath.Join(baseReport1Path, fmt.Sprintf("report-detector-%d.csv", i))
}
}
// merging lanes visual:
// -\-----\--------------------Start----------------d0----------/------------/-------d1--|----\-----\--------------------Start-------------
---d6----------/------------/-------d7-----
// \-----\d5-------| /-----d4---------d2---------/d3----------/ | \-----\d11------| /-----d10--------d8---------/d9----------/
// /------/ controller | /------/ controller
// /------/ | /------/
func getNormalDetectorsCopy(conf sim.Config) []simulator.DetectorInfo {
dInfo := []simulator.DetectorInfo{}
id := 0
for i, start := range rampStarts {
startOfMerge := start + conf.NonMergingRampLength + conf.MergingRampLength
endOfMerge := start + conf.NonMergingRampLength + conf.MergingRampLength + conf.MergingRegionLength
startOfParallel := start + conf.NonMergingRampLength
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: startOfMerge - detectorDelta,
})
id++
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: endOfMerge + adjustDelta*detectorDelta,
})
id++
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: startOfMerge - detectorDelta,
})
id++
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: startOfMerge,
})
id++
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: startOfParallel - detectorDelta,
})
id++
// NOTE: we are assuming that # of exit ramps is the same as number of merging ramps
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: exitStarts[i] + conf.ExitLaneLength + detectorDelta,
})
id++
}
for i := 0; i < getExtraManualDetectors(conf); i++ {
dInfo = append(dInfo, simulator.DetectorInfo{
ID: id,
DType: "normal",
Pos: float32((i + 1) * 1000),
})
id++
}
return dInfo
}
func getExtraManualDetectors(conf sim.Config) int {
maxCount := int(conf.LaneLength / 1000)
realCount := maxCount
for i := 0; i < maxCount; i++ {
pos := float32((i + 1) * 1000)
if pos >= conf.LaneLength {
realCount = i + 1
break
}
}
return realCount
}
func getLinkCopy(conf sim.Config) []simulator.LinkInfo {
totalRampDetectors := len(rampStarts) * 6
linkInfo := make([]simulator.LinkInfo, totalRampDetectors)
for i := 0; i < totalRampDetectors; i++ {
linkInfo[i] = simulator.LinkInfo{
DetectorID: i,
PrevDetecIDLane: -1,
PrevDetecIDRamp: -1,
PrevDetecIDExit: -1,
}
if i%6 < 2 {
linkInfo[i].LanesID = []int{0, 1, 2}
} else if i%6 < 5 {
linkInfo[i].LanesID = []int{getIthRampID(i / 6)}
} else {
linkInfo[i].LanesID = []int{getIthExitID(i / 6)}
}
if i/6 > 0 && i%6 == 1 {
linkInfo[i].PrevDetecIDLane = i - 6
linkInfo[i].PrevDetecIDRamp = i + 2
```

```
linkInfo[i].PrevDetectIDRampForTravelTime = i + 3
linkInfo[i].PrevDetecIDExit = i + 4
}
}
firstExtraDetector := totalRampDetectors
for i := 0; i < getExtraManualDetectors(conf); i++ {
linkInfo = append(linkInfo, simulator.LinkInfo{
DetectorID: firstExtraDetector + i,
PrevDetecIDLane: firstExtraDetector + i - 1,
PrevDetecIDRamp: -1,
PrevDetecIDExit: -1,
LanesID: []int{0, 1, 2},
})
}
return linkInfo
}
func getLaneCopy(conf sim.Config) []simulator.LaneInfo {
lInfo := []simulator.LaneInfo{
{
ID: 0,
Start: 0,
End: conf.LaneLength,
Name: top,
Ramp: false,
ExitLane: false,
Controllers: make([]sim.Controller, 0),
},
{
ID: 1,
Start: 0,
End: conf.LaneLength,
Name: mid,
Ramp: false,
ExitLane: false,
Controllers: make([]sim.Controller, 0),
},
{
ID: 2,
Start: 0,
End: conf.LaneLength,
Name: bot,
Ramp: false,
ExitLane: false,
Controllers: make([]sim.Controller, 0),
},
}
for i := range rampStarts {
// Exit lanes appear first, therefore they have a smaller id
lInfo = append(lInfo,
simulator.LaneInfo{
ID: getIthExitID(i),
Start: exitStarts[i],
Name: fmt.Sprintf("%s%d", exit, i),
Ramp: false,
ExitLane: true,
Controllers: make([]sim.Controller, 0),
},
)
// Merging lanes have light controllers
controllerPos := rampStarts[i] + conf.NonMergingRampLength + conf.MergingRampLength
var ctrler sim.Controller
if conf.UseAlwaysGreen {
ctrler = controller.NewAlwaysLight(controllerPos, controller.GREEN)
} else {
fileNamePattern := "optimize-detect-%d-%d.csv"
// TODO these are hard coded for 3 sepcific ramp
optimizerInfoPath := "Invalid"
switch i {
case 0:
log.Println("Skip optimizer for ramp 0")
case 1:
optimizerInfoPath = filepath.Join(conf.OptimizeDir, fmt.Sprintf(fileNamePattern, 7, 10))
case 2:
optimizerInfoPath = filepath.Join(conf.OptimizeDir, fmt.Sprintf(fileNamePattern, 13, 16))
case 3:
optimizerInfoPath = filepath.Join(conf.OptimizeDir, fmt.Sprintf(fileNamePattern, 19, 22))
}
ctrler = controller.NewOptimizedLight(controllerPos, optimizerInfoPath, 60, conf.TimeStep,
conf.TimeForGreenLight) // TODO hardcoded value 60
}
lInfo = append(lInfo,
simulator.LaneInfo{
ID: getIthRampID(i),
Start: rampStarts[i],
Name: fmt.Sprintf("%s%d", ramp, i),
Ramp: true,
ExitLane: false,
Controllers: []sim.Controller{ctrler},
},
)
}
return lInfo
}
func parseRamp(cells []string) {
if cells[0] == "AdjustDelta" {
adjustDelta = common.ToFloat(cells[1])
} else if cells[0] == "DetectorDelta" {
detectorDelta = common.ToFloat(cells[1])
} else {
if strings.Contains(cells[0], "RampStart") {
rampStarts = append(rampStarts, common.ToFloat(cells[1]))
} else if strings.Contains(cells[0], "ExitStart") {
ePosProb := strings.Split(cells[1], "-")
pos := common.ToFloat(ePosProb[0])
prob := common.ToFloat(ePosProb[1])
exitStarts = append(exitStarts, pos)
exitProbs = append(exitProbs, prob)
}
}
}
func adjustExitProbs() {
adjusted := make([]float32, len(exitProbs))
```

```
for i := 0; i < len(exitProbs); i++ {
adjusted[i] = exitProbs[i]
}
log.Printf("Exit probs: %.2f %.2f %.2f", adjusted[0], adjusted[1], adjusted[2])
exitProbs = adjusted
}
func getIthRampID(i int) int {
return 3 + 1 + i*2
}
func getIthExitID(i int) int {
return 3 + i*2
}
func parseProperty(cells []string, sim *sim.Config) bool {
switch cells[0] {
case "TimeStep":
sim.TimeStep = common.ToFloat(cells[1])
sim.LambdaOn = 0.75 * sim.TimeStep
case "SamplingDuration":
sim.SamplingDuration = common.ToFloat(cells[1])
case "LookAheadDistance":
sim.LookAheadDistance = common.ToFloat(cells[1])
case "Sigma1":
sim.Sigma1 = common.ToFloat(cells[1])
case "Sigma2":
sim.Sigma2 = common.ToFloat(cells[1])
case "Sigma":
sim.Sigma = common.ToFloat(cells[1])
case "K":
sim.K = common.ToFloat(cells[1])
case "Beta":
sim.Beta = common.ToFloat(cells[1])
case "MaxAcceleration":
sim.MaxAcceleration = common.ToFloat(cells[1])
case "MaxDeceleration":
sim.MaxDeceleration = common.ToFloat(cells[1])
case "DeltaVR1":
sim.DeltaVR1 = common.ToFloat(cells[1])
case "DeltaVR2":
sim.DeltaVR2 = common.ToFloat(cells[1])
case "Length":
sim.Length = common.ToFloat(cells[1])
case "MaxFreeFlowSpeed":
sim.MaxFreeFlowSpeed = common.ToFloat(cells[1])
case "MaxFreeFlowSpeedOnRamp":
sim.MaxFreeFlowSpeedOnRamp = common.ToFloat(cells[1])
case "DecelerationProb":
sim.DecelerationProb = common.ToFloat(cells[1])
case "AccelerationProb":
sim.AccelerationProb = common.ToFloat(cells[1])
case "LaneChangeProb":
sim.LaneChangeProb = common.ToFloat(cells[1])
case "MergeProb":
sim.MergeProb = common.ToFloat(cells[1])
case "SimulationDuration":
sim.SimulationDuration = common.ToFloat(cells[1])
case "LaneLength":
```

```
sim.LaneLength = common.ToFloat(cells[1])
case "SafeDistance":
sim.SafeDistance = common.ToFloat(cells[1])
case "DetectorRange":
sim.DetectorRange = common.ToFloat(cells[1])
case "ProbeProbability":
sim.ProbeProbability = common.ToFloat(cells[1])
case "SnapshotIncrement":
sim.SnapshotIncrement = common.ToFloat(cells[1])
case "MergingRegionLength":
sim.MergingRegionLength = common.ToFloat(cells[1])
case "MergingRampLength":
sim.MergingRampLength = common.ToFloat(cells[1])
case "NonMergingRampLength":
sim.NonMergingRampLength = common.ToFloat(cells[1])
case "KFN":
sim.KFN = common.ToInt(cells[1])
case "AverageCarLength":
sim.AverageCarLength = common.ToFloat(cells[1])
case "VSyn":
sim.VSyn = common.ToFloat(cells[1])
case "VJam":
sim.VJam = common.ToFloat(cells[1])
case "QJam_1":
sim.QJam1 = common.ToFloat(cells[1])
case "QMinT_1":
sim.QMinT1 = common.ToFloat(cells[1])
case "ROMaxT_1":
sim.ROMaxT1 = common.ToFloat(cells[1])
case "QJam_3":
sim.QJam3 = common.ToFloat(cells[1])
case "QMinT_3":
sim.QMinT3 = common.ToFloat(cells[1])
case "ROMaxT_3":
sim.ROMaxT3 = common.ToFloat(cells[1])
case "PatternMU":
sim.PatternMU = common.ToFloat(cells[1])
case "Tave":
sim.Tave = common.ToFloat(cells[1])
case "ExitDistance":
sim.ExitDistance = common.ToFloat(cells[1])
case "ExitLaneLength":
sim.ExitLaneLength = common.ToFloat(cells[1])
case "LightInterval":
sim.LightInterval = common.ToFloat(cells[1])
case "SimpleQtt":
sim.SimpleQtt = float64(common.ToFloat(cells[1]))
case "SimpleRtt":
sim.SimpleRtt = float64(common.ToFloat(cells[1]))
case "StepAheadCount":
sim.StepAheadCount = common.ToInt(cells[1])
case "Alpha1":
sim.Alpha1 = float64(common.ToInt(cells[1]))
case "Alpha2":
sim.Alpha2 = float64(common.ToInt(cells[1]))
case "Alpha3":
```

```
sim.Alpha3 = float64(common.ToInt(cells[1]))
case "Beta1":
sim.Beta1 = float64(common.ToInt(cells[1]))
case "Beta2":
sim.Beta2 = float64(common.ToInt(cells[1]))
case "Beta3":
sim.Beta3 = float64(common.ToInt(cells[1]))
case "W1":
sim.W1 = float64(common.ToInt(cells[1]))
case "W2":
sim.W2 = float64(common.ToInt(cells[1]))
case "W3":
sim.W3 = float64(common.ToInt(cells[1]))
case "Alpha":
sim.Alpha = float64(common.ToInt(cells[1]))
case "QueueRampMax1":
sim.QueueRampMax1 = float64(common.ToInt(cells[1]))
case "QueueRampMax2":
sim.QueueRampMax2 = float64(common.ToInt(cells[1]))
case "QueueRampMax3":
sim.QueueRampMax2 = float64(common.ToInt(cells[1]))
case "TimeForGreenLight":
sim.TimeForGreenLight = float64(common.ToInt(cells[1]))
case "RunSim2":
sim.RunSim2 = cells[1] == "yes"
case "PenalizedDeltaFlowOn":
sim.PenalizedDeltaFlowOn = int(common.ToInt(cells[1]))
default:
return false
}
return true
}
#----------------------NEW FILE-------------------------- sim/lane/lane_test.go
package lane
import (
"log"
"sim"
"sim/car"
"sim/change"
"testing"
)
func state(position float32, speed float32) *sim.CarState {
return &sim.CarState{
Position: position,
Speed: speed,
}
}
func TestLaneSingleNormal(t *testing.T) {
config := sim.DefaultConfig()
cars := []*sim.Car{
car.New(state(10, 30), 0, false, false, 0, -1, config),
car.New(state(11, 30), 1, false, false, 0, -1, config),
}
start := float32(0)
end := float32(100)
```

```
shoulder := NewNormalLane(NormalInfo{-1, car.NewCarRepo(), start, end, change.NewNever(),
change.NewNever(), "shoulder"}, car.New, car.NextState, config)
l := NewNormalLane(NormalInfo{0, car.NewCarRepo(), start, end, change.NewNever(), change.NewNever(),
"normal"}, car.New, car.NextState, config)
l.Adjacent(shoulder, shoulder)
for _, c := range cars {
l.Add(c)
}
l.Simulate()
log.Println(l)
l.Simulate()
log.Println(l)
}
func TestRamp(t *testing.T) {
config := sim.DefaultConfig()
cars := []*sim.Car{
car.New(state(10, 30), 0, false, false, 0, -1, config),
car.New(state(11, 30), 1, false, false, 0, -1, config),
}
start := float32(0)
mergeStart := float32(100)
mergeLength := float32(50)
end := float32(500)
shoulder := NewNormalLane(NormalInfo{-1, car.NewCarRepo(), start, end, change.NewNever(),
change.NewNever(), "shoulder"}, car.New, car.NextState, config)
normal := NewNormalLane(NormalInfo{0, car.NewCarRepo(), start, end, change.NewNever(),
change.NewNever(), "normal"}, car.New, car.NextState, config)
normal.Adjacent(shoulder, shoulder)
ramp := NewRampLane(RampInfo{1, car.NewCarRepo(), start, mergeStart, mergeLength, change.NewNever(),
change.NewNever(), "ramp"}, car.New, car.NextState, config)
ramp.Adjacent(normal, shoulder)
for _, c := range cars {
ramp.Add(c)
}
ramp.Simulate()
log.Println(ramp)
ramp.Simulate()
log.Println(ramp)
}
#-----------------------NEW FILE-------------------------- sim/lane/lane.go
package lane
import (
"bytes"
"fmt"
"sim"
)
type lane struct {
repo sim.CarRepo
leftChange sim.LaneChanger
rightChange sim.LaneChanger
neverChange sim.LaneChanger
left []sim.Lane
right []sim.Lane
start float32
mergeStart float32
end float32
```

```
id int
isRamp bool
newCar sim.NewCar
computeNext sim.ComputeNext
conf sim.Config
name string
n int
timeStep float32
moves map[*sim.Car]sim.Lane
controllers []sim.Controller
}
func (l *lane) String() string {
var buf []byte
b := bytes.NewBuffer(buf)
b.WriteString(l.name + ": ")
cars := make([]*sim.Car, 0, l.Leng())
for i := 0; ; i++ {
if c, ok := l.Get(i); ok {
cars = append(cars, c)
} else {
break
}
}
for i := 0; i < len(cars); i++ {
idx := len(cars) - 1 - i
b.WriteString(fmt.Sprintf("%s ", cars[idx]))
}
return b.String()
}
// NormalInfo holds the information necessary for creating a normal lane.
type NormalInfo struct {
ID int
Repo sim.CarRepo
Start float32
End float32
LChanger sim.LaneChanger
RChanger sim.LaneChanger
Name string
}
// RampInfo holds the information necessary for creating a ramp.
type RampInfo struct {
ID int
Repo sim.CarRepo
Start float32
MergeStart float32
MergeLength float32
LChanger sim.LaneChanger
Never sim.LaneChanger
Name string
}
// ExitInfo holds the information necessary for creating an exit ramp.
type ExitInfo struct {
ID int
Repo sim.CarRepo
Start float32
Length float32
```

```
LChanger sim.LaneChanger
Never sim.LaneChanger
Name string
}
type lightState struct {
isRed bool
pos float32
}
// NewNormalLane creates a non-ramp lane.
func NewNormalLane(inf NormalInfo, newCar sim.NewCar, computeNext sim.ComputeNext, controllers
[]sim.Controller, conf sim.Config) sim.Lane {
return &lane{moves: make(map[*sim.Car]sim.Lane, 0), controllers: controllers, repo: inf.Repo, start: inf.Start,
mergeStart: -1, end: inf.End, id: inf.ID, rightChange: inf.RChanger,
}
// NewRampLane creates a ramp lane.
func NewRampLane(inf RampInfo, newCar sim.NewCar, computeNext sim.ComputeNext, controllers
[]sim.Controller, conf sim.Config) sim.Lane {
end := inf.MergeStart + inf.MergeLength
return &lane{moves: make(map[*sim.Car]sim.Lane, 0), controllers: controllers, repo: inf.Repo, start: inf.Start,
mergeStart: inf.MergeStart, end: end, id: inf.ID, leftChange: inf.LChanger
}
// NewExitLane creates a ramp lane.
func NewExitLane(inf ExitInfo, newCar sim.NewCar, computeNext sim.ComputeNext, controllers []sim.Controller,
conf sim.Config) sim.Lane {
end := inf.Start + 3*conf.ExitLaneLength
return &lane{moves: make(map[*sim.Car]sim.Lane, 0), controllers: controllers, repo: inf.Repo, start: inf.Start,
mergeStart: -1, end: end, id: inf.ID, rightChange: inf.Never, leftChange
}
// Adjacent sets the adjacent lanes
func (l *lane) Adjacent(left []sim.Lane, right []sim.Lane) {
l.left = left
l.right = right
}
// Start returns the starting position of a lane
func (l *lane) Start() float32 {
return l.start
}
// End returns the end position of a lane
func (l *lane) End() float32 {
return l.end
}
// Add adds a car to the car repository
func (l *lane) Place(c *sim.CarState, id int) {
existing, ok := l.repo.GetByID(id)
if !ok {
now := float32(l.n) * l.timeStep
nc := l.newCar(c, id, l.isRamp, now, l.End(), l.conf)
l.repo.Add(nc)
} else {
existing.Next = c
curr := *c
prev := *c
existing.Curr = &curr
existing.Prev = &prev
l.repo.Sort()
}
```

```
}
func (l *lane) Add(c *sim.Car) {
l.repo.Add(c)
}
func (l *lane) Sort() {
l.repo.Sort()
}
// Remove removes a car to the car repository
func (l *lane) Remove(c *sim.Car) {
l.repo.Remove(c)
}
// After returns the first car after a position
func (l *lane) After(pos float32, excludingID int) *sim.Car {
return l.repo.After(pos, excludingID)
}
// Before returns the last car before a position
func (l *lane) Before(pos float32) *sim.Car {
return l.repo.Before(pos)
}
func (l *lane) AfterInNext(pos float32, excludingID int) *sim.Car {
return l.repo.AfterInNext(pos, excludingID)
}
// Before returns the last car before a position
func (l *lane) BeforeInNext(pos float32) *sim.Car {
return l.repo.BeforeInNext(pos)
}
// GetBetween returns all the cars between two points
func (l *lane) GetBetween(start float32, end float32) []*sim.Car {
return l.repo.GetBetween(start, end)
}
func (l *lane) Clean() []*sim.Car {
removeCars := make([]*sim.Car, 0, 1000000)
if !l.IsRamp() {
for i := 0; ; i++ {
if me, ok := l.Get(i); ok {
if me.Prev.Position >= l.end || me.Curr.Position >= l.end || me.Next.Position >= l.end {
removeCars = append(removeCars, me)
} else {
break
}
} else {
break
}
}
//for _, c := range removeCars {
for i := range removeCars {
l.Remove(removeCars[len(removeCars)-i-1])
}
}
return removeCars
}
func (l *lane) Name() string {
return l.name
}
// AllProbes returns all the cars that are marked as probe cars
func (l *lane) AllProbes() []*sim.Car {
```

```
return l.repo.AllProbes()
}
func (l *lane) Leng() int {
return l.repo.Leng()
}
// Get returns car at an index. Returns and error in case index is out of range.
func (l *lane) Get(idx int) (*sim.Car, bool) {
return l.repo.Get(idx)
}
func (l *lane) GetByID(id int) (*sim.Car, bool) {
return l.repo.GetByID(id)
}
func (l *lane) GetFirst() (*sim.Car, bool) {
return l.repo.GetFirst()
}
// move changes a car location from the current lane to the target lane.
func (l *lane) Move() map[*sim.Car]sim.Lane {
for c, target := range l.moves {
l.repo.Remove(c)
target.Add(c)
}
defer func() { l.moves = make(map[*sim.Car]sim.Lane, 0) }()
return l.moves
}
// IsRamp returns true if the lane is a ramp.
func (l *lane) IsRamp() bool {
return l.isRamp
}
func (l *lane) Simulate() error {
l.repo.Sort()
l.n++
lightStates := make([]lightState, len(l.controllers))
for i, controller := range l.controllers {
lightStates[i].isRed = controller.IsRed()
lightStates[i].pos = controller.Position()
}
nextLightIdx := len(lightStates) - 1
for i := 0; ; i++ {
if me, ok := l.Get(i); ok {
pos := me.Curr.Position
preced := l.After(pos, me.ID) // need to increase pos by a small amount, otherwise it would return itself
rPreced := l.right[0].AfterInNext(pos, sim.InvalidID)
rTrail := l.right[0].BeforeInNext(pos)
right := sim.ChangePack{
LaneChanger: l.rightChange,
Preceding: sim.TimePack{
Curr: rPreced.Next, // Since we simulate from left to right, by this point the right lane has not shifted next to current
Prev: rPreced.Curr,
},
Trailing: sim.TimePack{
Curr: rTrail.Next,
Prev: rTrail.Curr,
},
}
lPreced := l.left[0].After(pos, sim.InvalidID)
lTrail := l.left[0].Before(pos)
lc := l.getLeftChanger(me.Curr.Position)
left := sim.ChangePack{
LaneChanger: lc,
Preceding: sim.TimePack{
Curr: lPreced.Curr,
Prev: lPreced.Prev,
},
Trailing: sim.TimePack{
Curr: lTrail.Curr,
Prev: lTrail.Prev,
},
}
isOnRampMerging := l.isRamp && me.Curr.Position > l.mergeStart
isApproachingRedLight := false
if nextLightIdx >= 0 && pos < lightStates[nextLightIdx].pos {
if lightStates[nextLightIdx].isRed {
isApproachingRedLight = true
}
nextLightIdx--
}
direction, err := l.computeNext(me, preced, left, right, isApproachingRedLight, isOnRampMerging)
if err != nil {
return err
}
if direction == sim.ChangeLeft {
l.moves[me] = l.left[0]
} else if direction == sim.ChangeRight {
l.moves[me] = l.right[0]
} else if direction == sim.ExitLane {
if len(l.right) == 1 {
// behave as changeRight
l.moves[me] = l.right[0]
} else {
// behave as exit
for _, rl := range l.right {
if pos >= rl.Start() && pos <= rl.Start()+l.conf.ExitLaneLength {
l.moves[me] = rl
}
}
}
}
} else {
return nil
}
}
}
func (l *lane) getLeftChanger(position float32) sim.LaneChanger {
if l.isRamp && position < l.mergeStart {
return l.neverChange
}
return l.leftChange
}
#-----------------------NEW FILE--------------------------- sim/car/car_repo_test.go
package car
import (
"sim"
```

```
"testing"
)
func genCar(pos float32, probe bool, id int) *sim.Car {
return &sim.Car{
ID: id,
Curr: &sim.CarState{
Position: pos,
Speed: -1000,
SpaceGap: -1000,
VSafeN: -1000,
AccState: 0,
IsProbe: probe,
},
Prev: &sim.CarState{},
Next: &sim.CarState{},
Internal: &sim.CarInternalState{},
Extra: &sim.CarExtraInfo{},
}
}
func TestAddNormal(t *testing.T) {
c := genCar(10, true, 1)
rep := NewCarRepo()
r := rep.(*repo)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c)
if len(r.cars) != 1 {
t.Errorf("Expected to see length = 1, got length = %d", len(r.cars))
}
}
func TestAddExtend(t *testing.T) {
c1 := genCar(10, true, 1)
c2 := genCar(10, true, 2)
c3 := genCar(10, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
if cap(r.cars) != 2 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
if len(r.cars) != 3 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
if cap(r.cars) != 4 {
t.Errorf("Expected to see cap = 4, got cap = %d", cap(r.cars))
}
}
func TestAddSort(t *testing.T) {
```

```
c1 := genCar(10, true, 1)
c2 := genCar(20, true, 2)
c3 := genCar(5, true, 3)
c4 := genCar(30, true, 4)
c5 := genCar(6, true, 5)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
rep.Add(c4)
rep.Add(c5)
if len(r.cars) != 5 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
if r.cars[0].ID != 4 {
t.Errorf("Error in sorting cars. Expected id %d at position 0, got ID %d", 4, r.cars[0].ID)
}
if r.cars[1].ID != 2 {
t.Errorf("Error in sorting cars. Expected id %d at position 1, got ID %d", 2, r.cars[1].ID)
}
if r.cars[2].ID != 1 {
t.Errorf("Error in sorting cars. Expected id %d at position 2, got ID %d", 1, r.cars[2].ID)
}
if r.cars[3].ID != 5 {
t.Errorf("Error in sorting cars. Expected id %d at position 3, got ID %d", 5, r.cars[3].ID)
}
if r.cars[4].ID != 3 {
t.Errorf("Error in sorting cars. Expected id %d at position 4, got ID %d", 3, r.cars[4].ID)
}
}
// 2018/04/15 21:01:08 c_bot: [6:30.0@1440.0] [7:30.0@540.0] [3:30.0@1170.0] [2:30.0@2070.0]
func TestAddBugSort(t *testing.T) {
c1 := genCar(540, true, 7)
c2 := genCar(1170, true, 3)
c3 := genCar(2070, true, 2)
c4 := genCar(1440, true, 6)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
rep.Add(c4)
if len(r.cars) != 4 {
t.Errorf("Expected to see length = 4, got length = %d", len(r.cars))
}
```

```
if r.cars[0].ID != 2 {
t.Errorf("Error in sorting cars. Expected id %d at position 0, got ID %d", 2, r.cars[0].ID)
}
if r.cars[1].ID != 6 {
t.Errorf("Error in sorting cars. Expected id %d at position 1, got ID %d", 6, r.cars[1].ID)
}
if r.cars[2].ID != 3 {
t.Errorf("Error in sorting cars. Expected id %d at position 2, got ID %d", 3, r.cars[2].ID)
}
if r.cars[3].ID != 7 {
t.Errorf("Error in sorting cars. Expected id %d at position 3, got ID %d", 7, r.cars[3].ID)
}
}
func TestRemove(t *testing.T) {
c1 := genCar(10, true, 1)
c2 := genCar(20, true, 2)
c3 := genCar(5, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
if len(r.cars) != 3 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
rep.Remove(c1)
if len(r.cars) != 2 {
t.Errorf("Expected to see length = 2, got length = %d", len(r.cars))
}
if r.cars[0].ID != 2 {
t.Errorf("Error in sorting cars. Expected id %d at position 1, got ID %d", 2, r.cars[1].ID)
}
if r.cars[1].ID != 3 {
t.Errorf("Error in sorting cars. Expected id %d at position 0, got ID %d", 3, r.cars[0].ID)
}
}
func TestAfter(t *testing.T) {
c1 := genCar(10, true, 1)
c2 := genCar(20, true, 2)
c3 := genCar(5, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
if len(r.cars) != 3 {
```

```
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
tests := []struct {
pos float32
id int
}{
{
pos: 0,
id: 3,
},
{
pos: 5,
id: 3,
},
{
pos: 6,
id: 1,
},
{
pos: 30,
id: -1, // plus infinity car
},
}
for i, test := range tests {
a := rep.After(test.pos, -1)
if test.id != a.ID {
t.Errorf("Error in tast case %d: expected id %d, got %d", i, test.id, a.ID)
}
}
}
func TestAfterEmpty(t *testing.T) {
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
a := rep.After(0, -3)
if -1 != a.ID {
t.Errorf("Error expected plus inf, got %d", a.ID)
}
}
func TestBefore(t *testing.T) {
c1 := genCar(10, true, 1)
c2 := genCar(20, true, 2)
c3 := genCar(5, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
if len(r.cars) != 3 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
tests := []struct {
pos float32
id int
}{
{
pos: 0,
id: -2,
},
{
pos: 5,
id: 3,
},
{
pos: 10,
id: 3,
},
{
pos: 11,
id: 1,
},
{
pos: 6,
id: 3,
},
{
pos: 30,
id: 2,
},
}
for i, test := range tests {
a := rep.Before(test.pos)
if test.id != a.ID {
t.Errorf("Error in tast case %d: expected id %d, got %d", i, test.id, a.ID)
}
}
}
func TestBeforeEmpty(t *testing.T) {
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
a := rep.Before(10)
if -2 != a.ID {
t.Errorf("Error expected minues inf, got %d", a.ID)
}
}
func TestBetween(t *testing.T) {
c := make([]*sim.Car, 3)
c[0] = genCar(10, true, 1)
```

```
c[1] = genCar(20, true, 2)
c[2] = genCar(5, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c[0])
rep.Add(c[1])
rep.Add(c[2])
if len(r.cars) != 3 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
tests := []struct {
start float32
end float32
ids []int
}{
{
start: 0,
end: 30,
ids: []int{2, 1, 3},
},
{
start: 0,
end: 1,
ids: []int{},
},
{
start: 30,
end: 40,
ids: []int{},
},
{
start: 5,
end: 9,
ids: []int{3},
},
}
for _, test := range tests {
ids := rep.GetBetween(test.start, test.end)
for i, id := range ids {
if id.ID != test.ids[i] {
t.Errorf("Error in tast case %d: expected id %d, got %d", i, test.ids[i], id.ID)
}
}
}
}
func TestBetweenEmpty(t *testing.T) {
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
```

```
t.Errorf("Error in test setup")
}
ids := rep.GetBetween(10, 100)
if len(ids) != 0 {
t.Errorf("Did not expect any cars, got %v", ids)
}
}
func TestGet(t *testing.T) {
c1 := genCar(10, true, 1)
c2 := genCar(20, true, 2)
c3 := genCar(5, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
if len(r.cars) != 3 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
if c, ok := rep.Get(0); ok {
if c != r.cars[0] {
t.Errorf("Got incorrect car at index 0. Expected car ID %d, got ID %d.", r.cars[0].ID, c.ID)
}
} else {
t.Errorf("Got index out of range for 0")
}
if c, ok := rep.Get(2); ok {
if c != r.cars[2] {
t.Errorf("Got incorrect car at index 2. Expected car ID %d, got ID %d.", r.cars[2].ID, c.ID)
}
} else {
t.Errorf("Got index out of range for 2")
}
if _, ok := rep.Get(3); ok {
t.Errorf("Expected to get index out of range, did not get any error!")
}
}
func TestProbe(t *testing.T) {
c1 := genCar(10, false, 1)
c2 := genCar(20, true, 2)
c3 := genCar(5, true, 3)
rep := NewCarRepo()
r := rep.(*repo)
r.maxLen = 2
r.cars = make(sortedCars, 0, r.maxLen)
if len(r.cars) != 0 {
t.Errorf("Error in test setup")
}
rep.Add(c1)
rep.Add(c2)
rep.Add(c3)
```

```
if len(r.cars) != 3 {
t.Errorf("Expected to see length = 3, got length = %d", len(r.cars))
}
probes := rep.AllProbes()
if len(probes) != 2 {
t.Errorf("Did not return all probes, expected 2, got [%v]", probes)
}
for _, p := range probes {
if !p.Curr.IsProbe {
t.Errorf("Returned a car that is not a probe car %v", p)
}
}
}
#----------------------NEW FILE-------------------------- sim/car/car_test.go
package car
import (
"log"
"sim"
"testing"
)
func genState(position float32) *sim.CarState {
return &sim.CarState{
Position: position,
Speed: 30,
SpaceGap: 7,
AccState: 0,
IsProbe: false,
}
}
type never struct {
}
func (n *never) Can(current sim.TimePack, precedingInSame sim.TimePack, precedingInTarget sim.TimePack,
trailingInTarget sim.TimePack) (float32, float32, bool) {
return -1, -1, false
}
func (n *never) Want(probability float32) bool {
return false
}
func TestAll(t *testing.T) {
config := sim.DefaultConfig()
cCar := New(genState(10), 0, false, false, 0, -1, config)
// changing the values of cPrecd speed should affect the next speed of cCar
cPrecd := New(genState(50), 1, false, false, 0, -1, config)
right := sim.ChangePack{
LaneChanger: &never{},
Preceding: sim.TimePack{},
Trailing: sim.TimePack{},
}
left := sim.ChangePack{
LaneChanger: &never{},
Preceding: sim.TimePack{},
Trailing: sim.TimePack{},
}
log.Println(cCar)
NextState(cCar, cPrecd, left, right, false)
log.Printf("%s", cCar)
```

```
NextState(cCar, cPrecd, left, right, false)
log.Printf("%s", cCar)
}
#-----------------------NEW FILE--------------------------- sim/car/car.go
package car
import (
"bytes"
"fmt"
"math"
"math/rand"
"sim"
"sim/common"
)
var conf sim.Config
// New creates a new car using default settings
func New(initState *sim.CarState, id int, isOnRamp bool, genTime float32, rampEnd float32, config sim.Config)
*sim.Car {
conf = config
cPos := initState.Position
pPos := initState.Position
c := &sim.Car{
ID: id,
GenTime: genTime,
Prev: &sim.CarState{
Position: pPos,
Speed: initState.Speed,
SpaceGap: initState.SpaceGap,
VSafeN: initState.VSafeN,
AccState: initState.AccState,
IsProbe: initState.IsProbe,
SelectedForExit: initState.SelectedForExit,
ExitPosition: initState.ExitPosition,
},
Curr: &sim.CarState{
Position: cPos,
Speed: initState.Speed,
SpaceGap: initState.SpaceGap,
VSafeN: initState.VSafeN,
AccState: initState.AccState,
IsProbe: initState.IsProbe,
SelectedForExit: initState.SelectedForExit,
ExitPosition: initState.ExitPosition,
},
Next: &sim.CarState{
Position: initState.Position,
Speed: initState.Speed,
SpaceGap: initState.SpaceGap,
VSafeN: initState.VSafeN,
AccState: initState.AccState,
IsProbe: initState.IsProbe,
SelectedForExit: initState.SelectedForExit,
ExitPosition: initState.ExitPosition,
},
Internal: &sim.CarInternalState{},
Extra: &sim.CarExtraInfo{
SecondRun: false,
Preceding: &sim.CarState{},
RampMerging: false,
OnRamp: isOnRamp,
RampEnd: rampEnd,
PrecedingInLeftOfRamp: &sim.CarState{},
},
}
return c
}
func debugDetails(c *sim.Car) string {
var buff []byte
b := bytes.NewBuffer(buff)
b.WriteString(fmt.Sprintf("ID: %d, GenTime: %.1f\n", c.ID, c.GenTime))
b.WriteString(fmt.Sprintf("Prev: %+v\n", c.Prev))
b.WriteString(fmt.Sprintf("Curr: %+v\n", c.Curr))
b.WriteString(fmt.Sprintf("Next: %+v\n", c.Next))
b.WriteString(fmt.Sprintf("Internal: %+v\n", c.Internal))
b.WriteString(fmt.Sprintf("&{SecondRun:%v Preceding:%+v RampMerging:%v OnRamp:%v RampEnd:%.1f
PrecedingInLeftOfRamp:%+v}", c.Extra.SecondRun, c.Extra.Preceding, c.Extra.RampMerging,
return b.String()
}
// NextState computes and sets the next state of the car and returns the car's intent to change.
func NextState(c *sim.Car, preceding *sim.Car, left sim.ChangePack, right sim.ChangePack,
isApproachingRedLight bool, isOnRampMerging bool) (int, error) {
c.Extra.Preceding = preceding.Curr
c.Extra.RampMerging = isOnRampMerging
c.Extra.PrecedingInLeftOfRamp = left.Preceding.Curr
*c.Prev = *c.Curr
*c.Curr = *c.Next
simulate(c)
currPack := sim.TimePack{Curr: c.Curr, Prev: c.Prev}
precPack := sim.TimePack{Curr: preceding.Curr, Prev: preceding.Prev}
// If there is a red light, stay in the same lane and reduce speed. Otherwise, bahave as normal
if isApproachingRedLight {
c.Next.Speed = c.Curr.Speed - 2
if c.Next.Speed < 0 {
c.Next.Speed = 0
}
c.Next.Position -= c.Next.Speed * conf.TimeStep
calcNextTimestepPos(c)
} else {
wantsToExit := nearDesignatedExit(c)
if !c.Extra.OnRamp && wantsToExit {
rSpeed, rPosition, cr := right.Can(currPack, precPack, right.Preceding, right.Trailing)
if cr {
c.Next.Speed = rSpeed
c.Next.Position = rPosition
return sim.ExitLane, nil
}
// Since current is now in prev, check prev
}
if c.Extra.OnRamp || (!wantsToExit && (!c.Extra.OnRamp && (c.Prev.Speed > preceding.Curr.Speed))) {
lChangeProb := conf.LaneChangeProb
if c.Extra.OnRamp {
lChangeProb = conf.MergeProb
}
lSpeed, lPosition, cl := left.Can(currPack, precPack, left.Preceding, left.Trailing)
wl := left.Want(lChangeProb)
rSpeed, rPosition, cr := right.Can(currPack, precPack, right.Preceding, right.Trailing)
wr := right.Want(conf.LaneChangeProb)
if cl && wl {
c.Next.Speed = lSpeed
c.Next.Position = lPosition
c.Extra.OnRamp = false
return sim.ChangeLeft, nil
} else if cr && wr {
c.Next.Speed = rSpeed
c.Next.Position = rPosition
return sim.ChangeRight, nil
} else {
// If car is on merging region and cannot merge and there is no other car in front of it, reduce the speed by 2
// Then undo the position change and compute position again
if c.Extra.RampMerging && (c.Extra.Preceding.Position == math.MaxFloat32) {
c.Next.Speed = c.Curr.Speed - 2
if c.Next.Speed < 0 {
c.Next.Speed = 0
}
c.Next.Position -= c.Next.Speed * conf.TimeStep
calcNextTimestepPos(c)
}
}
}
}
return sim.Stay, nil
}
func nearDesignatedExit(c *sim.Car) bool {
// Since current is now in prev, check prev
distanceToExit := (c.Prev.ExitPosition + conf.ExitLaneLength) - c.Prev.Position
if distanceToExit < 0 {
return false
}
return c.Prev.SelectedForExit && (distanceToExit <= conf.ExitDistance)
}
func leaveRamp(c *sim.Car) {
c.Extra.OnRamp = false
}
func simulate(c *sim.Car) {
calcAccDec(c)
calcSynchronizationDistance(c)
calcDelta(c)
calcSpeedAdaptation(c)
calcSpaceGap(c)
calcVSafeN(c)
calcSafeSpeed(c)
calcNextTimestepSpeed(c)
calcNextTimestepPos(c)
}
func calcAccDec(c *sim.Car) {
r1 := rand.Float32()
// EQ-15
P0 := float32(0.575 + 0.125*math.Min(1, float64(c.Curr.Speed/10.0))) // p0
if c.Curr.AccState == 1 {
```

```
P0 = 1
}
// EQ-16
P1 := float32(0.3) // p1
if c.Curr.AccState == -1 {
P1 = 0.48 + 0.32*common.Theta(c.Curr.Speed-15) // p2
}
// EQ-13
c.Internal.Acceleration = conf.MaxAcceleration * common.Theta(P0-r1)
// EQ-14
c.Internal.Deceleration = conf.MaxAcceleration * common.Theta(P1-r1)
}
func adjustSpeedMergingRamp(c *sim.Car, vLN float32) float32 {
if c.Extra.RampMerging {
vPlusN := c.Extra.PrecedingInLeftOfRamp.Speed
vLN = float32(math.Max(0, math.Min(float64(conf.MaxFreeFlowSpeed), float64(vPlusN+conf.DeltaVR2))))
}
return vLN
}
// EQ-5
func calcSynchronizationDistance(c *sim.Car) {
vLN := c.Extra.Preceding.Speed
vLN = adjustSpeedMergingRamp(c, vLN)
c.Internal.SynchronizationDistance = conf.Length + common.G(c.Curr.Speed, vLN, conf.TimeStep, conf.K,
conf.MaxAcceleration, conf.Beta)
}
// Must be run after calcAccDec & calcSynchronizationDistance
// EQ-4
func calcDelta(c *sim.Car) {
vLN := c.Extra.Preceding.Speed
vLN = adjustSpeedMergingRamp(c, vLN)
secondVal := float32(math.Min(float64(c.Internal.Acceleration*conf.TimeStep), float64(vLN-c.Curr.Speed)))
currentDelta := float32(math.Max(float64(-c.Internal.Deceleration*conf.TimeStep), float64(secondVal)))
c.Internal.Delta = currentDelta
}
// Must be run after calcDelta
// EQ-3
func calcSpeedAdaptation(c *sim.Car) {
xLN := c.Extra.Preceding.Position
if c.Extra.RampMerging {
xLN = c.Extra.PrecedingInLeftOfRamp.Position
}
if (xLN - c.Curr.Position) <= c.Internal.SynchronizationDistance {
c.Internal.SpeedAdaptation = c.Curr.Speed + c.Internal.Delta
} else {
c.Internal.SpeedAdaptation = c.Curr.Speed + c.Internal.Acceleration*conf.TimeStep
}
}
// EQ-18
func calcSpaceGap(c *sim.Car) {
gap := c.Extra.Preceding.Position - c.Curr.Position - conf.Length
c.Next.SpaceGap = gap
}
// Must be called after calcSpaceGap
func calcVSafeN(c *sim.Car) {
vsn := common.VSafe(c.Next.SpaceGap, c.Extra.Preceding.Speed, conf.MaxDeceleration, conf.TimeStep)
```

```
c.Next.VSafeN = vsn
}
// Must be called after calcVSafeN
// EQ-17
func calcSafeSpeed(c *sim.Car) {
at := conf.MaxAcceleration * conf.TimeStep
secondVal := math.Min(float64(c.Extra.Preceding.Speed-at), float64(c.Extra.Preceding.SpaceGap/conf.TimeStep))
// EQ-19
// v^a_1
anticipatedSpeedOfPrecedingCar := math.Max(0, math.Min(float64(c.Extra.Preceding.VSafeN-at), secondVal))
c.Internal.SafeSpeed = float32(math.Min(float64(c.Next.VSafeN),
float64(c.Next.SpaceGap/conf.TimeStep)+anticipatedSpeedOfPrecedingCar))
}
// EQ-10
func calcAccState(c *sim.Car, noiseFreeSpeed float32) {
if noiseFreeSpeed < c.Curr.Speed-conf.Sigma {
c.Next.AccState = -1
} else if noiseFreeSpeed > c.Curr.Speed+conf.Sigma {
c.Next.AccState = 1
} else {
c.Next.AccState = 0
}
}
// EQ-9
func calcNoise(c *sim.Car, noiseFreeSpeed float32) float32 {
if c.Next.AccState == -1 {
// EQ-11
r := rand.Float32()
decelerationRandomSource := conf.MaxAcceleration * conf.TimeStep * common.Theta(conf.DecelerationProb-r)
return -1 * decelerationRandomSource
} else if c.Next.AccState == 1 {
// EQ-12
r := rand.Float32()
accelerationRandomSource := conf.MaxAcceleration * conf.TimeStep * common.Theta(conf.AccelerationProb-r)
return -1 * accelerationRandomSource
} else {
c.Next.AccState = 0
return 0
}
}
// EQ-7
func calcNoiseFreeSpeed(c *sim.Car) float32 {
return float32(math.Max(0, math.Min(float64(conf.MaxFreeFlowSpeed),
math.Min(float64(c.Internal.SpeedAdaptation), float64(c.Internal.SafeSpeed)))))
}
// EQ-8
func calcNoisySpeed(c *sim.Car, noiseFreeSpeed float32) float32 {
noise := calcNoise(c, noiseFreeSpeed)
firstMinVal := math.Min(float64(conf.MaxFreeFlowSpeed), float64(noiseFreeSpeed+noise))
secondMinVal := math.Min(float64(c.Curr.Speed+conf.MaxAcceleration*conf.TimeStep),
float64(c.Internal.SafeSpeed))
return float32(math.Max(0, math.Min(firstMinVal, secondMinVal)))
}
// Must be run after calcSpeedAdaptation & calcSafeSpeed
// EQ-1
func calcNextTimestepSpeed(c *sim.Car) {
```

```
newNoiseFreeSpeed := calcNoiseFreeSpeed(c)
calcAccState(c, newNoiseFreeSpeed)
if c.Extra.SecondRun {
newNoiseFreeSpeed = calcNoisySpeed(c, newNoiseFreeSpeed)
}
if !c.Extra.SecondRun {
c.Extra.SecondRun = true
}
c.Next.Speed = newNoiseFreeSpeed
}
func rampSpeedAdjustment(c *sim.Car, currentSpeed float32) float32 {
if c.Extra.OnRamp {
// set a speed that would keep it in lane
newPos := c.Curr.Position + currentSpeed*conf.TimeStep
if newPos > c.Extra.RampEnd {
for {
currentSpeed = currentSpeed / 2.0
newPos = c.Curr.Position + currentSpeed*conf.TimeStep
if newPos < c.Extra.RampEnd {
break
}
if currentSpeed < 1 {
currentSpeed = 0
break
}
}
}
}
return currentSpeed
}
// EQ-2
func calcNextTimestepPos(c *sim.Car) {
c.Next.Position = c.Curr.Position + c.Next.Speed*conf.TimeStep
}
#----------------------NEW FILE-------------------------- sim/car/car_repo.go
package car
import (
"bytes"
"fmt"
"log"
"math"
"sim"
"sort"
)
type sortedCars []*sim.Car
type repo struct {
cars sortedCars
maxLen int
plusInf *sim.Car
minusInf *sim.Car
}
// NewCarRepo creates a car repository
func NewCarRepo() sim.CarRepo {
maxLen := 1000000
cars := make(sortedCars, 0, maxLen)
plusInf := &sim.Car{
```

```
ID: sim.PlusInfID,
Prev: &sim.CarState{
Position: math.MaxFloat32,
Speed: math.MaxFloat32,
SpaceGap: math.MaxFloat32,
VSafeN: math.MaxFloat32,
AccState: 0,
IsProbe: false,
SelectedForExit: false,
ExitPosition: math.MaxFloat32,
},
Curr: &sim.CarState{
Position: math.MaxFloat32,
Speed: math.MaxFloat32,
SpaceGap: math.MaxFloat32,
VSafeN: math.MaxFloat32,
AccState: 0,
IsProbe: false,
SelectedForExit: false,
ExitPosition: math.MaxFloat32,
},
Next: &sim.CarState{
Position: math.MaxFloat32,
Speed: math.MaxFloat32,
SpaceGap: math.MaxFloat32,
VSafeN: math.MaxFloat32,
AccState: 0,
IsProbe: false,
SelectedForExit: false,
ExitPosition: math.MaxFloat32,
},
Internal: &sim.CarInternalState{},
Extra: &sim.CarExtraInfo{},
}
minusInf := &sim.Car{
ID: sim.MinusInfID,
Prev: &sim.CarState{
Position: -math.MaxFloat32,
Speed: -math.MaxFloat32,
SpaceGap: math.MaxFloat32,
VSafeN: -math.MaxFloat32,
AccState: 0,
IsProbe: false,
SelectedForExit: false,
ExitPosition: math.MaxFloat32,
},
Curr: &sim.CarState{
Position: -math.MaxFloat32,
Speed: -math.MaxFloat32,
SpaceGap: math.MaxFloat32,
VSafeN: -math.MaxFloat32,
AccState: 0,
IsProbe: false,
SelectedForExit: false,
ExitPosition: math.MaxFloat32,
},
Next: &sim.CarState{
Position: -math.MaxFloat32,
Speed: -math.MaxFloat32,
SpaceGap: math.MaxFloat32,
VSafeN: -math.MaxFloat32,
AccState: 0,
IsProbe: false,
SelectedForExit: false,
ExitPosition: math.MaxFloat32,
},
Internal: &sim.CarInternalState{},
Extra: &sim.CarExtraInfo{},
}
return &repo{
cars: cars,
maxLen: maxLen,
plusInf: plusInf,
minusInf: minusInf,
}
}
func (r *repo) Leng() int {
return len(r.cars)
}
func (s sortedCars) Len() int {
return len(s)
}
func (s sortedCars) Swap(i, j int) {
s[i], s[j] = s[j], s[i]
}
func (s sortedCars) Less(i, j int) bool {
return s[i].Curr.Position < s[j].Curr.Position
}
func (r repo) String() string {
var buf []byte
b := bytes.NewBuffer(buf)
for _, c := range r.cars {
b.WriteString(fmt.Sprintf("%s, ", c))
}
return b.String()
}
func (r *repo) Sort() {
sort.Sort(sort.Reverse(r.cars))
}
// Add adds a car to the car repository
func (r *repo) Add(c *sim.Car) {
if len(r.cars) == cap(r.cars) {
r.maxLen = 2 * r.maxLen
newCars := make(sortedCars, 0, r.maxLen)
for i := 0; i < len(r.cars); i++ {
newCars = append(newCars, r.cars[i])
}
r.cars = newCars
}
if len(r.cars) == 0 {
r.cars = append(r.cars, c)
} else {
```

```
p := bSearch(c.Curr.Position, r.cars, 0, len(r.cars)-1)
if c.Curr.Position >= r.cars[p].Curr.Position {
r.cars = append(r.cars, nil)
copy(r.cars[p+1:], r.cars[p:])
r.cars[p] = c
} else {
r.cars = append(r.cars, nil)
copy(r.cars[p+2:], r.cars[p+1:])
r.cars[p+1] = c
}
}
//r.cars = append(r.cars, c)
sort.Sort(sort.Reverse(r.cars))
}
// Remove removes a car to the car repository
func (r *repo) Remove(c *sim.Car) {
j := -1
for i := 0; i < len(r.cars); i++ {
if c == r.cars[i] {
j = i
break
}
}
if j == -1 {
log.Println("---------------------> ", c, r.cars)
}
r.cars = append(r.cars[:j], r.cars[j+1:]...)
}
func bSearch(pos float32, c sortedCars, s int, e int, next ...string) int {
m := (s + e) / 2
startPos := c[s].Curr.Position
midPos := c[m].Curr.Position
if len(next) > 0 {
startPos = c[s].Next.Position
midPos = c[m].Next.Position
}
if e <= s {
return s
}
if e == s+1 {
if pos > startPos {
return s
}
return e
}
if pos > midPos {
return bSearch(pos, c, s, m-1, next...)
}
if pos == midPos {
return m
}
return bSearch(pos, c, m, e, next...)
}
// After returns the first car after a position
func (r *repo) After(pos float32, excludeID int) *sim.Car {
if len(r.cars) == 0 {
```

```
return r.plusInf
}
i := bSearch(pos, r.cars, 0, len(r.cars)-1)
possible := []int{i, i - 1, i - 2}
for _, p := range possible {
if p >= 0 && p < len(r.cars) && r.cars[p].ID != excludeID && r.cars[p].Curr.Position >= pos {
return r.cars[p]
}
}
return r.plusInf
}
// After returns the first car after a position
func (r *repo) AfterInNext(pos float32, excludeID int) *sim.Car {
if len(r.cars) == 0 {
return r.plusInf
}
i := bSearch(pos, r.cars, 0, len(r.cars)-1, "next")
possible := []int{i, i - 1, i - 2}
for _, p := range possible {
if p >= 0 && p < len(r.cars) && r.cars[p].ID != excludeID && r.cars[p].Next.Position >= pos {
return r.cars[p]
}
}
return r.plusInf
}
// Before returns the last car before a position
func (r *repo) Before(pos float32) *sim.Car {
if len(r.cars) == 0 {
return r.minusInf
}
i := bSearch(pos, r.cars, 0, len(r.cars)-1)
possible := []int{i, i + 1, i + 2}
for _, p := range possible {
if p >= 0 && p < len(r.cars) && r.cars[p].Curr.Position <= pos {
return r.cars[p]
}
}
return r.minusInf
}
// Before returns the last car before a position
func (r *repo) BeforeInNext(pos float32) *sim.Car {
if len(r.cars) == 0 {
return r.minusInf
}
i := bSearch(pos, r.cars, 0, len(r.cars)-1, "next")
possible := []int{i, i + 1, i + 2}
for _, p := range possible {
if p >= 0 && p < len(r.cars) && r.cars[p].Next.Position <= pos {
return r.cars[p]
}
}
return r.minusInf
}
func (r *repo) GetBetween(start float32, end float32) []*sim.Car {
ret := make([]*sim.Car, 0)
for _, c := range r.cars {
```

```
if c.Prev.Position != 0 && c.Prev.Position < end && c.Curr.Position >= start {
ret = append(ret, c)
}
}
return ret
}
// AllProbes returns all the cars that are marked as probe cars
func (r *repo) AllProbes() []*sim.Car {
probes := make(sortedCars, 0, r.maxLen)
for i := 0; i < len(r.cars); i++ {
if r.cars[i].Curr.IsProbe {
probes = append(probes, r.cars[i])
}
}
return probes
}
// Get returns car at an index. Returns and error in case index is out of range.
func (r *repo) Get(idx int) (*sim.Car, bool) {
if idx < len(r.cars) {
return r.cars[idx], true
}
return nil, false
}
func (r *repo) GetByID(id int) (*sim.Car, bool) {
for _, car := range r.cars {
if car.ID == id {
return car, true
}
}
return nil, false
}
func (r *repo) GetFirst() (*sim.Car, bool) {
if len(r.cars) != 0 {
return r.cars[len(r.cars)-1], true
}
return nil, false
}
#----------------------NEW FILE--------------------------- sim/car/init.go
package car
import (
"math/rand"
"sim"
"time"
)
func init() {
if !sim.HasPresetSeeds {
sim.CarSeed = time.Now().UTC().UnixNano()
}
rand.Seed(sim.CarSeed)
}
#----------------------NEW FILE--------------------------- sim/change/security_test.go
package change
import (
"log"
"sim"
"testing"
```

```
)
func state(position float32, speed float32, spaceGap float32) *sim.CarState {
return &sim.CarState{
Position: position,
Speed: speed,
SpaceGap: spaceGap,
VSafeN: -1000,
AccState: 0,
IsProbe: false,
}
}
func TestPrecedingSecure(t *testing.T) {
conf := sim.DefaultConfig()
preceding := state(0, 30, 31)
current := state(0, 30, 31)
currSpeed := float32(30)
isSec := securePreceding(current, preceding, currSpeed, conf)
if !isSec {
t.Errorf("Expeceted to be secure, got insecure.")
}
}
func TestPrecedingInsecure(t *testing.T) {
conf := sim.DefaultConfig()
preceding := state(0, 30, 30)
current := state(0, 30, 30)
currSpeed := float32(30)
isSec := securePreceding(current, preceding, currSpeed, conf)
if isSec {
t.Errorf("Expeceted to be insecure, got secure.")
}
}
func TestTrailingSecure(t *testing.T) {
conf := sim.DefaultConfig()
trailing := state(0, 30, 31)
current := state(0, 30, 31)
currSpeed := float32(30)
isSec := secureTrailing(current, trailing, currSpeed, conf)
if !isSec {
t.Errorf("Expeceted to be secure, got insecure.")
}
}
func TestTrailingInsecure(t *testing.T) {
conf := sim.DefaultConfig()
trailing := state(0, 30, 30)
current := state(0, 30, 30)
currSpeed := float32(30)
isSec := secureTrailing(current, trailing, currSpeed, conf)
if isSec {
t.Errorf("Expeceted to be insecure, got secure.")
}
}
func TestRampSecurityStarAll(t *testing.T) {
current := sim.TimePack{
Curr: state(130, 30, 31),
Prev: state(100, 30, 31),
}
```

```
precedingInTarget := sim.TimePack{
Curr: state(170, 30, 31),
Prev: state(130, 30, 31),
}
trailingInTarget := sim.TimePack{
Curr: state(40, 30, 31),
Prev: state(10, 30, 31),
}
c := sim.DefaultConfig()
speed, pos, isSec := secureRamp(current, precedingInTarget, trailingInTarget, c)
log.Printf("speed: %.2f, position: %.2f, isSec: %v", speed, pos, isSec)
}
#----------------------NEW FILE-------------------------- sim/change/rightmost.go
package change
import (
"sim"
)
type changeRightmost struct {
config sim.Config
current *sim.CarState
precedingInSame *sim.CarState
precedingInTarget *sim.CarState
trailingInTarget *sim.CarState
}
// NewRightmost creates a new instance of right lane changer.
func NewRightmost(config sim.Config) sim.LaneChanger {
return &changeRightmost{config: config}
}
func (c *changeRightmost) Can(current sim.TimePack, precedingInSame sim.TimePack, precedingInTarget
sim.TimePack, trailingInTarget sim.TimePack) (float32, float32, bool) {
c.current = current.Curr
c.precedingInSame = precedingInSame.Curr
c.precedingInTarget = precedingInTarget.Curr
c.trailingInTarget = trailingInTarget.Curr
currPos := current.Curr.Position
start := current.Curr.ExitPosition
end := start + c.config.ExitLaneLength
cond1 := currPos >= start && currPos <= end
cond2 := current.Curr.SelectedForExit
return c.current.Speed, c.current.Position + c.current.Speed*c.config.TimeStep, cond1 && cond2
}
func (c *changeRightmost) Want(probability float32) bool {
return true
}
#----------------------NEW FILE-------------------------- sim/change/normal_right.go
package change
import (
"math"
"math/rand"
"sim"
)
type changeRight struct {
config sim.Config
current *sim.CarState
precedingInSame *sim.CarState
precedingInTarget *sim.CarState
```

```
trailingInTarget *sim.CarState
}
// NewNormalRight creates a new instance of right lane changer.
func NewNormalRight(config sim.Config) sim.LaneChanger {
return &changeRight{config: config}
}
func (c *changeRight) Can(current sim.TimePack, precedingInSame sim.TimePack, precedingInTarget
sim.TimePack, trailingInTarget sim.TimePack) (float32, float32, bool) {
c.current = current.Curr
c.precedingInSame = precedingInSame.Curr
c.precedingInTarget = precedingInTarget.Curr
c.trailingInTarget = trailingInTarget.Curr
cond1 := c.can()
cond2 := secureTrailing(current.Curr, c.trailingInTarget, c.trailingInTarget.Speed, c.config)
cond3 := securePreceding(current.Curr, c.precedingInTarget, c.precedingInTarget.Speed, c.config)
return c.current.Speed, c.current.Position + c.current.Speed*c.config.TimeStep, cond1 && cond2 && cond3
}
func (c *changeRight) Want(probability float32) bool {
return rand.Float32() < probability
}
func (c *changeRight) can() bool {
vPlusN := c.precedingInTarget.Speed
vLN := c.precedingInSame.Speed
if c.precedingInTarget.SpaceGap > c.config.LookAheadDistance {
vPlusN = math.MaxFloat32
}
if c.current.SpaceGap > c.config.LookAheadDistance {
vLN = math.MaxFloat32
}
cond1 := vPlusN > vLN+c.config.Sigma2
cond2 := vPlusN > c.current.Speed+c.config.Sigma2
return cond1 || cond2
}
#-----------------------NEW FILE--------------------------- sim/change/normal_left.go
package change
import (
"math"
"math/rand"
"sim"
)
type changeLeft struct {
config sim.Config
current *sim.CarState
precedingInSame *sim.CarState
precedingInTarget *sim.CarState
trailingInTarget *sim.CarState
}
// NewNormalLeft creates a new instance of left lane changer.
func NewNormalLeft(config sim.Config) sim.LaneChanger {
return &changeLeft{config: config}
}
func (c *changeLeft) Can(current sim.TimePack, precedingInSame sim.TimePack, precedingInTarget
sim.TimePack, trailingInTarget sim.TimePack) (float32, float32, bool) {
c.current = current.Curr
c.precedingInSame = precedingInSame.Curr
c.precedingInTarget = precedingInTarget.Curr
```

```
c.trailingInTarget = trailingInTarget.Curr
cond1 := c.can()
cond2 := secureTrailing(current.Curr, c.trailingInTarget, c.trailingInTarget.Speed, c.config)
cond3 := securePreceding(current.Curr, c.precedingInTarget, c.precedingInTarget.Speed, c.config)
return c.current.Speed, c.current.Position + c.current.Speed*c.config.TimeStep, cond1 && cond2 && cond3
}
func (c *changeLeft) Want(probability float32) bool {
return rand.Float32() < probability
}
func (c *changeLeft) can() bool {
vPlusN := c.precedingInTarget.Speed
vLN := c.precedingInSame.Speed
if c.precedingInTarget.SpaceGap > c.config.LookAheadDistance {
vPlusN = math.MaxFloat32
}
if c.current.Speed > c.config.LookAheadDistance {
vLN = math.MaxFloat32
}
cond1 := vPlusN >= vLN+c.config.Sigma1
cond2 := c.current.Speed >= vLN
return cond1 && cond2
}
#-----------------------NEW FILE--------------------------- sim/change/security.go
package change
import (
"math"
"sim"
"sim/common"
)
func securePreceding(current *sim.CarState, precedingInTarget *sim.CarState, vN float32, c sim.Config) bool {
if precedingInTarget.Speed == math.MaxFloat32 {
return true
}
ts := precedingInTarget.Position - current.Position - c.Length
s0 := vN * c.TimeStep
s1 := common.G(vN, precedingInTarget.Speed, c.TimeStep, c.K, c.MaxAcceleration, c.Beta)
return float64(ts) > math.Min(float64(s0), float64(s1))
}
func secureTrailing(current *sim.CarState, trailingInTarget *sim.CarState, vN float32, c sim.Config) bool {
if trailingInTarget.Position == -math.MaxFloat32 {
return true
}
ts := current.Position - trailingInTarget.Position - c.Length
s0 := trailingInTarget.Speed * c.TimeStep
s1 := common.G(trailingInTarget.Speed, vN, c.TimeStep, c.K, c.MaxAcceleration, c.Beta)
return float64(ts) > math.Min(float64(s0), float64(s1))
}
func applyRuleStar(current sim.TimePack, precedingInTarget sim.TimePack, c sim.Config) float32 {
vPlusN := precedingInTarget.Curr.Speed
vN := current.Curr.Speed
vN = float32(math.Min(float64(vPlusN), float64(vN+c.DeltaVR1)))
return vN
}
// Returns speed, position, and whether it is secure or not
func secureRamp(current sim.TimePack, precedingInTarget sim.TimePack, trailingInTarget sim.TimePack, c
sim.Config) (float32, float32, bool) {
```

```
vN := applyRuleStar(current, precedingInTarget, c)
if vN > 30 {
vN = 30
}
cond1 := secureTrailing(current.Curr, trailingInTarget.Curr, vN, c)
cond2 := securePreceding(current.Curr, precedingInTarget.Curr, vN, c)
security := cond1 && cond2
if security {
return vN, current.Curr.Position + vN*c.TimeStep, true
}
security = applyRuleDoubleStar(current, trailingInTarget, precedingInTarget, c)
if security {
var xMN float32
if precedingInTarget.Curr.Position == math.MaxInt32 {
xMN = trailingInTarget.Curr.Position + c.Length
if xMN == -math.MaxInt32 {
xMN = current.Curr.Position
}
} else if trailingInTarget.Curr.Position == -math.MaxInt32 {
xMN = precedingInTarget.Curr.Position - c.Length
} else {
xMN = (precedingInTarget.Curr.Position + trailingInTarget.Curr.Position) / 2.0
}
cc := xMN + vN*c.TimeStep
estimatedTrailing := trailingInTarget.Curr.Position + trailingInTarget.Curr.Speed*c.TimeStep
if cc <= estimatedTrailing {
cc = estimatedTrailing + vN*c.TimeStep
}
if cc <= current.Curr.Position {
cc = current.Curr.Position + vN*c.TimeStep
}
return vN, cc, true
}
return -1, -1, false
}
func applyRuleDoubleStar(current sim.TimePack, trailingInTarget sim.TimePack, precedingInTarget
sim.TimePack, c sim.Config) bool {
minGapOnRamp := c.LambdaOn*precedingInTarget.Curr.Speed + c.Length
cond1 := (precedingInTarget.Curr.Position - trailingInTarget.Curr.Position - c.Length) > minGapOnRamp
return cond1
}
#-----------------------NEW FILE--------------------------- sim/change/never.go
package change
import (
"sim"
)
type never struct {
}
// NewNever creates a new instance of lane changer that never permits the change.
func NewNever() sim.LaneChanger {
return &never{}
}
func (n *never) Can(current sim.TimePack, precedingInSame sim.TimePack, precedingInTarget sim.TimePack,
trailingInTarget sim.TimePack) (float32, float32, bool) {
return -1, -1, false
}
```

```
func (n *never) Want(probability float32) bool {
return false
}
#----------------------NEW FILE-------------------------- sim/change/ramp_left.go
package change
import (
"math"
"math/rand"
"sim"
)
type changeRampLeft struct {
config sim.Config
current *sim.CarState
precedingInSame *sim.CarState
precedingInTarget *sim.CarState
trailingInTarget *sim.CarState
}
// NewRampLeft creates a new instance of left lane changer for ramp's merging region.
func NewRampLeft(config sim.Config) sim.LaneChanger {
return &changeRampLeft{config: config}
}
func (c *changeRampLeft) Can(current sim.TimePack, precedingInSame sim.TimePack, precedingInTarget
sim.TimePack, trailingInTarget sim.TimePack) (float32, float32, bool) {
c.current = current.Curr
c.precedingInSame = precedingInSame.Curr
c.precedingInTarget = precedingInTarget.Curr
c.trailingInTarget = trailingInTarget.Curr
if c.trailingInTarget.Position == -math.MaxFloat32 && c.precedingInTarget.Position == math.MaxFloat32 {
return current.Curr.Speed, current.Curr.Position + current.Curr.Speed*c.config.TimeStep, true
}
cond1 := true
speed, position, cond2 := secureRamp(current, precedingInTarget, trailingInTarget, c.config)
return speed, position, cond1 && cond2
}
func (c *changeRampLeft) Want(probability float32) bool {
return rand.Float32() <= probability
}
func (c *changeRampLeft) can() bool {
vPlusN := c.precedingInTarget.Speed
vLN := c.precedingInSame.Speed
if c.precedingInTarget.SpaceGap > c.config.LookAheadDistance {
vPlusN = math.MaxFloat32
}
if c.current.Speed > c.config.LookAheadDistance {
vLN = math.MaxFloat32
}
cond1 := vPlusN >= vLN+c.config.Sigma1
cond2 := c.current.Speed >= vLN
return cond1 && cond2
}
#----------------------NEW FILE-------------------------- sim/change/init.go
package change
import (
"math/rand"
"sim"
"time"
```

```
)
func init() {
if !sim.HasPresetSeeds {
sim.ChangeSeed = time.Now().UTC().UnixNano()
}
rand.Seed(sim.ChangeSeed)
}
#----------------------NEW FILE-------------------------- sim/change/normal_left_test.go
package change
import (
"log"
"sim"
"testing"
)
func TestChangeSuccessLeft(t *testing.T) {
config := sim.DefaultConfig()
changer := NewNormalLeft(config)
current := sim.TimePack{
Curr: state(130, 30, 31),
Prev: state(100, 30, 31),
}
precedingInTarget := sim.TimePack{
Curr: state(170, 30, 31),
Prev: state(130, 30, 31),
}
precedingInSame := sim.TimePack{
Curr: state(170, 30, 31),
Prev: state(130, 30, 31),
}
trailingInTarget := sim.TimePack{
Curr: state(40, 30, 31),
Prev: state(10, 30, 31),
}
speed, position, can := changer.Can(current, precedingInSame, precedingInTarget, trailingInTarget)
log.Printf("Change lane: speed: %.2f, position: %.2f, can: %v", speed, position, can)
}
#----------------------NEW FILE-------------------------- sim/interfaces.go
package sim
import (
"fmt"
)
const (
// ChangeLeft is the intent to change to the left lane
ChangeLeft = iota
// Stay is the intent to stay in the current lane
Stay
// ChangeRight is the intent to change to the right lane
ChangeRight
// ExitLane is the intent to change to the right and exit the main lane
ExitLane
)
const (
GENERAL = "general"
ALL = "all"
TOP = "top"
MID = "bot"
```

```
BOT = "mid"
ALL_TRAVEL = "all_travel"
)
const (
// PlusInfID is the id of the car at the plus infinity
PlusInfID = -1
// MinusInfID is the id of the car at the minus infinity
MinusInfID = -2
InvalidID = -3
)
// NewCar creates stateful car from an initial state
type NewCar func(initState *CarState, id int, isOnRamp bool, genTime float32, rampEnd float32, config Config)
*Car
// ComputeNext calculates the next speed and position of the car
type ComputeNext func(current *Car, preceding *Car, left ChangePack, right ChangePack, isApproachingRedLight
bool, isOnRampMerging bool) (int, error)
// Car holds the informatios about a stateful car
type Car struct {
ID int
GenTime float32
Prev *CarState
Curr *CarState
Next *CarState
Internal *CarInternalState
Extra *CarExtraInfo
}
func (c Car) String() string {
//return fmt.Sprintf("[ID: %d, Probe: %v, Curr:(Speed: %.2f, Position: %.2f), Next: (Speed: %.2f, Position: %.2f)]",
c.ID, c.Curr.IsProbe, c.Curr.Speed, c.Curr.Position, c.Next.Speed, c.Next.Position)
return fmt.Sprintf("[%d:%.1f@%.1f->%.1f->%.1f]", c.ID, c.Curr.Speed, c.Prev.Position, c.Curr.Position,
c.Next.Position)
}
// CarState is a small interface exposing only the position and speed of the car.
type CarState struct {
Position float32
Speed float32
SpaceGap float32
VSafeN float32
AccState float32
IsProbe bool
SelectedForExit bool
ExitPosition float32
}
// CarInternalState is used for computing the next state of the car.
type CarInternalState struct {
// v_c,n
SpeedAdaptation float32
// v_s,n
SafeSpeed float32
// v^safeN
//vSafeN float32
// // g_n -> stored in state
// spaceGap float32
// b_n
Deceleration float32
// a_n
```

```
Acceleration float32
// D_n
SynchronizationDistance float32
// delta
Delta float32
}
// CarExtraInfo are some non-internal info about the car
type CarExtraInfo struct {
SecondRun bool
Preceding *CarState
RampMerging bool
OnRamp bool
RampEnd float32
PrecedingInLeftOfRamp *CarState
}
// TimePack is a wrapper for holding the current and previous state of a car.
type TimePack struct {
Curr *CarState
Prev *CarState
}
// Controller acts as the red/green light
type Controller interface {
IsRed() bool
Position() float32
Simulate()
}
// LaneChanger offers the functionality to check if a lane change is possible and whether it can happen.
type LaneChanger interface {
// Checks if a car can change lane. If it can, it also returns the new speed and position of the car.
Can(current TimePack, precedingInSame TimePack, precedingInTarget TimePack, trailingInTarget TimePack)
(float32, float32, bool)
Want(probability float32) bool
}
// ChangePack holds all the information about changing to a target lane.
type ChangePack struct {
LaneChanger
Preceding TimePack
Trailing TimePack
}
// CarRepo maintains a list of sorted cars.
type CarRepo interface {
// Add adds a car to the car repository
Add(c *Car)
// Remove removes a car to the car repository
Remove(c *Car)
// After returns the first car after a position
After(pos float32, excludeID int) *Car
// Before returns the last car before a position
Before(pos float32) *Car
// After returns the first car after a position
AfterInNext(pos float32, excludeID int) *Car
// Before returns the last car before a position
BeforeInNext(pos float32) *Car
// GetBetween returns all the cars between two points
GetBetween(start float32, end float32) []*Car
// AllProbes returns all the cars that are marked as probe cars
```

```
AllProbes() []*Car
// Get returns car at an index. Returns and error in case index is out of range.
Get(idx int) (*Car, bool)
GetByID(id int) (*Car, bool)
Leng() int
GetFirst() (*Car, bool)
Sort()
}
// Lane reprents the functionality that a lane of a road supports.
type Lane interface {
CarRepo
// Move changes a car location from the current lane to the target lane.
Simulate() error
// Start returns the starting position of a lane
Start() float32
// End returns the end position of a lane
End() float32
Move() map[*Car]Lane
// Adjacent sets the adjacent lanes
Adjacent(left []Lane, right []Lane)
// IsRamp returns true if the lane is ramp
IsRamp() bool
Clean() []*Car
Place(c *CarState, id int)
Name() string
}
// Generator creates new cars based on some internal distribution.
type Generator interface {
// Cars returns the cars generated in the given timestep and the corresponding time interval
Cars() ([]*CarState, []int, float32)
}
// CarPlacer generates cars and puts the into a lane.
// It is up to the CarPlacer to respect or ignore the preferred location of the generated cars.
type CarPlacer interface {
// Register adds a lane and its corresponding car generator in which cars are generated and placed
Register(l Lane, g Generator)
// GenNPlace generates and places the cars in all the registered lanes
GenNPlace()
}
// Detector simulates a detector on the road.
type Detector interface {
Register(l []Lane)
Detect()
// Predict is called at the begining of the timestep predict based on Kalman Filter
Predict()
ID() int
GatherForController() DetectorPredStat
Report() []Report
// TransferredFrom checks if the cars that have left the source detector have arrived at the current detector.
// It then call Transfer to signal the source detector to clears its just left list.
TransferredFrom()
// Transfer deletes the car ids from the just left list
Transfer(carIDs []int)
Stat(choice string) *DetectorStat
SetPrevious(d Detector, rampDetector Detector)
}
```

```
type Report struct {
Content string
Name string
}
type DetectorPredStat struct {
BotSpeedXHat []float32
TravelAVGMultiAhead [][]float64
FlowBotMultiAhead [][]float64
FlowAllMultiAhead [][]float64
}
type DetectorStat struct {
ID int
PredSpeed []float32
PredFlow []float32
RealSpeed []float32
RealFlow []float32
Density []float32
Location float32
LaneCount int
// JustLeft returns the car ids that have recently left the detector's range
JustLeft map[int]float32
}
type PredictGroup struct {
Curr *DetectorStat
PrevLane *DetectorStat
PrevRamp *DetectorStat
PrevExit *DetectorStat
Name string
}
type SetSimFunc func(probeInfoPaths []string, conf Config, outDir string, simulationDuration float32)
*OptimizerDecision
// Simulator connects the lanes and detectors and outputs the final result of the simulation.
type Simulator interface {
SetOtherSimulator(SetSimFunc, Config)
Simulate() *OptimizerDecision
Save(dir string) []string
}
type OptimizerDecision struct {
AlwaysGreen map[string]bool
Rate map[string]int
}
// Config holds the simulation parameters.
type Config struct {
LookAheadDistance float32
Sigma1 float32
Sigma2 float32
Sigma float32
TimeStep float32
K float32
Beta float32
MaxAcceleration float32
MaxDeceleration float32
DeltaVR1 float32
DeltaVR2 float32
Length float32
LambdaOn float32
```

```
MaxFreeFlowSpeed float32
MaxFreeFlowSpeedOnRamp float32
DecelerationProb float32
AccelerationProb float32
// P_c
LaneChangeProb float32
SimulationDuration float32
LaneLength float32
MergeProb float32
// l_safe
SafeDistance float32
SamplingDuration float32
DetectorRange float32
ProbeProbability float32
SnapshotIncrement float32
MergingRegionLength float32
MergingRampLength float32
NonMergingRampLength float32
OutDir string
OptimizeDir string
// KFN is the kalmant filter history, N
KFN int
AverageCarLength float32
VSyn float32
VJam float32
QJam1 float32 // veh/seconds -> user will input per minute
QMinT1 float32
ROMaxT1 float32
QJam3 float32 // veh/seconds -> user will input per minute
QMinT3 float32
ROMaxT3 float32
PatternMU float32
Tave float32
ExitDistance float32
ExitPositions []float32
ExitProbabilities []float32
ExitLaneLength float32
LightInterval float32
SimpleQtt float64
SimpleRtt float64
StepAheadCount int
Alpha1 float64
Alpha2 float64
Alpha3 float64
Beta1 float64
Beta2 float64
Beta3 float64
W1 float64
W2 float64
W3 float64
Alpha float64
QueueRampMax1 float64
QueueRampMax2 float64
QueueRampMax3 float64
TimeForGreenLight float64
RunSim2 bool
```

```
UseAlwaysGreen bool // set during runtime NOT by csv file
PenalizedDeltaFlowOn int
}
// DefaultConfig returns the default values as stated in the paper.
func DefaultConfig() Config {
timeStep := float32(1)
return Config{
LookAheadDistance: 500,
Sigma1: 1,
Sigma2: 2,
Sigma: 0.01,
TimeStep: timeStep,
K: 3,
Beta: 1,
MaxAcceleration: 0.5,
MaxDeceleration: 0.5,
DeltaVR1: 10,
DeltaVR2: 5,
Length: 7.5,
LambdaOn: 0.75 * timeStep,
MaxFreeFlowSpeed: 30,
MaxFreeFlowSpeedOnRamp: 15,
DecelerationProb: 0.1,
AccelerationProb: 0.17,
LaneChangeProb: 0.2,
MergeProb: 1,
SimulationDuration: 2 * 60 * 60, // 2 hours in seconds
LaneLength: 20 * 1000, // 20 KM
SafeDistance: 5,
SamplingDuration: 10,
DetectorRange: 1,
ProbeProbability: 1,
SnapshotIncrement: 10,
MergingRegionLength: 300,
MergingRampLength: 600,
NonMergingRampLength: 600,
OutDir: "./report",
KFN: 5,
AverageCarLength: 4,
VSyn: 19 * 3.6,
VJam: 10 * 3.6,
QJam1: 20 * 60, // veh/min
QMinT1: 0,
ROMaxT1: 120 * 3, // veh/km (for 3 lanes)
QJam3: 20 * 60, // veh/min
QMinT3: 0,
ROMaxT3: 120 * 3, // veh/km (for 3 lanes)
PatternMU: 1, // TODO don't know the value
Tave: 1, // minute
ExitDistance: 2000, // meter
ExitPositions: []float32{5000},
ExitProbabilities: []float32{0.1},
ExitLaneLength: 600,
LightInterval: 60,
}
}
```

```
// DefaultSecondConfig returns the default values as stated in the paper.
func DefaultSecondConfig() Config {
timeStep := float32(10)
return Config{
LookAheadDistance: 500,
Sigma1: 1,
Sigma2: 2,
Sigma: 0.01,
TimeStep: timeStep,
K: 3,
Beta: 1,
MaxAcceleration: 0.5,
MaxDeceleration: 0.5,
DeltaVR1: 10,
DeltaVR2: 5,
Length: 7.5,
LambdaOn: 0.75 * timeStep,
MaxFreeFlowSpeed: 30,
MaxFreeFlowSpeedOnRamp: 15,
DecelerationProb: 0.1,
AccelerationProb: 0.17,
LaneChangeProb: 0.2,
MergeProb: 1,
SimulationDuration: 2 * 60 * 60, // 2 hours in seconds
LaneLength: 20 * 1000, // 20 KM
SafeDistance: 5,
SamplingDuration: 60,
DetectorRange: 1,
ProbeProbability: 1,
SnapshotIncrement: 10,
MergingRegionLength: 300,
MergingRampLength: 600,
NonMergingRampLength: 600,
OutDir: "./report2",
KFN: 5,
AverageCarLength: 4,
VSyn: 19 * 3.6,
VJam: 10 * 3.6,
QJam1: 20 * 60, // veh/min
QMinT1: 0,
ROMaxT1: 120 * 3, // veh/km (for three lanes)
QJam3: 20 * 60, // veh/min
QMinT3: 0,
ROMaxT3: 120 * 3, // veh/km (for three lanes)
PatternMU: 1, // TODO don't know the value
Tave: 1, // minute
ExitDistance: 2000, // meter
ExitPositions: []float32{5000},
ExitProbabilities: []float32{0.1},
ExitLaneLength: 600,
LightInterval: 60,
}
}
#----------------------NEW FILE-------------------------- sim/placer/placer.go
package placer
import (
"math"
"sim"
"sim/common"
"sort"
)
type place struct {
lanes []sim.Lane
lg map[sim.Lane]sim.Generator
q map[sim.Lane][]*sim.CarState
ids map[sim.Lane][]int
place placer
conf sim.Config
}
type placer func(*place, sim.Lane, []*sim.CarState, float32) []*sim.CarState
var carID = 0
// NewRespectfulPlacer returns a car placer that positions the cars in their preferred position
// and at their preferred speed.
func NewRespectfulPlacer(conf sim.Config) sim.CarPlacer {
return &place{
lanes: make([]sim.Lane, 0),
lg: make(map[sim.Lane]sim.Generator),
q: make(map[sim.Lane][]*sim.CarState),
ids: make(map[sim.Lane][]int),
place: respect,
conf: conf,
}
}
// NewNormalPlacer returns a car placer that positions the cars based on the free spot avialability
// at the begining of the lane.
func NewNormalPlacer(conf sim.Config) sim.CarPlacer {
return &place{
lanes: make([]sim.Lane, 0),
lg: make(map[sim.Lane]sim.Generator),
q: make(map[sim.Lane][]*sim.CarState),
place: normal,
conf: conf,
}
}
// Register adds a lane and its corresponding car generator in which cars are generated and placed
func (p *place) Register(l sim.Lane, g sim.Generator) {
p.lanes = append(p.lanes, l)
p.lg[l] = g
}
// GenNPlace generates and places the cars in all the registered lanes
func (p *place) GenNPlace() {
for _, l := range p.lanes {
g := p.lg[l]
if g != nil { // for exit lanes where no car is generated
newCars, ids, timeInterval := g.Cars()
p.q[l] = append(p.q[l], newCars...)
if ids != nil {
p.ids[l] = append(p.ids[l], ids...)
}
p.q[l] = p.place(p, l, p.q[l], timeInterval)
}
}
```

```
}
type carIDPairs struct {
cars []*sim.CarState
ids []int
}
func (s carIDPairs) Len() int {
return len(s.cars)
}
func (s carIDPairs) Swap(i, j int) {
s.cars[i], s.cars[j] = s.cars[j], s.cars[i]
s.ids[i], s.ids[j] = s.ids[j], s.ids[i]
}
func (s carIDPairs) Less(i, j int) bool {
return s.cars[i].Position < s.cars[j].Position
}
func respect(p *place, l sim.Lane, cars []*sim.CarState, timeInterval float32) []*sim.CarState {
if len(cars) > 0 {
removeCars := make([]*sim.Car, 0, 1000000)
for i := 0; ; i++ {
if me, ok := l.Get(i); ok {
removeCars = append(removeCars, me)
} else {
break
}
}
for _, c := range removeCars {
l.Remove(c)
}
}
pairs := carIDPairs{cars, p.ids[l]}
sort.Sort(sort.Reverse(pairs))
for i, c := range pairs.cars {
l.Place(c, pairs.ids[i])
}
p.ids[l] = make([]int, 0)
return make([]*sim.CarState, 0)
}
func normal(p *place, l sim.Lane, cars []*sim.CarState, timeInterval float32) (remaining []*sim.CarState) {
if l.Leng() == 0 { // can place all cars in queue in equal distance
fixedPosInterval := p.conf.MaxFreeFlowSpeed * timeInterval
pos := l.Start()
var spaceGap float32
var precedingCarSpeed float32
for i := 0; i < len(cars); i++ {
if i == len(cars)-1 {
spaceGap = math.MaxInt32
precedingCarSpeed = math.MaxInt32
} else {
spaceGap = fixedPosInterval - p.conf.Length
precedingCarSpeed = p.conf.MaxFreeFlowSpeed
}
vSafeN := common.VSafe(spaceGap, precedingCarSpeed, p.conf.MaxDeceleration, p.conf.TimeStep)
if l.IsRamp() {
cars[i].Speed = p.conf.MaxFreeFlowSpeedOnRamp
} else {
cars[i].Speed = p.conf.MaxFreeFlowSpeed
}
cars[i].Position = pos
cars[i].SpaceGap = spaceGap
cars[i].VSafeN = vSafeN
l.Place(cars[i], carID)
carID++
pos += fixedPosInterval
}
return make([]*sim.CarState, 0)
}
i := 0
for ; i < len(cars); i++ {
furthestUpstream, _ := l.GetFirst()
if furthestUpstream.Curr.Position-l.Start() >= p.conf.SafeDistance {
speed := furthestUpstream.Curr.Speed
pos := float32(math.Max(float64(l.Start()), float64(furthestUpstream.Curr.Position)-
math.Max(float64(speed*timeInterval), float64(p.conf.SafeDistance))))
spaceGap := furthestUpstream.Curr.Position - pos - p.conf.Length
if spaceGap < 0 {
break
}
precedingCarSpeed := furthestUpstream.Curr.Speed
vSafeN := common.VSafe(spaceGap, precedingCarSpeed, p.conf.MaxDeceleration, p.conf.TimeStep)
cars[i].Speed = speed
cars[i].Position = pos
cars[i].SpaceGap = spaceGap
cars[i].VSafeN = vSafeN
l.Place(cars[i], carID)
carID++
} else {
break
}
}
if i == len(cars) {
return make([]*sim.CarState, 0)
}
return cars[i:]
}
#----------------------NEW FILE-------------------------- sim/gen/poisson_test.go
package gen
import (
"sim"
"testing"
)
func TestParseMUList(t *testing.T) {
path := "test.csv"
parseMUList(path, "1")
}
func TestPoisson(t *testing.T) {
conf := sim.DefaultConfig()
p := NewPoisson(map[string]string{
"path": "test.csv",
"col": "1",
}, conf)
histogram := make([]int, 0)
currMin := -1
```

```
elapsed := float32(0)
for elapsed < 120*60 {
m := int(elapsed / 60)
if m != currMin {
histogram = append(histogram, 0)
currMin = m
}
cars, _, _ := p.Cars()
histogram[m] += len(cars)
elapsed += conf.TimeStep
}
for _, h := range histogram {
str := ""
for i := 0; i < h; i++ {
str += "*"
}
}
}
#-----------------------NEW FILE--------------------------- sim/gen/factory.go
package gen
import (
"sim"
"sim/common"
)
func GenFactory(genConf map[string]string, conf sim.Config) sim.Generator {
if genConf["name"] == "normal" {
return NewNormal(genConf, conf)
} else if genConf["name"] == "lognormal" {
return NewLogNormal(genConf, conf)
} else if genConf["name"] == "fixed" {
return NewFixed(genConf, conf)
} else if genConf["name"] == "exact" {
return NewExact(genConf, conf)
} else if genConf["name"] == "poisson" {
return NewPoisson(genConf, conf)
}
return nil
}
func get(prop string, genConf map[string]string) float32 {
return common.ToFloat(genConf[prop])
}
#-----------------------NEW FILE--------------------------- sim/gen/exact_test.go
package gen
import (
"log"
"sim"
"testing"
)
func TestExact(t *testing.T) {
conf := sim.DefaultConfig()
p := NewExact(map[string]string{
"path": "exact_test.csv",
}, conf)
elapsed := float32(0)
for ; elapsed < 60*1; elapsed += conf.TimeStep {
_, _, _ = p.Cars()
}
cars, _, _ := p.Cars()
log.Println(cars)
}
#----------------------NEW FILE-------------------------- sim/gen/fixed.go
package gen
import (
"sim"
)
// generates fixed distribution of input
type fixed struct {
n int
ts float32
last float32
interval float32
speed float32
p *probeState
}
// NewFixedRate receives the number of cars per __minute__ as input and creates normal generator
func NewFixedRate(genConf map[string]string, conf sim.Config) sim.Generator {
rate := get("rate", genConf)
interval := float32(60) / float32(rate)
return &fixed{
n: 0,
ts: conf.TimeStep,
last: 0,
interval: interval,
speed: conf.MaxFreeFlowSpeed,
p: &probeState{
probeProbability: conf.ProbeProbability,
exitPos: conf.ExitPositions,
exitProb: conf.ExitProbabilities,
},
}
}
func (n *fixed) Cars() ([]*sim.CarState, []int, float32) {
n.n++
now := float32(n.n) * n.ts
timePassed := now - n.last
newCount := int(timePassed / n.interval)
n.last = n.last + float32(newCount)*n.interval
newCars := make([]*sim.CarState, 0)
for i := 0; i < newCount; i++ {
e, p := n.p.exit()
newCars = append(newCars, &sim.CarState{
Position: 0,
Speed: n.speed,
SpaceGap: -1000,
VSafeN: -1000,
AccState: 0.0,
IsProbe: n.p.gen(),
SelectedForExit: e,
ExitPosition: p,
})
}
return newCars, nil, n.interval
```

```
}
#-----------------------NEW FILE--------------------------- sim/gen/normal.go
package gen
import (
"encoding/csv"
"io"
"log"
"math"
"math/rand"
"os"
"sim"
"sim/common"
)
type normal struct {
timeInterval int
timeStep float32
mean float32
variance float32
mvList map[int]mv
genInterval float32
last float32
soFar float32
lastIDX int
n int
speed float32
pp *probeState
}
type mv struct {
mean float32
variance float32
}
func NewNormal(genConf map[string]string, conf sim.Config) sim.Generator {
return &normal{
timeInterval: 60,
timeStep: conf.TimeStep,
mvList: parseMeanVariance(genConf["path"]),
genInterval: 0,
last: 0,
soFar: 0,
lastIDX: -1,
n: 0,
speed: common.ToFloat(genConf["speed"]),
pp: &probeState{
probeProbability: conf.ProbeProbability,
exitPos: conf.ExitPositions,
exitProb: conf.ExitProbabilities,
},
}
}
func (p *normal) Cars() ([]*sim.CarState, []int, float32) {
p.n++
now := float32(p.n) * p.timeStep
idx := int(now / float32(60))
if _, ok := p.mvList[idx]; !ok {
idx = p.lastIDX
}
```

```
if idx != p.lastIDX {
// update generation parameters
p.lastIDX = idx
p.mean = p.mvList[idx].mean
p.variance = p.mvList[idx].variance
x := 0
X := rand.Float64()
acc := float64(0)
for {
newP := p.p(x)
if acc <= X && acc+newP >= X {
break
}
acc += newP
x++
}
p.genInterval = float32(60) / float32(x)
p.soFar = 0
}
timePassed := now - p.last
newCars := make([]*sim.CarState, 0)
if !math.IsInf(float64(p.genInterval), 1) {
newCount := int(timePassed / p.genInterval)
p.last = p.last + float32(newCount)*p.genInterval
for i := 0; i < newCount; i++ {
exit, exitPos := p.pp.exit()
newCars = append(newCars, &sim.CarState{
Position: 0,
Speed: p.speed,
SpaceGap: -1000,
VSafeN: -1000,
AccState: 0.0,
IsProbe: p.pp.gen(),
SelectedForExit: exit,
ExitPosition: exitPos,
})
}
} else {
p.last = now
}
//log.Println("Generated: ", len(newCars))
return newCars, nil, p.genInterval
}
func (p *normal) p(x int) float64 {
a := 1.0 / math.Sqrt(2*math.Pi*math.Pow(float64(p.variance), 2))
b := math.Pow(float64(float32(x)-p.mean), 2)
c := 2 * math.Pow(float64(p.variance), 2)
d := math.Pow(math.E, -1*b/c)
return a * d
}
func parseMeanVariance(path string) map[int]mv {
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
```

```
r := csv.NewReader(f)
_, err = r.Read()
if err != nil {
log.Fatalf("Could not read csv file: %v", err)
}
result := make(map[int]mv)
for {
record, err := r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Fatal(err)
}
result[common.ToInt(record[0])] = mv{
mean: common.ToFloat(record[1]),
variance: common.ToFloat(record[2]),
}
}
return result
}
#----------------------NEW FILE-------------------------- sim/gen/poisson.go
package gen
import (
"encoding/csv"
"io"
"log"
"math"
"math/rand"
"os"
"sim"
"sim/common"
)
// generates poisson distribution of input
type poisson struct {
timeInterval int
timeStep float32
mu float32
muList map[int]float32
genInterval float32
last float32
soFar float32
lastIDX int
n int
speed float32
pp *probeState
random bool
}
func NewPoisson(genConf map[string]string, conf sim.Config) sim.Generator {
return &poisson{
timeInterval: 60,
timeStep: conf.TimeStep,
muList: parseMUList(genConf["path"], genConf["col"]),
genInterval: 0,
last: 0,
soFar: 0,
lastIDX: -1,
n: 0,
speed: common.ToFloat(genConf["speed"]),
pp: &probeState{
probeProbability: conf.ProbeProbability,
exitPos: conf.ExitPositions,
exitProb: conf.ExitProbabilities,
},
random: true,
}
}
func NewFixed(genConf map[string]string, conf sim.Config) sim.Generator {
return &poisson{
timeInterval: 60,
timeStep: conf.TimeStep,
muList: parseMUList(genConf["path"], genConf["col"]),
genInterval: 0,
last: 0,
soFar: 0,
lastIDX: -1,
n: 0,
speed: common.ToFloat(genConf["speed"]),
pp: &probeState{
probeProbability: conf.ProbeProbability,
exitPos: conf.ExitPositions,
exitProb: conf.ExitProbabilities,
},
random: false,
}
}
func (p *poisson) Cars() ([]*sim.CarState, []int, float32) {
p.n++
now := float32(p.n) * p.timeStep
idx := int(now / float32(60))
if _, ok := p.muList[idx]; !ok {
idx = p.lastIDX
}
if idx != p.lastIDX {
// update generation parameters
p.lastIDX = idx
p.mu = p.muList[idx]
if p.random {
x := 0
X := rand.Float64()
acc := float64(0)
for {
newP := p.p(x)
if acc <= X && acc+newP >= X {
break
}
acc += newP
x++
}
p.genInterval = float32(60) / float32(x)
p.soFar = 0
} else {
```

```
p.genInterval = float32(60) / float32(p.mu)
p.soFar = 0
}
}
timePassed := now - p.last
newCars := make([]*sim.CarState, 0)
if !math.IsInf(float64(p.genInterval), 1) {
newCount := int(timePassed / p.genInterval)
p.last = p.last + float32(newCount)*p.genInterval
for i := 0; i < newCount; i++ {
exit, exitPos := p.pp.exit()
newCars = append(newCars, &sim.CarState{
Position: 0,
Speed: p.speed,
SpaceGap: -1000,
VSafeN: -1000,
AccState: 0.0,
IsProbe: p.pp.gen(),
SelectedForExit: exit,
ExitPosition: exitPos,
})
}
} else {
p.last = now
}
return newCars, nil, p.genInterval
}
func (p *poisson) p(x int) float64 {
a := math.Pow(float64(p.mu), float64(x)) * math.Pow(math.E, -float64(p.mu))
b := float64(factorial(x))
return a / b
}
func factorial(x int) int {
if x == 0 {
return 1
}
return x * factorial(x-1)
}
func parseMUList(path string, col string) map[int]float32 {
id := common.ToInt(col)
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
r := csv.NewReader(f)
_, err = r.Read()
if err != nil {
log.Fatalf("Could not read csv file: %v", err)
}
result := make(map[int]float32)
for {
record, err := r.Read()
if err == io.EOF {
break
}
```

```
if err != nil {
log.Fatal(err)
}
result[common.ToInt(record[0])] = common.ToFloat(record[id])
}
return result
}
#----------------------NEW FILE-------------------------- sim/gen/probe.go
package gen
import (
"math"
"math/rand"
)
type probeState struct {
probeProbability float32
exitPos []float32
exitProb []float32
}
func (p *probeState) gen() bool {
return rand.Float32() < p.probeProbability
}
func (p *probeState) exit() (bool, float32) {
prob := rand.Float32()
acc := make([]float32, len(p.exitProb))
acc[0] = p.exitProb[0]
for i := 1; i < len(p.exitProb); i++ {
acc[i] = acc[i-1] + p.exitProb[i]
}
for i, pp := range acc {
if prob < pp {
return true, p.exitPos[i]
}
}
return false, math.MaxFloat32
}
#----------------------NEW FILE-------------------------- sim/gen/exact.go
package gen
import (
"encoding/csv"
"io"
"log"
"os"
"sim"
"sim/common"
)
// generates poisson distribution of input
type exact struct {
timeStep float32
pos []map[int]carInfo
lastIdx int
n int
snapRefreshInterval float32
}
type carInfo struct {
id int
timeGen float32
```

```
position float32
speed float32
spaceGap float32
vSafeN float32
selectedForExit bool
exitPosition float32
}
func NewExact(genConf map[string]string, conf sim.Config) sim.Generator {
interval := common.ToFloat(genConf["interval"])
return &exact{
timeStep: conf.TimeStep,
pos: parsePositions(genConf["path"], interval),
n: 0,
snapRefreshInterval: interval,
}
}
func (e *exact) Cars() ([]*sim.CarState, []int, float32) {
e.n++
newCars := make([]*sim.CarState, 0)
newIDs := make([]int, 0)
now := float32(e.n) * e.timeStep
idx := int(now / e.snapRefreshInterval)
if idx < len(e.pos) && e.lastIdx != idx {
e.lastIdx = idx
for id, info := range e.pos[idx] {
newCars = append(newCars, &sim.CarState{
Position: info.position,
Speed: info.speed,
SpaceGap: info.spaceGap,
VSafeN: info.vSafeN,
AccState: 0.0,
IsProbe: true,
SelectedForExit: info.selectedForExit,
ExitPosition: info.exitPosition,
})
newIDs = append(newIDs, id)
}
}
return newCars, newIDs, -1
}
func parsePositions(path string, snapRefreshInterval float32) []map[int]carInfo {
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
r := csv.NewReader(f)
_, err = r.Read()
if err != nil {
log.Fatalf("Could not read csv file: %v", err)
}
currIntervalIdx := 0
result := make([]map[int]carInfo, 0)
result = append(result, make(map[int]carInfo, 0))
for {
record, err := r.Read()
if err == io.EOF {
```

```
break
}
if err != nil {
log.Fatal(err)
}
e := parseExact(record)
if e.timeGen > float32(currIntervalIdx)*snapRefreshInterval {
for e.timeGen > float32(currIntervalIdx)*snapRefreshInterval {
currIntervalIdx++
result = append(result, make(map[int]carInfo, 0))
}
}
if e.position == -1 {
delete(result[currIntervalIdx], e.id)
} else {
result[currIntervalIdx][e.id] = e
}
}
return result
}
func parseExact(cells []string) carInfo {
return carInfo{
timeGen: common.ToFloat(cells[0]),
id: common.ToInt(cells[1]),
speed: common.ToFloat(cells[2]),
position: common.ToFloat(cells[3]),
spaceGap: common.ToFloat(cells[4]),
vSafeN: common.ToFloat(cells[5]),
selectedForExit: common.ToBool(cells[6]),
exitPosition: common.ToFloat(cells[7]),
}
}
#----------------------NEW FILE-------------------------- sim/gen/lognormal.go
package gen
import (
"encoding/csv"
"io"
"log"
"math"
"math/rand"
"os"
"sim"
"sim/common"
)
type lognormal struct {
timeInterval int
timeStep float32
mean float32
theta float32
variance float32
mvtList map[int]mvt
genInterval float32
last float32
soFar float32
lastIDX int
n int
```

```
speed float32
pp *probeState
}
type mvt struct {
mean float32
variance float32
theta float32
}
func NewLogNormal(genConf map[string]string, conf sim.Config) sim.Generator {
return &lognormal{
timeInterval: 60,
timeStep: conf.TimeStep,
mvtList: parseMeanVarianceTheta(genConf["path"]),
genInterval: 0,
last: 0,
soFar: 0,
lastIDX: -1,
n: 0,
speed: common.ToFloat(genConf["speed"]),
pp: &probeState{
probeProbability: conf.ProbeProbability,
exitPos: conf.ExitPositions,
exitProb: conf.ExitProbabilities,
},
}
}
func (p *lognormal) Cars() ([]*sim.CarState, []int, float32) {
p.n++
now := float32(p.n) * p.timeStep
idx := int(now / float32(60))
if _, ok := p.mvtList[idx]; !ok {
idx = p.lastIDX
}
if idx != p.lastIDX {
// update generation parameters
p.lastIDX = idx
p.mean = p.mvtList[idx].mean
p.variance = p.mvtList[idx].variance
p.theta = p.mvtList[idx].theta
x := 0
X := rand.Float64()
acc := float64(0)
for {
newP := p.p(x)
if acc <= X && acc+newP >= X {
break
}
acc += newP
x++
}
p.genInterval = float32(60) / float32(x)
p.soFar = 0
}
timePassed := now - p.last
newCars := make([]*sim.CarState, 0)
if !math.IsInf(float64(p.genInterval), 1) {
```

```
newCount := int(timePassed / p.genInterval)
p.last = p.last + float32(newCount)*p.genInterval
for i := 0; i < newCount; i++ {
exit, exitPos := p.pp.exit()
newCars = append(newCars, &sim.CarState{
Position: 0,
Speed: p.speed,
SpaceGap: -1000,
VSafeN: -1000,
AccState: 0.0,
IsProbe: p.pp.gen(),
SelectedForExit: exit,
ExitPosition: exitPos,
})
}
} else {
p.last = now
}
//log.Println("Generated: ", len(newCars))
return newCars, nil, p.genInterval
}
func (p *lognormal) p(x int) float64 {
xTheta := float64(x) - float64(p.theta)
a := math.Log(xTheta / float64(p.mean))
b := math.Pow(a, 2)
c := 2 * math.Pow(float64(p.variance), 2)
d := b / c
e := math.Pow(math.E, -1*d)
f := math.Sqrt(2 * math.Pi)
g := xTheta * float64(p.variance) * f
h := e / g
return h
}
func parseMeanVarianceTheta(path string) map[int]mvt {
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
r := csv.NewReader(f)
_, err = r.Read()
if err != nil {
log.Fatalf("Could not read csv file: %v", err)
}
result := make(map[int]mvt)
for {
record, err := r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Fatal(err)
}
result[common.ToInt(record[0])] = mvt{
mean: common.ToFloat(record[1]),
variance: common.ToFloat(record[2]),
theta: common.ToFloat(record[3]),
}
}
return result
}
#----------------------NEW FILE--------------------------- sim/gen/init.go
package gen
import (
"math/rand"
"sim"
"time"
)
func init() {
if sim.HasPresetSeeds {
sim.GeneratorSeed = time.Now().UTC().UnixNano()
}
rand.Seed(sim.GeneratorSeed)
}
#----------------------NEW FILE--------------------------- sim/gen/generator_test.go
package gen
import (
"math"
"sim"
"testing"
)
func TestFixedEmpty(t *testing.T) {
conf := sim.DefaultConfig()
conf.TimeStep = 1
g := NewFixedRate(map[string]string{"rate": "0.1"}, conf)
cars, _, interval := g.Cars()
if interval != float32(60)/float32(0.1) {
t.Error("Incorrect time interval")
}
if len(cars) != 0 {
t.Errorf("Did not expect to get any cars, got %v", cars)
}
}
func TestFixedExact(t *testing.T) {
conf := sim.DefaultConfig()
conf.TimeStep = 1
g := NewFixedRate(map[string]string{"rate": "1"}, conf)
for i := 0; i < 59; i++ {
cars, _, interval := g.Cars()
if interval != float32(60)/float32(1) {
t.Error("Incorrect time interval")
}
if len(cars) != 0 {
t.Errorf("Did not expect to get any cars, got %v", cars)
}
}
cars, _, interval := g.Cars()
if interval != float32(60)/float32(1) {
t.Error("Incorrect time interval")
}
if len(cars) != 1 {
t.Errorf("Expected to see a car, got %v", cars)
}
```

```
}
func TestFixedRemaining(t *testing.T) {
conf := sim.DefaultConfig()
conf.TimeStep = 1
g := NewFixedRate(map[string]string{"rate": "0.9"}, conf)
for i := 0; i < 60+6; i++ {
cars, _, interval := g.Cars()
if interval != float32(60)/float32(0.9) {
t.Error("Incorrect time interval")
}
if len(cars) != 0 {
t.Errorf("Did not expect to get any cars, got %v", cars)
}
}
cars, _, interval := g.Cars()
if interval != float32(60)/float32(0.9) {
t.Error("Incorrect time interval")
}
if len(cars) != 1 {
t.Errorf("Expected to see a car, got %v", cars)
}
expect := 60 + float32(60*0.1)/float32(0.9)
if math.Abs(float64(g.(*fixed).last-expect)) > float64(0.00001) {
t.Errorf("Expected the last car to be generated at %f seconds and one third of a second, but got %f", expect,
g.(*fixed).last)
}
}
#----------------------NEW FILE-------------------------- sim/controller/optimized_light.go
package controller
import (
"encoding/csv"
"io"
"log"
"math"
"os"
"sim"
"sim/common"
)
type optimized struct {
position float32
infoFilePath string
interval float32
timeStep float32
light int
n int
i int
durationOfEachLight float32
allowedCarsInInterval []bool
}
func NewOptimizedLight(position float32, infoFilePath string, interval float32, timeStep float32, timeForEachCar
float64) sim.Controller {
return &optimized{
position: position, infoFilePath: infoFilePath, interval: interval, timeStep: timeStep,
light: GREEN, n: 0, i: 0,
allowedCarsInInterval: make([]bool, int(interval/float32(timeForEachCar))),
durationOfEachLight: float32(timeForEachCar),
}
}
func (p *optimized) IsRed() bool {
return p.light == RED
}
func (p *optimized) Position() float32 {
return p.position
}
func (p *optimized) Simulate() {
p.n++
t := int(math.Floor(float64((float32(p.n) * p.timeStep) / p.interval)))
alwaysGreen := true
if p.i != t {
p.i = t
hourlyRate := 0
alwaysGreen, hourlyRate = parseControllerData(p.infoFilePath, t)
// convert rate from per hour to per minute
rate := int(math.Ceil(float64(hourlyRate) * float64(60) / float64(3600)))
if !alwaysGreen {
// reset
numberOfLightChanges := len(p.allowedCarsInInterval)
for j := 0; j < numberOfLightChanges; j++ {
p.allowedCarsInInterval[j] = false
}
// set new values
if rate == numberOfLightChanges {
for j := 0; j < numberOfLightChanges; j++ {
p.allowedCarsInInterval[j] = true
}
} else if rate < numberOfLightChanges {
skip := numberOfLightChanges / rate
soFar := 0
for j := 0; j < numberOfLightChanges; j += skip {
soFar++
p.allowedCarsInInterval[j] = true
if soFar == rate {
break
}
}
if soFar != rate {
if rate-soFar == 1 {
for j := numberOfLightChanges - 1; j >= 0; j-- {
if p.allowedCarsInInterval[j] == false {
p.allowedCarsInInterval[j] = true
break
}
}
} else if rate-soFar == -1 {
for j := numberOfLightChanges - 1; j >= 0; j-- {
if p.allowedCarsInInterval[j] == true {
p.allowedCarsInInterval[j] = false
break
}
}
}
count := 0
for j := numberOfLightChanges - 1; j >= 0; j-- {
if p.allowedCarsInInterval[j] == true {
count++
}
}
if count != rate {
log.Fatalf("expected at most one left-over, got %d. numberOfLightChanges: %d, rate: %d, allowedCarsInInterval:
%v", rate-count, numberOfLightChanges, rate, p.allowedCarsInInterval
}
}
} else {
// Note: discussed and decided to set the max rate to 700
//log.Fatalf("rate bigger than numberOfLigtchanges: %d > %d. Hourly rate: %d", rate, numberOfLightChanges,
hourlyRate)
log.Printf("Warnning: rate bigger than numberOfLigtchanges: %d > %d. Hourly rate: %d. Setting the light to always
green.", rate, numberOfLightChanges, hourlyRate)
for j := 0; j < numberOfLightChanges; j++ {
p.allowedCarsInInterval[j] = true
}
}
}
}
if alwaysGreen {
p.light = GREEN
} else {
// Calc which interval we are in
// Then set the light to green or red based on allowedCarsInInterval
currTime := int(float32(p.n) * p.timeStep)
timeInsideMinute := currTime % 60
intervalIdx := timeInsideMinute / len(p.allowedCarsInInterval)
if p.allowedCarsInInterval[intervalIdx] {
p.light = GREEN
} else {
p.light = RED
}
}
}
func parseControllerData(path string, t int) (bool, int) {
log.Println("looking for ", t)
//Step,....,always-green,rate
f, err := os.Open(path)
if os.IsNotExist(err) {
log.Println("----> no record of controller file, defaulting to always green")
return true, 0
}
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
r := csv.NewReader(f)
alwaysGreen := true
rate := 0
record, err := r.Read()
alwaysGreenIdx := 0
rateIdx := 0
for i, token := range record {
```

```
if token == "always-green" {
alwaysGreenIdx = i
}
if token == "rate" {
rateIdx = i
}
}
for {
record, err = r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Println(err, record, r.FieldsPerRecord)
log.Fatal(err)
}
step := int(common.ToFloat(record[0]))
log.Println("----> Found", step)
if step == t {
alwaysGreen = common.ToBool(record[alwaysGreenIdx])
rate = common.ToInt(record[rateIdx])
log.Printf("Found a matching timestep. always-green: %v, rate: %d\n", alwaysGreen, rate)
return alwaysGreen, rate
}
}
log.Printf("----> did not find a matching record, defaulting to green. always-green: %v, rate: %d\n", alwaysGreen,
rate)
return alwaysGreen, rate
}
#----------------------NEW FILE-------------------------- sim/controller/always_light.go
package controller
import (
"sim"
)
type always struct {
position float32
light int
}
func NewAlwaysLight(position float32, light int) sim.Controller {
return &always{position: position, light: light}
}
func (p *always) IsRed() bool {
return p.light == RED
}
func (p *always) Position() float32 {
return p.position
}
func (p *always) Simulate() {
}
#----------------------NEW FILE-------------------------- sim/controller/periodic_light.go
package controller
import (
"math"
"sim"
)
type periodic struct {
```

```
position float32
interval float32
timeStep float32
light int
n int
i int
soFar int
}
const (
GREEN = iota
RED
)
func NewPeriodicLight(position float32, interval float32, timeStep float32) sim.Controller {
return &periodic{position: position, interval: interval, timeStep: timeStep, light: GREEN, n: 0, i: 0, soFar: 0}
}
func (p *periodic) IsRed() bool {
return p.light == RED
}
func (p *periodic) Position() float32 {
return p.position
}
func (p *periodic) Simulate() {
p.n++
i := int(math.Floor(float64((float32(p.n) * p.timeStep) / p.interval)))
if p.i != i {
p.i = i
if p.IsRed() {
p.soFar++
if p.soFar == 1 {
p.soFar = 0
p.light = GREEN
}
} else {
p.soFar++
if p.soFar == 4 {
p.soFar = 0
p.light = RED
}
}
}
}
#-----------------------NEW FILE--------------------------- sim/common/common.go
package common
import (
"log"
"math"
"strconv"
)
// EQ-6
func G(mySpeed float32, precedingCarSpeed float32, timeStep float32, k float32, maxAcceleration float32, beta
float32) float32 {
v1 := timeStep * k * mySpeed
v2 := (beta / maxAcceleration) * mySpeed * (mySpeed - precedingCarSpeed)
return float32(math.Max(0, float64(v1+v2)))
}
func Theta(input float32) float32 {
```

```
if input < 0 {
return 0
}
return 1
}
func VSafe(gN float32, vLN float32, macDecel float32, timeStep float32) float32 {
return OldVSafe(gN, vLN, macDecel)
}
func OldVSafe(gN float32, vLN float32, macDecel float32) float32 {
div := vLN / macDecel
betaP := float32(math.Mod(float64(div), 1))
alphaP := div - betaP
dPN := macDecel * (alphaP*betaP + (alphaP*(alphaP-1))/2)
tmp := 2 * ((dPN + gN) / macDecel)
alphaSafe := float32(math.Sqrt(float64(tmp+0.25)) - 0.5)
alphaSafe = float32(math.Abs(float64(alphaSafe)))
betaSafe := (tmp - float32(math.Pow(float64(alphaSafe), 2)) - (alphaSafe)) / (2 + 2*alphaSafe)
vsn := macDecel * (alphaSafe + betaSafe)
if math.IsNaN(float64(vsn)) {
vsn = math.MaxFloat32
}
return vsn
}
func ToInt(str string) int {
val, err := strconv.Atoi(str)
if err != nil {
log.Fatal(err)
}
return val
}
func ToFloat(str string) float32 {
val, err := strconv.ParseFloat(str, 32)
if err != nil {
log.Fatal(err)
}
return float32(val)
}
func ToBool(str string) bool {
val, err := strconv.ParseBool(str)
if err != nil {
log.Fatal(err)
}
return val
}
#----------------------NEW FILE-------------------------- sim/detector/detector.go
package detector
import (
"bytes"
"fmt"
"log"
"math"
"sim"
"strconv"
)
type simpleDetector struct {
id int
```

```
start float32
end float32
timeStep float32
sampling float32
n int
statLength int
stat *statStore
lanes []sim.Lane
t int
averageCarLength float32
vSyn float32
vJam float32
qJam1 float32
qJam3 float32
prev sim.Detector
prevRamp sim.Detector
}
type statStore struct {
travelTime []float32
travelTimeFromPrev []float32
travelTimeLaneFromPrev map[sim.Lane][]float32
totalCount []int
uniqueCount []int
fromPrevCount []int
fromPrevInLaneCount map[sim.Lane][]int
speeds []float32
occupancy []float32
laneCount map[sim.Lane][]int
laneProbeCount map[sim.Lane][]int
laneSpeed map[sim.Lane][]float32
laneProbeSpeed map[sim.Lane][]float32
under map[int]sim.Lane
dStats map[string]*sim.DetectorStat
}
var s struct{}
// NewDetector creates a detector on a location at road.
func NewDetector(id int, location float32, conf sim.Config) sim.Detector {
statLen := int(conf.SimulationDuration/conf.SamplingDuration) + 1
dStats := make(map[string]*sim.DetectorStat)
dStats[sim.GENERAL] = &sim.DetectorStat{
JustLeft: make(map[int]float32),
Location: location,
Density: make([]float32, statLen),
PredSpeed: make([]float32, statLen),
PredFlow: make([]float32, statLen),
}
stat := &statStore{
travelTime: make([]float32, statLen),
travelTimeFromPrev: make([]float32, statLen),
travelTimeLaneFromPrev: make(map[sim.Lane][]float32),
totalCount: make([]int, statLen),
uniqueCount: make([]int, statLen),
fromPrevCount: make([]int, statLen),
fromPrevInLaneCount: make(map[sim.Lane][]int),
speeds: make([]float32, statLen),
occupancy: make([]float32, statLen),
under: make(map[int]sim.Lane),
dStats: dStats,
}
return &simpleDetector{
id: id,
start: location - conf.DetectorRange/2,
end: location + conf.DetectorRange/2,
timeStep: conf.TimeStep,
sampling: conf.SamplingDuration,
n: 0,
statLength: statLen,
t: 0,
averageCarLength: conf.AverageCarLength,
vSyn: conf.VSyn,
vJam: conf.VJam,
qJam1: conf.QJam1,
qJam3: conf.QJam3,
stat: stat,
}
}
func (d *simpleDetector) ID() int {
return d.id
}
func (d *simpleDetector) SetPrevious(prev sim.Detector, prevRamp sim.Detector) {
d.prev = prev
d.prevRamp = prevRamp
}
func (d *simpleDetector) Register(lanes []sim.Lane) {
d.lanes = lanes
d.stat.laneCount = make(map[sim.Lane][]int)
d.stat.laneProbeCount = make(map[sim.Lane][]int)
d.stat.laneSpeed = make(map[sim.Lane][]float32)
d.stat.laneProbeSpeed = make(map[sim.Lane][]float32)
for _, l := range lanes {
d.stat.laneCount[l] = make([]int, d.statLength)
d.stat.laneProbeCount[l] = make([]int, d.statLength)
d.stat.laneSpeed[l] = make([]float32, d.statLength)
d.stat.laneProbeSpeed[l] = make([]float32, d.statLength)
}
d.stat.dStats[sim.GENERAL].LaneCount = len(lanes)
for _, l := range lanes {
d.stat.travelTimeLaneFromPrev[l] = make([]float32, d.statLength)
d.stat.fromPrevInLaneCount[l] = make([]int, d.statLength)
}
}
func (d *simpleDetector) Detect() {
d.n++
t := int(math.Floor(float64((float32(d.n) * d.timeStep) / d.sampling)))
d.t = t
occupancyInc := float32(0)
now := float32(d.n) * d.timeStep
for _, l := range d.lanes {
for _, car := range l.GetBetween(d.end, d.end) {
speed := truncate(car.Curr.Speed)
d.stat.totalCount[t]++
d.stat.speeds[t] += speed
d.stat.laneCount[l][t]++
if car.Curr.IsProbe {
d.stat.laneProbeCount[l][t]++
}
d.stat.laneSpeed[l][t] += speed
if car.Curr.IsProbe {
d.stat.laneProbeSpeed[l][t] += speed
}
}
}
for _, l := range d.lanes {
for _, car := range l.GetBetween(d.start, d.end) {
occupancyInc = d.timeStep
if _, ok := d.stat.under[car.ID]; !ok {
// have not seen this car before
d.stat.under[car.ID] = l
d.stat.travelTime[t] += now - car.GenTime
d.stat.uniqueCount[t]++
}
}
}
d.stat.occupancy[t] += occupancyInc
// density keeps updating. its stable and correct value will be its last value
d.stat.dStats[sim.GENERAL].Density[d.t] = d.stat.occupancy[d.t] * 1000 / float32(d.averageCarLength*d.sampling)
cars := make(map[int]struct{})
for _, l := range d.lanes {
for _, car := range l.GetBetween(d.start, d.end) {
cars[car.ID] = s
}
}
for seen := range d.stat.under {
if _, ok := cars[seen]; !ok {
d.stat.dStats[sim.GENERAL].JustLeft[seen] = now
}
}
for car := range d.stat.dStats[sim.GENERAL].JustLeft {
delete(d.stat.under, car)
}
}
func (d *simpleDetector) TransferredFrom() {
i := int(math.Floor(float64((float32(d.n) * d.timeStep) / d.sampling)))
now := float32(d.n+1) * d.timeStep
if d.prev != nil {
jl := d.prev.Stat(sim.GENERAL).JustLeft
transferred := intersection(jl, d.stat.under)
// Update traveltime in between
for _, id := range transferred {
l := d.stat.under[id]
d.stat.travelTimeFromPrev[i] += now - jl[id]
d.stat.travelTimeLaneFromPrev[l][i] += now - jl[id]
d.stat.fromPrevCount[i]++
d.stat.fromPrevInLaneCount[l][i]++
}
d.prev.Transfer(transferred)
}
if d.prevRamp != nil {
```

```
jl := d.prevRamp.Stat(sim.GENERAL).JustLeft
transferred := intersection(jl, d.stat.under)
// Update traveltime in between
for _, id := range transferred {
l := d.stat.under[id]
d.stat.travelTimeFromPrev[i] += now - jl[id]
d.stat.travelTimeLaneFromPrev[l][i] += now - jl[id]
d.stat.fromPrevCount[i]++
d.stat.fromPrevInLaneCount[l][i]++
}
d.prevRamp.Transfer(transferred)
}
}
func intersection(a map[int]float32, b map[int]sim.Lane) []int {
common := make([]int, 0)
for k := range a {
if _, ok := b[k]; ok {
common = append(common, k)
}
}
return common
}
func (d *simpleDetector) Transfer(carIDs []int) {
for _, id := range carIDs {
delete(d.stat.dStats[sim.GENERAL].JustLeft, id)
}
}
func (d *simpleDetector) Stat(choice string) *sim.DetectorStat {
return d.stat.dStats[choice]
}
func truncate(speed float32) float32 {
places := float32(1 * 100)
truncated := float32(int(speed*places)) / places
return truncated
}
func (d *simpleDetector) Report() []sim.Report {
var b []byte
buf := bytes.NewBuffer(b)
_, _ = buf.WriteString("Averaging Time,Average Speed,")
for i := 0; i < len(d.lanes); i++ {
_, _ = buf.WriteString("Count(lane-")
_, _ = buf.WriteString(d.lanes[i].Name())
_, _ = buf.WriteString("),")
_, _ = buf.WriteString("ProbeCount(lane-")
_, _ = buf.WriteString(d.lanes[i].Name())
_, _ = buf.WriteString("),")
_, _ = buf.WriteString("Speed(lane-")
_, _ = buf.WriteString(d.lanes[i].Name())
_, _ = buf.WriteString("),")
_, _ = buf.WriteString("ProbeSpeed(lane-")
_, _ = buf.WriteString(d.lanes[i].Name())
_, _ = buf.WriteString("),")
}
_, _ = buf.WriteString("Count(all lanes),Occupancy,Density,Travel Time,Travel Time from Previous (lane-
all),Count from prev travel time (lane-all),")
for i := 0; i < len(d.lanes); i++ {
```

```
_, _ = buf.WriteString("Travel Time from Previous (lane-")
_, _ = buf.WriteString(d.lanes[i].Name())
_, _ = buf.WriteString("),")
_, _ = buf.WriteString("Count from prev travel time (lane-")
_, _ = buf.WriteString(d.lanes[i].Name())
_, _ = buf.WriteString("),")
}
buf.WriteString("\n")
for i := 0; i < d.statLength; i++ {
averagingTime := d.sampling
speeds := float32(0)
total := d.stat.totalCount[i]
if total != 0 {
speeds += d.stat.speeds[i] / float32(total)
}
occupancy := d.stat.occupancy[i] * 100.0 / averagingTime
density := d.stat.dStats[sim.GENERAL].Density[i]
laneCounts := make([]int, len(d.lanes))
laneProbeCounts := make([]int, len(d.lanes))
laneSpeeds := make([]float32, len(d.lanes))
laneProbeSpeeds := make([]float32, len(d.lanes))
for k, l := range d.lanes {
laneCounts[k] = d.stat.laneCount[l][i]
laneProbeCounts[k] = d.stat.laneProbeCount[l][i]
laneSpeeds[k] = d.stat.laneSpeed[l][i]
laneProbeSpeeds[k] = d.stat.laneProbeSpeed[l][i]
}
_, _ = buf.WriteString(fmt.Sprintf("%.2f", (averagingTime * float32(i+1))))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", speeds))
_, _ = buf.WriteString(",")
for j, laneCount := range laneCounts {
_, _ = buf.WriteString(strconv.Itoa(laneCount))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(strconv.Itoa(laneProbeCounts[j]))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", laneSpeeds[j]/(float32(laneCount))))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", laneProbeSpeeds[j]/(float32(laneProbeCounts[j]))))
_, _ = buf.WriteString(",")
}
_, _ = buf.WriteString(strconv.Itoa(total))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", occupancy))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", density))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", d.stat.travelTime[i]))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f", d.stat.travelTimeFromPrev[i]))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%d", d.stat.fromPrevCount[i]))
_, _ = buf.WriteString(",")
for _, l := range d.lanes {
_, _ = buf.WriteString(fmt.Sprintf("%.2f", d.stat.travelTimeLaneFromPrev[l][i]))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%d", d.stat.fromPrevInLaneCount[l][i]))
_, _ = buf.WriteString(",")
}
_, _ = buf.WriteString("\n")
}
return []sim.Report{
sim.Report{Content: buf.String(), Name: ""},
}
}
func (d *simpleDetector) Predict() {
log.Fatal("Predict is not supported in simple detector")
}
func (f *simpleDetector) GatherForController() sim.DetectorPredStat {
log.Fatal("gather for controller is not supported in simple detector")
return sim.DetectorPredStat{}
}
#----------------------NEW FILE-------------------------- sim/detector/factory.go
package detector
import "sim"
func Factory(shouldReport bool, id int, location float32, dType string, path string, conf sim.Config) sim.Detector {
switch dType {
case "normal":
return NewDetector(id, location, conf)
case "predict":
return NewFlowDetectPredict(shouldReport, id, location, path, conf)
}
return nil
}
#----------------------NEW FILE-------------------------- sim/detector/kalman_cal.go
package detector
import (
"bytes"
"fmt"
"sim/kalman"
"strconv"
)
// ----------------------------------------------------------
type SimpleKalman struct {
states []*kalman.State
cumulativeReal []float32
cumulativeNoisy []float32
KFN int
LastN int
SimpleQtt float64
SimpleRtt float64
stepAheadCount int
}
func (k *SimpleKalman) StepAheadStates(state int) []float64 {
return k.states[state].StepAheadStates
}
func (k *SimpleKalman) AddFirstStates(i int) {
state := &kalman.State{
X: float64(k.cumulativeReal[i]), // This comes from first simulation and is calculated at the end of timestep
Y: float64(k.cumulativeNoisy[i]), // use the latest detection result
P: 1,
XHat: float64(k.cumulativeNoisy[i]),
}
k.states = append(k.states, state)
}
func (k *SimpleKalman) Predict(i int, historyStart int) float32 {
// newState is the prediction for the end of this timestep
k.states[len(k.states)-1].X = float64(k.cumulativeReal[i])
k.states[len(k.states)-1].Y = float64(k.cumulativeNoisy[i])
newState := kalman.SimplePredict(k.states[historyStart:], k.KFN, k.SimpleQtt, k.SimpleRtt, k.stepAheadCount)
k.states = append(k.states, &newState)
return float32(newState.XHat)
}
func (k *SimpleKalman) ToCsv(sampling float32, realProbeRate []float32) string {
var b []byte
buf := bytes.NewBuffer(b)
_, _ = buf.WriteString("Step,T,time,Xt,Yt,N,P(t/t-1),Kt,P(t/t),XHat(t/t-
1),Xhat(t/t),Xhat(t+1/t),P(t+1/t),rhat(t),Rhat(t),qhat(t),Qhat(t),Xhat(t/t),Xhat(t+1/t)")
for i := 0; i < k.stepAheadCount; i++ {
_, _ = buf.WriteString(fmt.Sprintf(",Xhat(t+%d/t)", i+2))
}
_, _ = buf.WriteString(",RealProbeRate\n")
t := -k.KFN
for i, state := range k.states {
_, _ = buf.WriteString(strconv.Itoa(i + 1))
_, _ = buf.WriteString(",")
if t < 0 {
_, _ = buf.WriteString("-,")
} else {
_, _ = buf.WriteString("t")
_, _ = buf.WriteString(strconv.Itoa(t))
_, _ = buf.WriteString(",")
}
_, _ = buf.WriteString(strconv.Itoa(i * int(sampling)))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f,%.2f,%d,", float64(k.cumulativeReal[i])/float64(60), k.cumulativeNoisy[i],
k.KFN))
_, _ = buf.WriteString(state.String())
_, _ = buf.WriteString(fmt.Sprintf(",%.2f", realProbeRate[i]))
_, _ = buf.WriteString("\n")
t++
}
return buf.String()
}
// ------------------------------------------------------------
type MultiStepAheadKalman struct {
states []*kalman.State
cumulativeReal []float32
cumulativeNoisy []float32
KFN int
LastN int
stepAheadCount int
}
func (k *MultiStepAheadKalman) StepAheadStates(state int) []float64 {
return k.states[state].StepAheadStates
}
func (k *MultiStepAheadKalman) AddFirstStates(i int) {
state := &kalman.State{
```

```
X: float64(k.cumulativeReal[i]), // This comes from first simulation and is calculated at the end of timestep
Y: float64(k.cumulativeNoisy[i]), // use the latest detection result
P: 1,
XHat: float64(k.cumulativeNoisy[i]),
}
k.states = append(k.states, state)
}
func (k *MultiStepAheadKalman) Predict(i int, historyStart int) float32 {
// newState is the prediction for the end of this timestep
k.states[len(k.states)-1].X = float64(k.cumulativeReal[i])
k.states[len(k.states)-1].Y = float64(k.cumulativeNoisy[i])
newState := kalman.MultiStepAheadPredict(k.states[historyStart:], k.KFN, k.stepAheadCount)
k.states = append(k.states, &newState)
return float32(newState.XHat)
}
func (k *MultiStepAheadKalman) ToCsv(sampling float32, realProbeRate []float32) string {
var b []byte
buf := bytes.NewBuffer(b)
//_, _ = buf.WriteString("Step,T,time,Xt,Yt,N,P(t/t-1),Kt,P(t/t),XHat(t/t-
1),Xhat(t/t),Xhat(t+1/t),P(t+1/t),rhat(t),Rhat(t),qhat(t),Qhat(t),RealProbeRate\n")
_, _ = buf.WriteString("Step,T,time,Xt,Yt,N,P(t/t-1),Kt,P(t/t),XHat(t/t-
1),Xhat(t/t),Xhat(t+1/t),P(t+1/t),rhat(t),Rhat(t),qhat(t),Qhat(t),Xhat(t/t),Xhat(t+1/t)")
for i := 0; i < k.stepAheadCount; i++ {
_, _ = buf.WriteString(fmt.Sprintf(",Xhat(t+%d/t)", i+2))
}
_, _ = buf.WriteString(",RealProbeRate\n")
t := -k.KFN
for i, state := range k.states {
_, _ = buf.WriteString(strconv.Itoa(i + 1))
_, _ = buf.WriteString(",")
if t < 0 {
_, _ = buf.WriteString("-,")
} else {
_, _ = buf.WriteString("t")
_, _ = buf.WriteString(strconv.Itoa(t))
_, _ = buf.WriteString(",")
}
_, _ = buf.WriteString(strconv.Itoa(i * int(sampling)))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f,%.2f,%d,", k.cumulativeReal[i], k.cumulativeNoisy[i], k.KFN))
_, _ = buf.WriteString(state.String())
_, _ = buf.WriteString(fmt.Sprintf(",%.2f", realProbeRate[i]))
_, _ = buf.WriteString("\n")
t++
}
return buf.String()
}
// -----------------------------------------------------------
type TravelKalman struct {
*SimpleKalman
}
func (f *TravelKalman) CalcCMReal(real []float32, curr float32, i int) {
f.SimpleKalman.cumulativeReal[i] = curr // cumulativeLastN(real, curr, i, f.LastN)
}
func (f *TravelKalman) CalcCMNoisy(real []float32, curr float32, i int) {
f.SimpleKalman.cumulativeNoisy[i] = curr // cumulativeLastN(real, curr, i, f.LastN)
```

```
}
func NewTravelKalman(statLen int, LastN int, KFN int, Qtt, Rtt float64, stepAheadCount int) *TravelKalman {
k := &TravelKalman{
SimpleKalman: &SimpleKalman{
KFN: KFN,
LastN: LastN,
states: make([]*kalman.State, 0),
cumulativeReal: make([]float32, statLen),
cumulativeNoisy: make([]float32, statLen),
SimpleQtt: Qtt,
SimpleRtt: Rtt,
stepAheadCount: stepAheadCount,
},
}
return k
}
// ------------------------------------------------------------------
type FlowKalman struct {
*MultiStepAheadKalman
Sampling float32
}
func (f *FlowKalman) CalcCMReal(real []float32, curr float32, i int) {
f.cumulativeReal[i] = cumulativeLastN(real, curr, i, f.LastN, f.Sampling)
}
func (f *FlowKalman) CalcCMNoisy(real []float32, curr float32, i int) {
f.cumulativeNoisy[i] = cumulativeLastN(real, curr, i, f.LastN, f.Sampling)
}
func NewFlowKalman(statLen int, LastN int, KFN int, sampling float32, stepAheadCount int) *FlowKalman {
k := &FlowKalman{
MultiStepAheadKalman: &MultiStepAheadKalman{
KFN: KFN,
LastN: LastN,
states: make([]*kalman.State, 0),
cumulativeReal: make([]float32, statLen),
cumulativeNoisy: make([]float32, statLen),
stepAheadCount: stepAheadCount,
},
Sampling: sampling,
}
return k
}
// ---------------------------------------------------------------
type SpeedKalman struct {
*MultiStepAheadKalman
}
func (f *SpeedKalman) CalcCMReal(real []float32, i int) {
f.cumulativeReal[i] = real[i] * 3.6
}
func (f *SpeedKalman) CalcCMNoisy(noisy []float32, i int) {
f.cumulativeNoisy[i] = noisy[i] * 3.6
}
func NewSpeedKalman(statLen int, LastN int, KFN int, stepAheadCount int) *SpeedKalman {
k := &SpeedKalman{
MultiStepAheadKalman: &MultiStepAheadKalman{
KFN: KFN,
LastN: LastN,
states: make([]*kalman.State, 0),
cumulativeReal: make([]float32, statLen),
cumulativeNoisy: make([]float32, statLen),
stepAheadCount: stepAheadCount,
},
}
return k
}
func cumulativeLastN(real []float32, curr float32, i int, lastN int, sampling float32) float32 {
dvd := sampling / 60.0
if i < lastN {
return curr * (12.0 / dvd)
}
sum := curr
for j := 1; j < lastN; j++ {
sum += real[i-j]
}
return sum * (12.0 / dvd) // To report hourly
}
#-----------------------NEW FILE--------------------------- sim/detector/pred_groups.go
package detector
import (
"encoding/csv"
"fmt"
"io"
"log"
"os"
"sim"
"sim/common"
"sim/kalman"
"strings"
)
type IPredGroup interface {
CalcCM(t int, sampling, probeRate float32)
Predict(t, newT, historyStart int, vSyn, vJam float32)
AddFirstStates(t int)
KalmanStates() []*kalman.State
CalcDensity(t int)
init()
DS() *sim.DetectorStat
}
// -------------------------- FLOW/SPEED --------------------
type PredGroup struct {
name string
flow *FlowKalman
speed *SpeedKalman
SpeedNoisy []float32
FlowNoisy []float32
SpeedReal []float32
FlowReal []float32
RealProbeRate []float32
seenCount []int
DetectorStat *sim.DetectorStat
}
func newPredGroup(name string, statLen int, LastN int, KFN int, location float32, detectorID int, flow, speed,
realProbeRate []float32, sampling float32, stepAheadCount int) *PredGroup
```

```
return &PredGroup{
name: name,
flow: NewFlowKalman(statLen, LastN, KFN, sampling, stepAheadCount),
speed: NewSpeedKalman(statLen, LastN, KFN, stepAheadCount),
DetectorStat: &sim.DetectorStat{
ID: detectorID,
JustLeft: nil,
Location: location,
Density: make([]float32, statLen),
PredSpeed: make([]float32, 0, statLen),
PredFlow: make([]float32, 0, statLen),
RealSpeed: make([]float32, 0, statLen),
RealFlow: make([]float32, 0, statLen),
},
seenCount: make([]int, statLen),
SpeedNoisy: make([]float32, statLen),
FlowNoisy: make([]float32, statLen),
SpeedReal: speed,
FlowReal: flow,
RealProbeRate: realProbeRate,
}
}
func averagePreviousN(probeRates []float32, t, n int) float32 {
var sum float32
count := 0
for i := t; i > t-n; i-- {
if i == -1 {
break
}
count++
sum += probeRates[i]
}
if count == 0 {
if sum == 0 {
return 1
}
return sum
}
avg := sum / float32(count)
if avg == 0 {
return 1
}
return avg
}
func (p *PredGroup) CalcCM(t int, sampling, probeRate float32) {
// Calculate noisy values
p.FlowNoisy[t] = p.FlowNoisy[t] / (sampling / float32(60)) // convert to average per minute
if t != 0 && p.RealProbeRate[t] < 0.0000001 {
// nothing, keep noisy flow as is
} else if t != 0 {
p.FlowNoisy[t] = p.FlowNoisy[t] / averagePreviousN(p.RealProbeRate, t-1, 5) // extrapolate the estimated noisy
flow
}
if p.seenCount[t] == 0 {
p.SpeedNoisy[t] = 0
if t-1 >= 0 {
```

```
p.SpeedNoisy[t] = p.SpeedNoisy[t-1]
}
} else {
p.SpeedNoisy[t] = p.SpeedNoisy[t] / float32(p.seenCount[t]) // just compute the average
}
p.flow.CalcCMReal(p.FlowReal, p.FlowReal[t], t)
p.flow.CalcCMNoisy(p.FlowReal, p.FlowNoisy[t], t)
p.speed.CalcCMReal(p.SpeedReal, t)
p.speed.CalcCMNoisy(p.SpeedNoisy, t)
}
func (p *PredGroup) Predict(t, newT, historyStart int, vSyn, vJam float32) {
flowXHat := p.flow.Predict(t, historyStart)
p.DetectorStat.PredFlow = append(p.DetectorStat.PredFlow, flowXHat)
p.DetectorStat.RealFlow = append(p.DetectorStat.RealFlow, p.flow.cumulativeReal[t])
speedXHat := p.speed.Predict(t, historyStart)
p.DetectorStat.PredSpeed = append(p.DetectorStat.PredSpeed, speedXHat)
p.DetectorStat.RealSpeed = append(p.DetectorStat.RealSpeed, p.speed.cumulativeReal[t])
}
func (p *PredGroup) KalmanStates() []*kalman.State {
return p.flow.states
}
func (p *PredGroup) AddFirstStates(t int) {
p.flow.AddFirstStates(t)
p.speed.AddFirstStates(t)
p.DetectorStat.PredFlow = append(p.DetectorStat.PredFlow, 0)
p.DetectorStat.PredSpeed = append(p.DetectorStat.PredSpeed, 0)
p.DetectorStat.RealFlow = append(p.DetectorStat.RealFlow, 0)
p.DetectorStat.RealSpeed = append(p.DetectorStat.RealSpeed, 0)
}
func (p *PredGroup) CalcDensity(t int) {
// Unit: veh/km
if p.DetectorStat.PredSpeed[t] == 0 {
p.DetectorStat.Density[t] = 0
} else {
p.DetectorStat.Density[t] = p.DetectorStat.PredFlow[t] / p.DetectorStat.PredSpeed[t]
}
}
func (p *PredGroup) DS() *sim.DetectorStat {
return p.DetectorStat
}
func (p *PredGroup) init() {
if len(p.DetectorStat.PredFlow) == 0 {
p.DetectorStat.PredFlow = append(p.DetectorStat.PredFlow, 0)
p.DetectorStat.PredSpeed = append(p.DetectorStat.PredSpeed, 0)
p.DetectorStat.RealFlow = append(p.DetectorStat.RealFlow, 0)
p.DetectorStat.RealSpeed = append(p.DetectorStat.RealSpeed, 0)
}
}
func parseRealFlow(path string, sampling float32) ([]float32, []float32, []float32) {
// REMOVED this line -> Detector 0: located at 1000 meters down the road
//Averaging Time,Average Speed,Count(lane1),Count(lane2),Count(lane3),Count(merge1),Count(all
lanes),Occupancy,Density,Travel Time,Travel Time from Previous
//10.00,0.00,0,0,0,0,0,0.00,0.00,0.00,0.00
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
```

```
}
defer f.Close()
r := csv.NewReader(f)
resultSpeed := make([]float32, 0)
resultFlow := make([]float32, 0)
realProbeRate := make([]float32, 0)
record, err := r.Read()
allIdx := 0
probeTopIdx := 0
probeMidIdx := 0
probeBotIdx := 0
for i, token := range record {
if strings.Contains(token, "Count(all lanes)") {
allIdx = i
break
}
if strings.Contains(token, "ProbeCount(lane-a_top)") {
probeTopIdx = i
}
if strings.Contains(token, "ProbeCount(lane-b_mid)") {
probeMidIdx = i
}
if strings.Contains(token, "ProbeCount(lane-c_bot)") {
probeBotIdx = i
}
}
speed := float32(0)
count := 0
probeCount := 0
for {
record, err = r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Println(err, record, r.FieldsPerRecord)
log.Fatal(err)
}
step := int(common.ToFloat(record[0]))
s := common.ToFloat(record[1])
all := int(common.ToFloat(record[allIdx]))
probes := int(common.ToFloat(record[probeTopIdx]))
probes += int(common.ToFloat(record[probeMidIdx]))
probes += int(common.ToFloat(record[probeBotIdx]))
if step%int(sampling) == 0 {
resultFlow = append(resultFlow, float32(count))
if count == 0 {
realProbeRate = append(realProbeRate, 0)
} else {
realProbeRate = append(realProbeRate, float32(probeCount)/float32(count))
}
if count == 0 {
resultSpeed = append(resultSpeed, 0)
} else {
resultSpeed = append(resultSpeed, speed/float32(count))
}
count = 0
probeCount = 0
speed = 0
}
count += all
probeCount += probes
speed += (s * float32(all))
}
return resultFlow, resultSpeed, realProbeRate
}
func parseRealForLane(name, path string, sampling float32) ([]float32, []float32, []float32) {
// REMOVED this line -> Detector 0: located at 1000 meters down the road
//Averaging Time,Average Speed,Count(lane1),Count(lane2),Count(lane3),Count(merge1),Count(all
lanes),Occupancy,Density,Travel Time,Travel Time from Previous
//10.00,0.00,0,0,0,0,0,0.00,0.00,0.00,0.00
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
r := csv.NewReader(f)
resultSpeed := make([]float32, 0)
resultFlow := make([]float32, 0)
probeRate := make([]float32, 0)
record, err := r.Read()
flowIdx := 0
probeIdx := 0
speedIdx := 0
for i, token := range record {
if token == fmt.Sprintf("Count(lane-%s)", name) {
flowIdx = i
}
if token == fmt.Sprintf("ProbeCount(lane-%s)", name) {
probeIdx = i
}
if token == fmt.Sprintf("Speed(lane-%s)", name) {
speedIdx = i
}
}
speed := float32(0)
count := 0
probeCount := 0
for {
record, err = r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Println(err, record, r.FieldsPerRecord)
log.Fatal(err)
}
step := int(common.ToFloat(record[0]))
s := float32(0)
if record[speedIdx] != "NaN" {
s = common.ToFloat(record[speedIdx])
}
```

```
all := int(common.ToFloat(record[flowIdx]))
probe := int(common.ToFloat(record[probeIdx]))
if step%int(sampling) == 0 {
resultFlow = append(resultFlow, float32(count))
if count == 0 {
probeRate = append(probeRate, 0)
} else {
probeRate = append(probeRate, float32(probeCount)/float32(count))
}
if count == 0 {
resultSpeed = append(resultSpeed, 0)
} else {
resultSpeed = append(resultSpeed, speed/float32(count))
}
count = 0
probeCount = 0
speed = 0
}
count += all
probeCount += probe
speed += (s * float32(all))
}
return resultFlow, resultSpeed, probeRate
}
// -------------------------- TRAVEL --------------------
type TravelPredGroup struct {
travel *TravelKalman
Noisy []float32
FlowNoisy []int
Real []float32
RealProbeRate []float32
DetectorStat *sim.DetectorStat
}
func newTravelPredGroup(statLen int, LastN int, KFN int, location float32, real []float32, realProbeRate []float32,
Qtt, Rtt float64, stepAheadCount int) *TravelPredGroup {
return &TravelPredGroup{
travel: NewTravelKalman(statLen, LastN, KFN, Qtt, Rtt, stepAheadCount),
DetectorStat: &sim.DetectorStat{
JustLeft: nil,
Location: location,
Density: nil,
PredSpeed: nil,
PredFlow: nil,
RealSpeed: nil,
RealFlow: nil,
},
Noisy: make([]float32, statLen),
FlowNoisy: make([]int, statLen),
Real: real,
RealProbeRate: realProbeRate,
}
}
func (p *TravelPredGroup) CalcCM(t int, sampling, probeRate float32) {
count := float32(p.FlowNoisy[t])
if count != 0 {
p.Noisy[t] = p.Noisy[t] / count
```

```
}
if p.Noisy[t] <= 0.01 && t-1 >= 0 { // 0.01 is chosen to mean 0 since after the above calculations, the travel time
could be a float
p.Noisy[t] = p.Noisy[t-1]
}
p.travel.CalcCMReal(p.Real, p.Real[t], t)
p.travel.CalcCMNoisy(p.Real, p.Noisy[t], t)
}
func (p *TravelPredGroup) Predict(t, newT, historyStart int, vSyn, vJam float32) {
p.travel.SimpleKalman.Predict(t, historyStart)
}
func (p *TravelPredGroup) KalmanStates() []*kalman.State {
return p.travel.SimpleKalman.states
}
func (p *TravelPredGroup) AddFirstStates(t int) {
p.travel.SimpleKalman.AddFirstStates(t)
}
func (p *TravelPredGroup) CalcDensity(t int) {
// Nothing
}
func (p *TravelPredGroup) init() {
if len(p.DetectorStat.PredFlow) == 0 {
p.DetectorStat.RealFlow = append(p.DetectorStat.RealFlow, 0)
}
}
func (p *TravelPredGroup) DS() *sim.DetectorStat {
return p.DetectorStat
}
func parseRealTravel(name string, path string, sampling float32) []float32 {
// REMOVED this line -> Detector 0: located at 1000 meters down the road
//Averaging Time,Average Speed,Count(lane1),Count(lane2),Count(lane3),Count(merge1),Count(all
lanes),Occupancy,Density,Travel Time,Travel Time from Previous
//10.00,0.00,0,0,0,0,0,0.00,0.00,0.00,0.00
f, err := os.Open(path)
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
r := csv.NewReader(f)
result := make([]float32, 0)
record, err := r.Read()
idx := 0
for i, token := range record {
if strings.Contains(token, fmt.Sprintf("Travel Time from Previous (lane-%s)", name)) {
idx = i
}
}
count := 0
for {
record, err = r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Println(err, record, r.FieldsPerRecord)
log.Fatal(err)
}
step := int(common.ToFloat(record[0]))
all := int(common.ToFloat(record[idx]))
if step%int(sampling) == 0 {
result = append(result, float32(count))
count = 0
}
count += all
}
return result
}
// -------------------- GENERAL
type GeneralGroup struct {
DetectorStat *sim.DetectorStat
}
func newGeneralGroup() *GeneralGroup {
return &GeneralGroup{DetectorStat: &sim.DetectorStat{JustLeft: make(map[int]float32)}}
}
func (p *GeneralGroup) DS() *sim.DetectorStat {
return p.DetectorStat
}
func (p *GeneralGroup) CalcCM(t int, sampling, probeRate float32) {}
func (p *GeneralGroup) Predict(t, newT, historyStart int, vSyn, vJam float32) {}
func (p *GeneralGroup) AddFirstStates(t int) {}
func (p *GeneralGroup) CalcDensity(t int) {}
func (p *GeneralGroup) init() {}
func (p *GeneralGroup) KalmanStates() []*kalman.State { return nil }
#----------------------NEW FILE-------------------------- sim/detector/predict_detect.go
package detector
import (
"bytes"
"fmt"
"log"
"math"
"sim"
"strconv"
)
type flowDetect struct {
id int
shouldReport bool
lanes []sim.Lane
n int
start float32
end float32
timeStep float32
sampling float32
probeRate float32
KFN int
LastN int
historyStart int
t int
occupancy []float32
averageCarLength float32
vSyn float32
vJam float32
qJam1 float32
```

```
qJam3 float32
prev sim.Detector
prevRamp sim.Detector
group map[string]IPredGroup
under map[int]sim.Lane
travelFromPrevAll []float32
travelFromPrevCount []int
}
// NewFlowDetectPredict creates a detector on a location at road.
func NewFlowDetectPredict(shouldReport bool, id int, location float32, path string, conf sim.Config) sim.Detector {
statLen := int(conf.SimulationDuration/conf.SamplingDuration) + 1
group := make(map[string]IPredGroup)
LastN := 5 // TODO: change 5 to a config as stated above
realFlow, realSpeed, realProbeRate := parseRealFlow(path, conf.SamplingDuration)
group[sim.ALL] = newPredGroup(sim.ALL, statLen, LastN, conf.KFN, location, id, realFlow, realSpeed,
realProbeRate, conf.SamplingDuration, conf.StepAheadCount)
topRealFlow, topRealSpeed, topRealProbeRate := parseRealForLane("a_top", path, conf.SamplingDuration)
group[sim.TOP] = newPredGroup(sim.TOP, statLen, LastN, conf.KFN, location, id, topRealFlow, topRealSpeed,
topRealProbeRate, conf.SamplingDuration, conf.StepAheadCount)
midRealFlow, midRealSpeed, midRealProbeRate := parseRealForLane("b_mid", path, conf.SamplingDuration)
group[sim.MID] = newPredGroup(sim.MID, statLen, LastN, conf.KFN, location, id, midRealFlow, midRealSpeed,
midRealProbeRate, conf.SamplingDuration, conf.StepAheadCount)
botRealFlow, botRealSpeed, botRealProbeRate := parseRealForLane("c_bot", path, conf.SamplingDuration)
group[sim.BOT] = newPredGroup(sim.BOT, statLen, LastN, conf.KFN, location, id, botRealFlow, botRealSpeed,
botRealProbeRate, conf.SamplingDuration, conf.StepAheadCount)
group[sim.GENERAL] = newGeneralGroup()
allRealTravel := parseRealTravel("all", path, conf.SamplingDuration)
group[sim.ALL_TRAVEL] = newTravelPredGroup(statLen, LastN, conf.KFN, location, allRealTravel,
realProbeRate, conf.SimpleQtt, conf.SimpleRtt, conf.StepAheadCount)
return &flowDetect{
shouldReport: shouldReport,
id: id,
n: 0,
start: location - conf.DetectorRange/2,
end: location + conf.DetectorRange/2,
timeStep: conf.TimeStep,
sampling: conf.SamplingDuration,
probeRate: conf.ProbeProbability,
KFN: conf.KFN,
LastN: LastN,
historyStart: -conf.KFN,
t: 0,
occupancy: make([]float32, statLen),
travelFromPrevAll: make([]float32, statLen),
travelFromPrevCount: make([]int, statLen),
averageCarLength: conf.AverageCarLength,
vSyn: conf.VSyn,
vJam: conf.VJam,
qJam1: conf.QJam1,
qJam3: conf.QJam3,
group: group,
under: make(map[int]sim.Lane),
}
}
func (d *flowDetect) ID() int {
return d.id
```

```
}
func (d *flowDetect) Register(lanes []sim.Lane) {
d.lanes = lanes
d.group[sim.ALL].DS().LaneCount = len(d.lanes)
d.group[sim.TOP].DS().LaneCount = 1
d.group[sim.MID].DS().LaneCount = 1
d.group[sim.BOT].DS().LaneCount = 1
d.group[sim.ALL_TRAVEL].DS().LaneCount = len(d.lanes)
}
func (d *flowDetect) SetPrevious(prev sim.Detector, prevRamp sim.Detector) {
d.prev = prev
d.prevRamp = prevRamp
}
func (d *flowDetect) Detect() {
d.n++
i := int(math.Floor(float64((float32(d.n) * d.timeStep) / d.sampling)))
for _, l := range d.lanes {
for _, car := range l.GetBetween(d.end, d.end) { // this returns the cars that have just passed
speed := truncate(car.Curr.Speed)
d.group[sim.ALL].(*PredGroup).FlowNoisy[i]++
d.group[sim.ALL].(*PredGroup).SpeedNoisy[i] += speed
d.group[sim.ALL].(*PredGroup).seenCount[i]++
switch l.Name() {
case "a_top":
d.group[sim.TOP].(*PredGroup).FlowNoisy[i]++
d.group[sim.TOP].(*PredGroup).SpeedNoisy[i] += speed
d.group[sim.TOP].(*PredGroup).seenCount[i]++
case "b_mid":
d.group[sim.MID].(*PredGroup).FlowNoisy[i]++
d.group[sim.MID].(*PredGroup).SpeedNoisy[i] += speed
d.group[sim.MID].(*PredGroup).seenCount[i]++
case "c_bot":
d.group[sim.BOT].(*PredGroup).FlowNoisy[i]++
d.group[sim.BOT].(*PredGroup).SpeedNoisy[i] += speed
d.group[sim.BOT].(*PredGroup).seenCount[i]++
}
// travel time
if _, ok := d.under[car.ID]; !ok {
// have not seen this car before
d.under[car.ID] = 1
}
}
}
occupancyInc := float32(0)
for _, l := range d.lanes {
cars := l.GetBetween(d.start, d.end)
if len(cars) > 0 {
occupancyInc = d.timeStep
}
}
d.occupancy[i] += occupancyInc
// travel time
now := float32(d.n) * d.timeStep
cars := make(map[int]struct{})
for _, l := range d.lanes {
for _, car := range l.GetBetween(d.start, d.end) {
```

```
cars[car.ID] = s
}
}
for seen := range d.under {
if _, ok := cars[seen]; !ok {
d.group[sim.GENERAL].DS().JustLeft[seen] = now
}
}
for car := range d.group[sim.GENERAL].DS().JustLeft {
delete(d.under, car)
}
}
func (d *flowDetect) Predict() {
newT := int(math.Floor(float64((float32(d.n) * d.timeStep) / d.sampling)))
if newT == 0 {
for gName := range d.group {
d.group[gName].init()
}
return
}
if d.t != newT {
for gName := range d.group {
d.group[gName].CalcCM(d.t, d.sampling, d.probeRate)
if d.historyStart >= 0 {
if d.historyStart == 0 {
// Adding another one that will be overwritten by the next call to predict
d.group[gName].AddFirstStates(d.t)
}
d.group[gName].Predict(d.t, newT, d.historyStart, d.vSyn, d.vJam)
if d.ID() == 7 && gName == sim.BOT {
log.Println(d.t, d.group[gName].(*PredGroup).speed.cumulativeReal[d.t],
d.group[gName].(*PredGroup).speed.cumulativeNoisy[d.t])
}
} else {
d.group[gName].AddFirstStates(d.t)
}
d.group[gName].CalcDensity(d.t)
}
d.t = newT
d.historyStart++
}
}
func (d *flowDetect) GatherForController() sim.DetectorPredStat {
speedsBot := d.group[sim.BOT].DS().PredSpeed
avgTravel := d.group[sim.ALL_TRAVEL].KalmanStates()
flowAll := d.group[sim.ALL].KalmanStates()
flowBot := d.group[sim.BOT].KalmanStates()
avgTravelMSA := make([][]float64, len(avgTravel))
for i := range avgTravel {
avgTravelMSA[i] = avgTravel[i].StepAheadStates
}
flowAllMSA := make([][]float64, len(flowAll))
for i := range flowAll {
flowAllMSA[i] = flowAll[i].StepAheadStates
}
flowBotMSA := make([][]float64, len(flowBot))
```

```
for i := range flowAll {
flowBotMSA[i] = flowBot[i].StepAheadStates
}
return sim.DetectorPredStat{
BotSpeedXHat: speedsBot,
TravelAVGMultiAhead: avgTravelMSA,
FlowAllMultiAhead: flowAllMSA,
FlowBotMultiAhead: flowBotMSA,
}
}
func (d *flowDetect) Report() []sim.Report {
// Other than KF output, print the following:
// x, l, tt from ASADA and FODA
if d.shouldReport {
return []sim.Report{
sim.Report{Content: d.group[sim.ALL].(*PredGroup).flow.ToCsv(d.sampling,
d.group[sim.ALL].(*PredGroup).RealProbeRate), Name: "flow-all"},
sim.Report{Content: d.group[sim.ALL].(*PredGroup).speed.ToCsv(d.sampling,
d.group[sim.ALL].(*PredGroup).RealProbeRate), Name: "speed-all"},
sim.Report{Content: d.group[sim.TOP].(*PredGroup).flow.ToCsv(d.sampling,
d.group[sim.TOP].(*PredGroup).RealProbeRate), Name: "flow-top"},
sim.Report{Content: d.group[sim.TOP].(*PredGroup).speed.ToCsv(d.sampling,
d.group[sim.TOP].(*PredGroup).RealProbeRate), Name: "speed-top"},
sim.Report{Content: d.group[sim.MID].(*PredGroup).flow.ToCsv(d.sampling,
d.group[sim.MID].(*PredGroup).RealProbeRate), Name: "flow-mid"},
sim.Report{Content: d.group[sim.MID].(*PredGroup).speed.ToCsv(d.sampling,
d.group[sim.MID].(*PredGroup).RealProbeRate), Name: "speed-mid"},
sim.Report{Content: d.group[sim.BOT].(*PredGroup).flow.ToCsv(d.sampling,
d.group[sim.BOT].(*PredGroup).RealProbeRate), Name: "flow-bot"},
sim.Report{Content: d.group[sim.BOT].(*PredGroup).speed.ToCsv(d.sampling,
d.group[sim.BOT].(*PredGroup).RealProbeRate), Name: "speed-bot"},
sim.Report{Content: d.group[sim.ALL_TRAVEL].(*TravelPredGroup).travel.SimpleKalman.ToCsv(d.sampling,
d.group[sim.ALL_TRAVEL].(*TravelPredGroup).RealProbeRate), Name: "travel-all-average"
sim.Report{Content: d.totalTravelFromFlow(), Name: "total-travel-all"},
sim.Report{Content: d.ToCsv(), Name: "new-travel"},
}
}
return []sim.Report{}
}
func (d *flowDetect) totalTravelFromFlow() string {
travelGroup := d.group[sim.ALL_TRAVEL].(*TravelPredGroup)
flowGroup := d.group[sim.ALL].(*PredGroup).flow
var b []byte
buf := bytes.NewBuffer(b)
_, _ = buf.WriteString("Step,T,time,Xt,Xhat(t+1/t)")
for i := range travelGroup.travel.SimpleKalman.StepAheadStates(0) {
_, _ = buf.WriteString(fmt.Sprintf("MSA_%d,", i))
}
_, _ = buf.WriteString("\n")
t := -travelGroup.travel.SimpleKalman.KFN
for i, state := range travelGroup.travel.SimpleKalman.states {
_, _ = buf.WriteString(strconv.Itoa(i + 1))
_, _ = buf.WriteString(",")
if t < 0 {
_, _ = buf.WriteString("-,")
} else {
```

```
_, _ = buf.WriteString("t")
_, _ = buf.WriteString(strconv.Itoa(t))
_, _ = buf.WriteString(",")
}
_, _ = buf.WriteString(strconv.Itoa(i * int(d.sampling)))
_, _ = buf.WriteString(",")
_, _ = buf.WriteString(fmt.Sprintf("%.2f,", travelGroup.travel.SimpleKalman.cumulativeReal[i]))
// coverting to total
travelNextXHat := state.NextXHat()
flowNextXHat := float64(flowGroup.MultiStepAheadKalman.states[i].NextXHat()) / float64(60)
_, _ = buf.WriteString(fmt.Sprintf("%.2f", travelNextXHat*flowNextXHat))
// printing MultiStepAhead values
for j, msaTravel := range travelGroup.travel.SimpleKalman.StepAheadStates(i) {
msaFlow := flowGroup.MultiStepAheadKalman.StepAheadStates(i)[j]
_, _ = buf.WriteString(fmt.Sprintf("%.2f", msaTravel*msaFlow))
}
_, _ = buf.WriteString("\n")
t++
}
return buf.String()
}
func (d *flowDetect) ToCsv() string {
var b []byte
buf := bytes.NewBuffer(b)
_, _ = buf.WriteString("Step,T,travel_time_all,travel_time_all_count\n")
length := len(d.group[sim.ALL_TRAVEL].(*TravelPredGroup).Noisy)
t := -d.KFN
for i := 0; i < length; i++ {
_, _ = buf.WriteString(strconv.Itoa(i + 1))
_, _ = buf.WriteString(",")
if t < 0 {
_, _ = buf.WriteString("-,")
} else {
_, _ = buf.WriteString("t")
_, _ = buf.WriteString(strconv.Itoa(t))
_, _ = buf.WriteString(",")
}
_, _ = buf.WriteString(fmt.Sprintf("%.2f,", d.travelFromPrevAll[i]))
_, _ = buf.WriteString(fmt.Sprintf("%d,", d.travelFromPrevCount[i]))
_, _ = buf.WriteString("\n")
t++
}
return buf.String()
}
func (d *flowDetect) TransferredFrom() {
i := int(math.Floor(float64((float32(d.n) * d.timeStep) / d.sampling)))
now := float32(d.n+1) * d.timeStep
if d.prev != nil {
jl := d.prev.Stat(sim.GENERAL).JustLeft
transferred := intersection(jl, d.under)
// Update traveltime in between
for _, id := range transferred {
timeDiff := now - jl[id]
d.group[sim.ALL_TRAVEL].(*TravelPredGroup).Noisy[i] += timeDiff
d.group[sim.ALL_TRAVEL].(*TravelPredGroup).FlowNoisy[i]++
d.travelFromPrevAll[i] += timeDiff
```

```
d.travelFromPrevCount[i]++
}
d.prev.Transfer(transferred)
}
if d.prevRamp != nil {
jl := d.prevRamp.Stat(sim.GENERAL).JustLeft
transferred := intersection(jl, d.under)
// Update traveltime in between
for _, id := range transferred {
timeDiff := now - jl[id]
d.group[sim.ALL_TRAVEL].(*TravelPredGroup).Noisy[i] += timeDiff
d.group[sim.ALL_TRAVEL].(*TravelPredGroup).FlowNoisy[i]++
d.travelFromPrevAll[i] += timeDiff
d.travelFromPrevCount[i]++
}
d.prevRamp.Transfer(transferred)
}
}
func (d *flowDetect) Transfer(carIDs []int) {
for _, id := range carIDs {
delete(d.group[sim.GENERAL].DS().JustLeft, id)
}
}
func (d *flowDetect) Stat(choice string) *sim.DetectorStat {
return d.group[choice].DS()
}
#-----------------------NEW FILE--------------------------- sim/simulator/simulator.go
package simulator
import (
"bytes"
"fmt"
"image"
"image/color"
"io/ioutil"
"log"
"math"
"os"
"sim"
"sim/car"
"sim/change"
"sim/detector"
"sim/gen"
"sim/lane"
"sort"
"time"
)
var ss struct{}
type initialSimulator struct {
currTime float32
maxTime float32
snapshotIncrement float32
currSnapshotTime float32
conf sim.Config
conf2 sim.Config
lanes []sim.Lane
cp sim.CarPlacer
```

```
detectors []sim.Detector
probeSnaps map[sim.Lane][]*probeSnapshot
n int
chart map[sim.Lane][][][]float32
moves map[sim.Lane]map[*sim.Car]struct{}
controllers []sim.Controller
timing map[string]int64
callableSecondSim sim.SetSimFunc
}
// LaneInfo holds information need to creat a lane
type LaneInfo struct {
ID int
Name string
Ramp bool
ExitLane bool
Start float32
End float32 // in case of ramp, end is the merge begining
Gen map[string]string
Controllers []sim.Controller
}
// DetectorInfo holds information need to creat a detector
type DetectorInfo struct {
ID int
DType string
Pos float32
Path string
}
type state struct {
pos float32
speed float32
spaceGap float32
vSafeN float32
selectedForExit bool
exitPosition float32
}
type probeSnapshot struct {
t float32
cars map[int]state
}
// LaneInfo and DetectorInfo -> LinkInfo
type LinkInfo struct {
DetectorID int
LanesID []int
PrevDetecIDLane int
PrevDetecIDRamp int
PrevDetectIDRampForTravelTime int
PrevDetecIDExit int
}
// NewInitialSim creates the first simulator that will be run
func NewInitialSim(lInf []LaneInfo, dInfo []DetectorInfo, linkInof []LinkInfo, iCP sim.CarPlacer, config
sim.Config) sim.Simulator {
s := new(initialSimulator)
s.conf = config
s.maxTime = s.conf.SimulationDuration
s.currTime = 0
s.snapshotIncrement = s.conf.SnapshotIncrement
```

```
s.currSnapshotTime = 0
s.n = 0
s.chart = make(map[sim.Lane][][][]float32)
s.moves = make(map[sim.Lane]map[*sim.Car]struct{})
rampCount := createLanes(s, lInf)
log.Println("Number of ramps: ", rampCount)
log.Println("Starting a simulation with max time of ", s.maxTime)
fmt.Println("Starting a simulation with max time of ", s.maxTime)
for _, l := range s.lanes {
s.chart[l] = make([][][]float32, int(s.maxTime/config.SnapshotIncrement)+1)
s.moves[l] = make(map[*sim.Car]struct{})
for i := 0; i < len(s.chart); i++ {
s.chart[l][i] = make([][]float32, 0)
}
}
s.probeSnaps = make(map[sim.Lane][]*probeSnapshot)
s.cp = iCP
for i, l := range s.lanes {
s.probeSnaps[l] = make([]*probeSnapshot, 0)
s.cp.Register(l, gen.GenFactory(lInf[i].Gen, s.conf))
}
s.detectors = make([]sim.Detector, len(dInfo))
for i, inf := range dInfo {
//shouldReport := inf.ID%6 == 1
shouldReport := true
s.detectors[i] = detector.Factory(shouldReport, inf.ID, inf.Pos, inf.DType, inf.Path, s.conf)
relatedLanes := make([]sim.Lane, 0)
for _, j := range linkInof[i].LanesID {
relatedLanes = append(relatedLanes, s.lanes[j])
}
s.detectors[i].Register(relatedLanes)
}
// Refer to the visual in the main.go for detector ids
for i := range dInfo {
idx := linkInof[i].PrevDetecIDLane
ridx := linkInof[i].PrevDetectIDRampForTravelTime
if i != dInfo[i].ID {
log.Fatalf("Mismatch in detector id: %d %d", i, dInfo[i].ID)
}
if idx != -1 && i%6 == 1 {
s.detectors[i].SetPrevious(s.detectors[idx], s.detectors[ridx])
}
}
for i := 0; i < len(s.detectors); i++ {
if linkInof[i].PrevDetecIDLane != -1 && linkInof[i].PrevDetecIDRamp != -1 {
group := make(map[string]*sim.PredictGroup)
group[sim.ALL] = &sim.PredictGroup{
Name: sim.ALL,
Curr: s.detectors[i].Stat(sim.ALL),
PrevLane: s.detectors[linkInof[i].PrevDetecIDLane].Stat(sim.ALL),
PrevRamp: s.detectors[linkInof[i].PrevDetecIDRamp].Stat(sim.ALL),
PrevExit: s.detectors[linkInof[i].PrevDetecIDExit].Stat(sim.ALL),
}
group[sim.TOP] = &sim.PredictGroup{
Name: sim.TOP,
Curr: s.detectors[i].Stat(sim.TOP),
PrevLane: s.detectors[linkInof[i].PrevDetecIDLane].Stat(sim.TOP),
PrevRamp: s.detectors[linkInof[i].PrevDetecIDRamp].Stat(sim.TOP),
PrevExit: s.detectors[linkInof[i].PrevDetecIDExit].Stat(sim.TOP),
}
group[sim.MID] = &sim.PredictGroup{
Name: sim.MID,
Curr: s.detectors[i].Stat(sim.MID),
PrevLane: s.detectors[linkInof[i].PrevDetecIDLane].Stat(sim.MID),
PrevRamp: s.detectors[linkInof[i].PrevDetecIDRamp].Stat(sim.MID),
PrevExit: s.detectors[linkInof[i].PrevDetecIDExit].Stat(sim.MID),
}
group[sim.BOT] = &sim.PredictGroup{
Name: sim.BOT,
Curr: s.detectors[i].Stat(sim.BOT),
PrevLane: s.detectors[linkInof[i].PrevDetecIDLane].Stat(sim.BOT),
PrevRamp: s.detectors[linkInof[i].PrevDetecIDRamp].Stat(sim.BOT),
PrevExit: s.detectors[linkInof[i].PrevDetecIDExit].Stat(sim.BOT),
}
//log.Println(s.detectors[i].ID(), ": ", linkInof[i].PrevDetecIDLane, linkInof[i].PrevDetecIDRamp,
linkInof[i].PrevDetecIDExit)
}
}
s.timing = make(map[string]int64)
return s
}
func createLanes(s *initialSimulator, inf []LaneInfo) int {
normal := make([]LaneInfo, 0)
ramp := make([]LaneInfo, 0)
exit := make([]LaneInfo, 0)
for _, li := range inf {
if li.Ramp {
ramp = append(ramp, li)
} else if li.ExitLane {
exit = append(exit, li)
} else {
normal = append(normal, li)
}
}
shoulder := []sim.Lane{lane.NewNormalLane(lane.NormalInfo{ID: -1, Repo: car.NewCarRepo(), Start: -
math.MaxFloat32, End: s.conf.LaneLength, RChanger: change.NewNever(), LChanger: change
s.lanes = make([]sim.Lane, 0)
for i := 0; i < len(normal); i++ {
right := change.NewNormalRight(s.conf)
left := change.NewNormalLeft(s.conf)
if i == 0 {
left = change.NewNever()
}
if i == len(normal)-1 {
right = change.NewRightmost(s.conf)
}
l := lane.NewNormalLane(lane.NormalInfo{ID: normal[i].ID, Repo: car.NewCarRepo(), Start: normal[i].Start, End:
normal[i].End, RChanger: right, LChanger: left, Name: normal[i].Name},
s.lanes = append(s.lanes, l)
if len(s.lanes)-1 != normal[i].ID {
log.Fatalf("Mismatch in normal lane id: %d %d", len(s.lanes)-1, normal[i].ID)
}
```

```
}
if len(ramp) != len(exit) {
log.Fatalf("Mismatch in ramp and exit: %d vs %d", len(ramp), len(exit))
}
exitLanes := make([]sim.Lane, 0)
for i := 0; i < len(ramp); i++ {
e := lane.NewExitLane(lane.ExitInfo{ID: exit[i].ID, Repo: car.NewCarRepo(), Start: exit[i].Start, Never:
change.NewNever(), Name: exit[i].Name}, car.New, car.NextState, exit[i].Controllers
s.lanes = append(s.lanes, e)
exitLanes = append(exitLanes, e)
if len(s.lanes)-1 != exit[i].ID {
log.Fatalf("Mismatch in exit lane id: %d %d", len(s.lanes)-1, exit[i].ID)
}
r := lane.NewRampLane(lane.RampInfo{ID: ramp[i].ID, Repo: car.NewCarRepo(), Start: ramp[i].Start, MergeStart:
ramp[i].Start + s.conf.NonMergingRampLength + s.conf.MergingRampLength
s.lanes = append(s.lanes, r)
if len(s.lanes)-1 != ramp[i].ID {
log.Fatalf("Mismatch in ramp lane id: %d %d", len(s.lanes)-1, ramp[i].ID)
}
}
s.lanes[0].Adjacent(shoulder, []sim.Lane{s.lanes[1]})
s.lanes[1].Adjacent([]sim.Lane{s.lanes[0]}, []sim.Lane{s.lanes[2]})
s.lanes[2].Adjacent([]sim.Lane{s.lanes[1]}, append(shoulder, exitLanes...))
for i := 4; i < len(s.lanes); i += 2 { // ramps
s.lanes[i].Adjacent([]sim.Lane{s.lanes[2]}, shoulder)
}
for i := 3; i < len(s.lanes); i += 2 { // exit lanes
s.lanes[i].Adjacent(shoulder, shoulder)
}
s.controllers = make([]sim.Controller, 0)
for _, i := range inf {
s.controllers = append(s.controllers, i.Controllers...)
}
return len(ramp)
}
func (s *initialSimulator) Simulate() *sim.OptimizerDecision {
for s.currTime < s.maxTime {
s.currTime += s.conf.TimeStep
log.Printf("----------------------------------- %.0f ----------------------\n", s.currTime)
start := time.Now()
s.cp.GenNPlace()
s.timing["generate"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, c := range s.controllers {
c.Simulate()
}
s.timing["controller_sim"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, l := range s.lanes {
l.Simulate()
}
s.timing["lane_sim"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, d := range s.detectors {
d.Detect()
}
```

```
for i := 1; i < len(s.detectors); i++ {
s.detectors[i].TransferredFrom()
}
s.timing["detector"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, l := range s.lanes {
moves := l.Move()
for m, _ := range moves {
s.moves[l][m] = ss
}
}
s.timing["change_lanes"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, l := range s.lanes {
cleaned := l.Clean()
for _, c := range cleaned {
s.moves[l][c] = ss
}
}
s.timing["clean"] += time.Since(start).Nanoseconds()
start = time.Now()
if s.currTime >= s.currSnapshotTime {
s.currSnapshotTime += s.snapshotIncrement
for _, l := range s.lanes {
currentProbeCars := l.AllProbes()
s.recordProbeCars(s.probeSnaps, l, currentProbeCars)
s.moves[l] = make(map[*sim.Car]struct{})
}
}
s.timing["snapshot"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, l := range s.lanes {
for i := 0; ; i++ {
if me, ok := l.Get(i); ok {
s.chart[l][s.n] = append(s.chart[l][s.n], []float32{float32(me.ID), me.Curr.Position, me.Curr.Speed})
} else {
break
}
}
}
s.timing["chart"] += time.Since(start).Nanoseconds()
if int(s.currTime/s.conf.SnapshotIncrement) != s.n {
s.n++
}
//for _, l := range s.lanes {
// log.Printf("%s\n\n", l)
//}
if int(s.currTime)%60 == 0 {
reports := s.Save(fmt.Sprintf("%s_%.2f", s.conf.OutDir, s.currTime))
sort.Strings(reports)
if s.conf.RunSim2 {
callSecondAt := float32(360)
if s.conf.UseAlwaysGreen {
callSecondAt = s.maxTime - s.conf.TimeStep
}
// Call 2nd simulation
if s.currTime >= callSecondAt {
log.Println("Invoking simulation 2")
decision := s.callableSecondSim(reports, s.conf2, fmt.Sprintf("%s_%.2f", s.conf2.OutDir, s.currTime), s.currTime)
log.Println("Ending simulation 2")
log.Println("--> recieved from simulation 2: ", decision)
// Set it for the next call to controller
}
}
}
}
return nil
}
func (s *initialSimulator) SetOtherSimulator(callableSecondSim sim.SetSimFunc, conf2 sim.Config) {
s.callableSecondSim = callableSecondSim
s.conf2 = conf2
}
func (s *initialSimulator) recordProbeCars(probeSnaps map[sim.Lane][]*probeSnapshot, l sim.Lane,
currentProbeCars []*sim.Car) {
newSnap := s.takeSnapshot(currentProbeCars, l)
probeSnaps[l] = append(probeSnaps[l], newSnap)
}
func (s *initialSimulator) takeSnapshot(cars []*sim.Car, l sim.Lane) *probeSnapshot {
snap := &probeSnapshot{
t: s.currTime,
cars: make(map[int]state),
}
for _, car := range cars {
snap.cars[car.ID] = state{pos: car.Curr.Position, speed: car.Curr.Speed, spaceGap: car.Curr.SpaceGap, vSafeN:
car.Curr.SpaceGap, selectedForExit: car.Curr.SelectedForExit, exitPosition
}
for c, _ := range s.moves[l] {
snap.cars[c.ID] = state{pos: -1, speed: -1, spaceGap: -1, vSafeN: -1, selectedForExit: false, exitPosition:
math.MaxFloat32}
}
return snap
}
func (s *initialSimulator) Save(dir string) []string {
start := time.Now()
reports := make([]string, 0)
os.Mkdir(dir, 0755)
for _, d := range s.detectors {
ioutil.WriteFile(fmt.Sprintf("%s/report-detector-%d.csv", dir, d.ID()), []byte(d.Report()[0].Content), 0755)
}
for l, snaps := range s.probeSnaps {
var b []byte
buf := bytes.NewBuffer(b)
buf.WriteString("gen_time,id,speed,position,spacegap,vsafen,selectedForExit,exitPosition\n")
for _, snap := range snaps {
if len(snap.cars) > 0 {
for id, car := range snap.cars {
buf.WriteString(fmt.Sprintf("%.2f,%d,%.2f,%.2f,%.2f,%.2f,%v,%.1f\n", snap.t, id, car.speed, car.pos, car.spaceGap,
car.vSafeN, car.selectedForExit, car.exitPosition))
}
}
}
path := fmt.Sprintf("%s/probes-in-%s.csv", dir, l.Name())
```

```
ioutil.WriteFile(path, buf.Bytes(), 0755)
reports = append(reports, path)
}
maxID := 0
for _, l := range s.lanes {
for _, carsPlaces := range s.chart[l] {
for _, carPlace := range carsPlaces {
if int(carPlace[0]) > maxID {
maxID = int(carPlace[0])
}
}
}
}
for _, l := range s.lanes {
sPath := fmt.Sprintf("%s/v-t-%s.csv", dir, l.Name())
lPath := fmt.Sprintf("%s/x-t-%s.csv", dir, l.Name())
var sB []byte
var lB []byte
sBuf := bytes.NewBuffer(sB)
lBuf := bytes.NewBuffer(lB)
sBuf.WriteString("car id,")
lBuf.WriteString("car id,")
for i := 0; i <= maxID; i++ {
sBuf.WriteString(fmt.Sprintf("%d,", i))
lBuf.WriteString(fmt.Sprintf("%d,", i))
}
sBuf.Bytes()[len(sBuf.Bytes())-1] = '\n'
lBuf.Bytes()[len(lBuf.Bytes())-1] = '\n'
sBuf.WriteString("Time(sec)\n")
lBuf.WriteString("Time(sec)\n")
for i, carsPlaces := range s.chart[l] {
sBuf.WriteString(fmt.Sprintf("%d,", (i+1)*10))
lBuf.WriteString(fmt.Sprintf("%d,", (i+1)*10))
sVal := make([]float32, maxID+1)
lVal := make([]float32, maxID+1)
for _, carPlace := range carsPlaces {
sVal[int(carPlace[0])] = carPlace[2]
lVal[int(carPlace[0])] = carPlace[1]
}
for j := 0; j <= maxID; j++ {
sBuf.WriteString(fmt.Sprintf("%.1f,", sVal[j]))
lBuf.WriteString(fmt.Sprintf("%.1f,", lVal[j]))
}
sBuf.Bytes()[len(sBuf.Bytes())-1] = '\n'
lBuf.Bytes()[len(lBuf.Bytes())-1] = '\n'
}
ioutil.WriteFile(sPath, sBuf.Bytes(), 0755)
ioutil.WriteFile(lPath, lBuf.Bytes(), 0755)
}
log.Println("----------- Timing for first simulation:")
s.timing["report"] += time.Since(start).Nanoseconds()
for k, v := range s.timing {
log.Printf("%s:\t%.1f sec\n", k, float64(v)/float64(1000000000))
}
log.Println("----------------------------------------")
return reports
```

```
}
func drawXs(width int, height int, img *image.NRGBA) {
for x := 0; x < width; x += 60 {
c := color.NRGBA{255, 0, 0, 255}
for i := 0; i < 50; i++ {
img.Set(x, height-i, c)
img.Set(x+1, height-i, c)
}
}
}
func drawYs(width int, height int, img *image.NRGBA) {
for y := 0; y < height; y += 1000 {
c := color.NRGBA{255, 0, 0, 255}
for i := 0; i < 50; i++ {
img.Set(i, height-y, c)
img.Set(i, height-y-1, c)
}
}
}
#----------------------NEW FILE--------------------------- sim/simulator/predict_simulator.go
package simulator
import (
"bytes"
"encoding/csv"
"fmt"
"io"
"io/ioutil"
"log"
"math"
"os"
"path/filepath"
"sim"
"sim/common"
"time"
)
type flowSim struct {
*initialSimulator
}
// NewFlowSim creates a simulator for flow simulation
func NewFlowSim(inf []LaneInfo, dInfo []DetectorInfo, linkInfo []LinkInfo, iCP sim.CarPlacer, config sim.Config)
sim.Simulator {
return &flowSim{NewInitialSim(inf, dInfo, linkInfo, iCP, config).(*initialSimulator)}
}
func (f *flowSim) Simulate() *sim.OptimizerDecision {
optimizerCalled := false
decision := &sim.OptimizerDecision{}
for f.currTime <= f.maxTime+f.conf.TimeStep {
f.currTime += f.conf.TimeStep
log.Printf("----------------------------------^%.0f ---------------------\n", f.currTime)
start := time.Now()
f.cp.GenNPlace()
f.timing["generate"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, c := range f.controllers {
// Preserve choices from controller in sim1
c.Simulate()
}
f.timing["controller"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, l := range f.lanes {
l.Simulate()
}
f.timing["lane_sim"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, d := range f.detectors {
d.Detect()
}
for i := 1; i < len(f.detectors); i++ {
f.detectors[i].TransferredFrom()
}
for _, l := range f.lanes {
l.Move()
}
f.timing["detect"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, l := range f.lanes {
l.Clean()
}
f.timing["clean"] += time.Since(start).Nanoseconds()
start = time.Now()
for _, d := range f.detectors {
d.Predict()
}
f.timing["predict"] += time.Since(start).Nanoseconds()
start = time.Now()
if f.currTime == f.maxTime && !optimizerCalled {
// Calling at the end of the last 60 minutes and save the result for the next call of simulation 1
decision = f.optimizeForController()
log.Println(decision)
optimizerCalled = true
}
f.timing["optimize-controller"] += time.Since(start).Nanoseconds()
}
return decision
}
type detectorConsts struct {
mainKey string
rampKey string
alpha float64
beta float64
otherDets []int
}
type optimizer struct {
consts []detectorConsts
dets map[string]sim.DetectorPredStat
detsGroupCount int
t int
msa int
ttDisHat []float64
ttLocalHat []float64
flowOnHat []int
alpha float64
```

```
w []float64
queuerampMax []float64
d *sim.OptimizerDecision
VSyn float32
outDir string
useDefaults bool
penalizeDeltaFlowOn int
}
// Convention
// - i is index of needOptimization
// - r is index of detector r = needOptimization[i]
// - j is index of flowOns
// - p is index of multistep ahead timestep
// - k is used when none of the above are usable
// ---> We also have the following dimentions
// - 3D: [i][j][p] or [r][j][p]
// - 2D: [i][p] or [r][p]
func (f *flowSim) optimizeForController() *sim.OptimizerDecision {
consts := []detectorConsts{
{mainKey: "7", rampKey: "10", alpha: f.conf.Alpha1, beta: f.conf.Beta1, otherDets: []int{1, 2}},
{mainKey: "13", rampKey: "16", alpha: f.conf.Alpha2, beta: f.conf.Beta2, otherDets: []int{0, 2}},
{mainKey: "19", rampKey: "22", alpha: f.conf.Alpha3, beta: f.conf.Beta3, otherDets: []int{0, 1}},
}
d := &sim.OptimizerDecision{
AlwaysGreen: map[string]bool{
"7": false,
"13": false,
"19": false,
},
Rate: map[string]int{
"7": 0,
"13": 0,
"19": 0,
},
}
dets := make(map[string]sim.DetectorPredStat)
dets["7"] = f.detectors[7].GatherForController()
dets["10"] = f.detectors[10].GatherForController()
dets["13"] = f.detectors[13].GatherForController()
dets["16"] = f.detectors[16].GatherForController()
dets["19"] = f.detectors[19].GatherForController()
dets["22"] = f.detectors[19].GatherForController()
detsGroupCount := 3
msa := 5
t := int(math.Floor(float64((f.currTime / f.conf.SamplingDuration)))) - 1
prevOptimizerPath := filepath.Join(f.conf.OptimizeDir, "optimize-detect-%s-%s.csv")
// Read the previously calculated values from the file. If all the values are -1,
// then use the default values
ttDisHat, ttLocalHat, flowOnHat, useDefaults, err := retrieveHatValues(detsGroupCount, msa, prevOptimizerPath,
t-1)
if err != nil {
log.Fatalf("Error in reading previous hat values from file: %v", err)
}
o := &optimizer{
consts: consts,
dets: dets,
msa: msa,
d: d,
t: t,
VSyn: f.conf.VSyn,
alpha: f.conf.Alpha,
w: []float64{f.conf.W1, f.conf.W2, f.conf.W3},
queuerampMax: []float64{f.conf.QueueRampMax1, f.conf.QueueRampMax2, f.conf.QueueRampMax3},
detsGroupCount: detsGroupCount,
ttDisHat: ttDisHat,
ttLocalHat: ttLocalHat,
flowOnHat: flowOnHat,
useDefaults: useDefaults,
outDir: f.conf.OptimizeDir,
penalizeDeltaFlowOn: f.conf.PenalizedDeltaFlowOn,
}
o.optimize()
return o.d
}
func retrieveHatValues(detsGroupCount, msa int, path string, t int) ([]float64, []float64, []int, bool, error) {
ttDisHat := make([]float64, detsGroupCount)
ttLocalHat := make([]float64, detsGroupCount)
flowOnHat := make([]int, detsGroupCount)
// Read data from file
f, err := os.Open(path)
if os.IsNotExist(err) {
log.Println("No record of controller file. Using defaults")
return ttDisHat, ttLocalHat, flowOnHat, true, nil
}
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
r := csv.NewReader(f)
headerRecord, err := r.Read()
var matchingRecord []string = nil
for {
record, err := r.Read()
if err == io.EOF {
break
}
if err != nil {
log.Println(err, record, r.FieldsPerRecord)
log.Fatalf("Unexpected error in file %s: %v", path, err)
}
step := int(common.ToFloat(record[0]))
if step == t {
matchingRecord = record
}
}
if matchingRecord == nil {
return nil, nil, nil, false, fmt.Errorf("Could not find timestep %d in %s", t, path)
}
// interpret the data
useDefaults := true
for r := 0; r < detsGroupCount; r++ {
pattern, err := getItemFromRecord(headerRecord, matchingRecord, ",TTdishat")
```

```
if err != nil {
return nil, nil, nil, false, err
}
ttDisHat[r] = pattern
pattern, err = getItemFromRecord(headerRecord, matchingRecord, ",TTlocalhat")
if err != nil {
return nil, nil, nil, false, err
}
ttLocalHat[r] = pattern
pattern, err = getItemFromRecord(headerRecord, matchingRecord, ",Flowonhat")
if err != nil {
return nil, nil, nil, false, err
}
flowOnHat[r] = int(pattern)
if ttDisHat[r] != -1 || ttLocalHat[r] != -1 || flowOnHat[r] != -1 {
useDefaults = false
}
}
return ttDisHat, ttLocalHat, flowOnHat, useDefaults, nil
}
func getItemFromRecord(headerRecord, contentRecord []string, pattern string) (float64, error) {
for i, token := range headerRecord {
if token == pattern {
return float64(common.ToFloat(contentRecord[i])), nil
}
}
return 0, fmt.Errorf("could not find %s in the csv file", pattern)
}
func parseControllerData(path string, t int) (bool, int) {
log.Println("looking for ", t)
//Step,....,always-green,rate
f, err := os.Open(path)
if os.IsNotExist(err) {
log.Println("----> no record of controller file, defaulting to always green")
return true, 0
}
if err != nil {
log.Fatalf("Could not open csv file: %v", err)
}
defer f.Close()
r := csv.NewReader(f)
alwaysGreen := true
rate := 0
record, err := r.Read()
alwaysGreenIdx := 0
rateIdx := 0
for i, token := range record {
if token == "always-green" {
alwaysGreenIdx = i
}
if token == "rate" {
rateIdx = i
}
}
for {
record, err = r.Read()
```

```
if err == io.EOF {
break
}
if err != nil {
log.Println(err, record, r.FieldsPerRecord)
log.Fatal(err)
}
step := int(common.ToFloat(record[0]))
log.Println("----> Found", step)
if step == t {
alwaysGreen = common.ToBool(record[alwaysGreenIdx])
rate = common.ToInt(record[rateIdx])
log.Printf("Found a matching timestep. always-green: %v, rate: %d\n", alwaysGreen, rate)
return alwaysGreen, rate
}
}
log.Printf("----> did not find a matching record, defaulting to green. always-green: %v, rate: %d\n", alwaysGreen,
rate)
return alwaysGreen, rate
}
func (o *optimizer) optimize() {
if o.useDefaults {
// ttDisHat[r][p] = 1 in case of default, use formula 4 and use xhat of flowall of detec 10 instead of flowon-dis use
previous values only if the previous also needed optimization, otherwise use defaults
// ttLocalHat[r][p] = 1 the same ttDisHat
// flowOnHat[r][p] = 400 in case of default, use the xhat of flowall of detector 10
flowOnDisHat := make([][]float64, o.detsGroupCount)
sigmaFlowOnDisHat := make([]int, o.detsGroupCount)
for r := 0; r < o.detsGroupCount; r++ {
flowOnDisHat[r] = make([]float64, o.msa)
sigmaFlowOnDisHat[r] = int(0)
for p := 0; p < o.msa; p++ {
kRamp := o.consts[r].rampKey
o.flowOnHat[r] = int(o.dets[kRamp].FlowAllMultiAhead[o.t][0]) // TODO which timestep?
flowOnDisHat[r][p] = o.dets[kRamp].FlowAllMultiAhead[o.t][p] // TODO which timestep?
sigmaFlowOnDisHat[r] += int(flowOnDisHat[r][p])
}
}
allttDisHat, _ := o.calcAllLocalDisagreementCosts([]int{0, 1, 2}, flowOnDisHat, sigmaFlowOnDisHat)
allttLocalHat, _ := o.calcAllLocalDisagreementCosts([]int{0, 1, 2}, flowOnDisHat, sigmaFlowOnDisHat)
o.ttDisHat = allttDisHat[0]
o.ttLocalHat = allttLocalHat[0]
}
flowOns := []float64{100, 200, 300, 400, 500, 600, 700}
needOptimization := make([]int, 0)
// This loop has a corresponding part at the end as well
//for r := range o.consts {
// if o.dets[o.consts[r].mainKey].BotSpeedXHat[o.t] >= o.VSyn {
// o.d.AlwaysGreen[o.consts[r].mainKey] = true
// } else {
// needOptimization = append(needOptimization, r)
// }
//}
for r := range o.consts {
needOptimization = append(needOptimization, r)
}
```

```
if len(needOptimization) != 0 {
pb, queuerampPB := o.calcAllProbabilityOfBreakdown(needOptimization, flowOns)
// loop over pb and find flowons that MINimize it
// pb should be converted to another pb where newGrid[index of need optimization][index of timestep]
flowOnDises := make([][]float64, len(needOptimization)) // each o.msa elements
for i := range pb {
flowOnDises[i] = make([]float64, o.msa)
for p := 0; p < o.msa; p++ {
minFlowOn := flowOns[len(flowOns)-1] // choosing 700 as default min
min := pb[i][len(flowOns)-1][p] // choosing 700 as default min
for j := 0; j < len(flowOns); j++ {
r := needOptimization[i]
if queuerampPB[i][j][p] >= 0 && queuerampPB[i][j][p] <= o.queuerampMax[r] {
min = math.Min(pb[i][j][p], min)
minFlowOn = flowOns[j]
}
}
flowOnDises[i][p] = minFlowOn
}
}
log.Println("--------------calcAllLocalDisagreementCosts-----------------------------------")
ttDis, queuerampDis := o.calcAllLocalDisagreementCosts(needOptimization, flowOnDises, o.flowOnHat)
log.Println("--------------calcAllLocalCosts-----------------------------------")
ttLocalWithVariable, queuerampWithVariable := o.calcAllLocalCosts(needOptimization, flowOns, o.flowOnHat)
ttLocal, selectedFlowOns := o.optimizeLocalCost(ttLocalWithVariable, queuerampWithVariable, ttDis,
needOptimization, flowOns)
// Set for values for the next iteration by saving them on disk
for r := 0; r < o.detsGroupCount; r++ {
if i, ok := intInSlice(r, needOptimization); ok {
wasAbleToOptimize := true
sigmaTTDis := float64(0)
for p := 0; p < o.msa; p++ {
if selectedFlowOns[i][p] == -1 {
wasAbleToOptimize = false
}
sigmaTTDis += ttDis[i][p]
}
if wasAbleToOptimize {
o.ttDisHat[r] = sigmaTTDis - o.alpha*(sigmaTTDis-ttLocal[i])
o.ttLocalHat[r] = ttLocal[i]
o.flowOnHat[r] = int(flowOns[selectedFlowOns[i][0]])
} else {
o.ttDisHat[r] = ttLocal[i]
o.ttLocalHat[r] = ttLocal[i]
//o.flowOnHat[r][p] will remain as before
}
} else {
// keep the previous values
}
}
for i := range needOptimization {
// We always use 0 for decision making
rate := o.flowOnHat[needOptimization[i]]
if o.dets[o.consts[needOptimization[i]].mainKey].BotSpeedXHat[o.t] <= o.VSyn {
o.d.Rate[o.consts[needOptimization[i]].mainKey] = rate
} else {
```

```
o.d.AlwaysGreen[o.consts[needOptimization[i]].mainKey] = true
}
}
o.save(needOptimization, pb, flowOns, flowOnDises, ttDis, queuerampDis, ttLocalWithVariable,
queuerampWithVariable, selectedFlowOns)
} else {
// it is already set to be always green
// reseting the hat values
for r := 0; r < o.detsGroupCount; r++ {
for p := 0; p < o.msa; p++ {
o.ttDisHat[r] = -1
o.ttLocalHat[r] = -1
o.flowOnHat[r] = -1
}
}
o.save(needOptimization, nil, flowOns, nil, nil, nil, nil, nil, nil)
}
}
func fileExists(filename string) bool {
info, err := os.Stat(filename)
if os.IsNotExist(err) {
return false
}
return !info.IsDir()
}
func (o *optimizer) save(needOptimization []int, pb [][][]float64, flowOns []float64, flowOnDises [][]float64,
ttDis, queuerampDis [][]float64,
ttLocalWithVariables, queuerampWithVariables [][][]float64,
selectedFlowOns [][]int,
) {
os.Mkdir(o.outDir, 0755)
fileNamePattern := filepath.Join(o.outDir, "optimize-detect-%s-%s.csv")
for r := 0; r < o.detsGroupCount; r++ {
var b []byte
buf := bytes.NewBuffer(b)
fileName := fmt.Sprintf(fileNamePattern, o.consts[r].mainKey, o.consts[r].rampKey)
if !fileExists(fileName) {
o.writeHeader(buf, flowOns)
} else {
prv, err := ioutil.ReadFile(fileName)
if err != nil {
log.Fatalf("could not read the previous file %s: %v", fileName, err)
}
buf.Write(prv)
}
// --------- new content
// This file will be read by controllers in the next timestep
// Therefore, we increase the timestep to make reading easier
_, _ = buf.WriteString(fmt.Sprintf("%d", o.t+1))
if i, ok := intInSlice(r, needOptimization); ok {
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(fmt.Sprintf(",%.2f", pb[i][j][p]))
}
}
for p := 0; p < o.msa; p++ {
```

```
buf.WriteString(fmt.Sprintf(",%d", int(flowOnDises[i][p])))
}
for p := 0; p < o.msa; p++ {
buf.WriteString(fmt.Sprintf(",%.2f", ttDis[i][p]))
}
for p := 0; p < o.msa; p++ {
buf.WriteString(fmt.Sprintf(",%.2f", queuerampDis[i][p]))
}
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(fmt.Sprintf(",%.2f", ttLocalWithVariables[i][j][p]))
}
}
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(fmt.Sprintf(",%.2f", queuerampWithVariables[i][j][p]))
}
}
for p := 0; p < o.msa; p++ {
index := selectedFlowOns[i][p]
if index == -1 {
buf.WriteString(fmt.Sprintf(",%d", index))
} else {
buf.WriteString(fmt.Sprintf(",%.2f", flowOns[index]))
}
}
} else {
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(",-")
}
}
for p := 0; p < o.msa; p++ {
buf.WriteString(",-")
}
for p := 0; p < o.msa; p++ {
buf.WriteString(",-")
}
for p := 0; p < o.msa; p++ {
buf.WriteString(",-")
}
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(",-")
}
}
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(",-")
}
}
for p := 0; p < o.msa; p++ {
buf.WriteString(",-")
}
}
buf.WriteString(fmt.Sprintf(",%.2f", o.ttDisHat[r]))
buf.WriteString(fmt.Sprintf(",%.2f", o.ttLocalHat[r]))
buf.WriteString(fmt.Sprintf(",%d", o.flowOnHat[r]))
buf.WriteString(fmt.Sprintf(",%v,%d\n", o.d.AlwaysGreen[o.consts[r].mainKey], o.d.Rate[o.consts[r].mainKey]))
if err := ioutil.WriteFile(fileName, buf.Bytes(), 0755); err != nil {
log.Fatalf("could not write file %s: %v", fileName, err)
}
}
}
func (o *optimizer) writeHeader(buf *bytes.Buffer, flowOns []float64) {
_, _ = buf.WriteString("Step")
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(fmt.Sprintf(",PB[t+%d/t]-flow-%d", p+1, int(flowOns[j])))
}
}
for p := 0; p < o.msa; p++ {
buf.WriteString(fmt.Sprintf(",flowon-dis[t+%d/t]", p+1))
}
for p := 0; p < o.msa; p++ {
buf.WriteString(fmt.Sprintf(",TTdis[t+%d/t]", p+1))
}
for p := 0; p < o.msa; p++ {
buf.WriteString(fmt.Sprintf(",queueramp-dis[t+%d/t]", p+1))
}
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(fmt.Sprintf(",TTlocal[t+%d/t]-flow-%d", p+1, int(flowOns[j])))
}
}
for p := 0; p < o.msa; p++ {
for j := 0; j < len(flowOns); j++ {
buf.WriteString(fmt.Sprintf(",queueramp[t+%d/t]-flow-%d", p+1, int(flowOns[j])))
}
}
for p := 0; p < o.msa; p++ {
buf.WriteString(fmt.Sprintf(",selectedFlows[t+%d/t]", p+1))
}
buf.WriteString(",TTdishat")
buf.WriteString(",TTlocalhat")
buf.WriteString(",Flowonhat")
buf.WriteString(",always-green,rate\n")
}
func (o *optimizer) optimizeLocalCost(ttLocalWithVariable [][][]float64, queuerampWithVariable [][][]float64,
ttDis [][]float64, needOptimization []int, flowOns []float64) ([]float64
ttLocal := make([]float64, len(needOptimization))
selectedFlowOns := make([][]int, len(needOptimization))
for i := range needOptimization {
r := needOptimization[i]
r1 := o.consts[r].otherDets[0]
r2 := o.consts[r].otherDets[1]
selectedFlowOns[i] = make([]int, o.msa)
sigmaTTDis := float64(0)
sigmaTTLocalHat1 := o.ttLocalHat[r1]
sigmaTTDisHat1 := o.ttDisHat[r1]
sigmaTTLocalHat2 := o.ttLocalHat[r2]
sigmaTTDisHat2 := o.ttDisHat[r2]
```

```
for p := 0; p < o.msa; p++ {
sigmaTTDis += ttDis[i][p]
}
fixed1 := float64(0)
if _, ok := intInSlice(r1, needOptimization); ok {
if sigmaTTDisHat1 == sigmaTTLocalHat1 {
fixed1 = o.w[r1]
} else {
fixed1 = o.w[r1] * math.Log10(sigmaTTDisHat1-sigmaTTLocalHat1)
}
}
fixed2 := float64(0)
if _, ok := intInSlice(r2, needOptimization); ok {
if sigmaTTDisHat2 == sigmaTTLocalHat2 {
fixed2 = o.w[r2]
} else {
fixed2 = o.w[r2] * math.Log10(sigmaTTDisHat2-sigmaTTLocalHat2)
}
}
total := float64(-1)
selectedFlowOns[i] = []int{-1, -1, -1, -1, -1}
start := time.Now()
log.Printf("Started permutation at %v", start)
for j0 := 0; j0 < 7; j0++ {
for j1 := 0; j1 < 7; j1++ {
for j2 := 0; j2 < 7; j2++ {
for j3 := 0; j3 < 7; j3++ {
for j4 := 0; j4 < 7; j4++ {
sigmaTTLocalWithVariable := ttLocalWithVariable[i][j0][0] + ttLocalWithVariable[i][j1][1] +
ttLocalWithVariable[i][j2][2] + ttLocalWithVariable[i][j3][3] + ttLocalWithVariable
variable := o.w[r]
if sigmaTTDis != sigmaTTLocalWithVariable {
variable *= math.Log10(sigmaTTDis - sigmaTTLocalWithVariable)
}
// compute maximum
if sigmaTTDis >= sigmaTTLocalWithVariable && queuerampWithVariable[i][j0][0] >= 0 &&
queuerampWithVariable[i][j0][0] <= o.queuerampMax[r] && queuerampWithVariable[i][j1][1] >=
log.Printf("----> fixed1: %.2f, fixed2: %.2f\n", fixed1, fixed2)
if variable+fixed1+fixed2 > total {
total = variable + fixed1 + fixed2
selectedFlowOns[i][0] = j0
selectedFlowOns[i][1] = j1
selectedFlowOns[i][2] = j2
selectedFlowOns[i][3] = j3
selectedFlowOns[i][4] = j4
ttLocal[i] = sigmaTTLocalWithVariable
}
}
}
}
}
}
}
log.Printf("-------> took: %v to do the maximization\n", time.Since(start).Nanoseconds())
}
return ttLocal, selectedFlowOns
```

```
}
func intInSlice(a int, list []int) (int, bool) {
for j, b := range list {
if b == a {
return j, true
}
}
return -1, false
}
func (o *optimizer) calcAllProbabilityOfBreakdown(needOptimization []int, flowOns []float64) ([][][]float64,
[][][]float64) {
grid := make([][][]float64, len(needOptimization))
queueramp := make([][][]float64, len(needOptimization))
for i := range needOptimization {
kMain := o.consts[needOptimization[i]].mainKey
alpha := o.consts[needOptimization[i]].alpha
beta := o.consts[needOptimization[i]].beta
grid[i] = make([][]float64, len(flowOns))
queueramp[i] = make([][]float64, len(flowOns))
for j, flowOn := range flowOns {
kRamp := o.consts[needOptimization[i]].rampKey
grid[i][j], queueramp[i][j] = o.calcProbabilityOfBreakdown(kMain, alpha, beta, kRamp, flowOn)
}
}
return grid, queueramp
}
func (o *optimizer) calcProbabilityOfBreakdown(kMain string, alpha, beta float64, kRamp string, flowOn float64)
([]float64, []float64) {
pb := make([]float64, o.msa)
queueramp := make([]float64, o.msa)
for p := range pb {
flowBot := o.dets[kMain].FlowBotMultiAhead[o.t][p]
val := (flowBot + flowOn) / beta
term1 := 1.0 // alpha * math.Pow(val, alpha-1) / beta
term2 := math.Exp(-(math.Pow(val, alpha)))
//pb[p] = term1 * term2
pb[p] = term1 - term2
log.Printf("----> PB[%.2f]: flowBot: %.2f, val: %.2f, term2: %.2f, pb: %.2f\n", flowOn, flowBot, val, term2, pb[p])
flowAllRamp := o.dets[kRamp].FlowAllMultiAhead[o.t][p]
queueramp[p] = (flowAllRamp - flowOn) / float64(60)
if queueramp[p] < 0 {
queueramp[p] = 0
}
queueramp[p] = math.Ceil(queueramp[p])
}
return pb, queueramp
}
func (o *optimizer) calcAllLocalDisagreementCosts(needOptimization []int, flowOnDises [][]float64,
optimizedFlowOnHat []int) ([][]float64, [][]float64) {
ttDis := make([][]float64, len(needOptimization))
queuerampDis := make([][]float64, len(needOptimization))
for i := range needOptimization {
kMain := o.consts[needOptimization[i]].mainKey
kRamp := o.consts[needOptimization[i]].rampKey
ttDis[i] = make([]float64, o.msa)
queuerampDis[i] = make([]float64, o.msa)
```

```
ttDis[i], queuerampDis[i] = o.calcLocalCost(kMain, kRamp, flowOnDises[i], float64(optimizedFlowOnHat[i]))
}
return ttDis, queuerampDis
}
func (o *optimizer) calcAllLocalCosts(needOptimization []int, flowOns []float64, optimizedFlowOnHat []int)
([][][]float64, [][][]float64) {
ttLocal := make([][][]float64, len(needOptimization))
queueramp := make([][][]float64, len(needOptimization))
for i := range needOptimization {
kMain := o.consts[needOptimization[i]].mainKey
kRamp := o.consts[needOptimization[i]].rampKey
ttLocal[i] = make([][]float64, len(flowOns))
queueramp[i] = make([][]float64, len(flowOns))
for j, flowOn := range flowOns {
flowOnSame := make([]float64, o.msa)
for p := range flowOnSame {
flowOnSame[p] = flowOn
}
ttLocal[i][j], queueramp[i][j] = o.calcLocalCost(kMain, kRamp, flowOnSame, float64(optimizedFlowOnHat[i]))
}
}
return ttLocal, queueramp
}
func (o *optimizer) calcLocalCost(kMain, kRamp string, flowOn []float64, optimizedFlowOnHat float64)
([]float64, []float64) {
ttLocal := make([]float64, o.msa)
queueramp := make([]float64, o.msa)
for p := range ttLocal {
averageTT := o.dets[kMain].TravelAVGMultiAhead[o.t][p]
flowAllMain := o.dets[kMain].FlowAllMultiAhead[o.t][p]
flowAllRamp := o.dets[kRamp].FlowAllMultiAhead[o.t][p]
queueramp[p] = (flowAllRamp - flowOn[p]) / float64(60)
if queueramp[p] < 0 {
queueramp[p] = 0
}
queueramp[p] = math.Ceil(queueramp[p])
ttLocal[p] = averageTT*(flowAllMain)/float64(60*60) + queueramp[p] +
float64(o.penalizeDeltaFlowOn)*(math.Pow(flowOn[p]-optimizedFlowOnHat, 2))
log.Printf("-----------> [%.2f]: averageTT: %.2f, flowAllMain: %.2f flowAllRamp: %.2f, queueramp: %.2f, part1:
%.2f, part2: %.2f, penalized: %d, optimizedFlowOnHat: %.2f\n", flowOn[
}
return ttLocal, queueramp
}
func (f *flowSim) Save(dir string) []string {
start := time.Now()
for _, d := range f.detectors {
report := d.Report()
for _, r := range report {
ioutil.WriteFile(fmt.Sprintf("%s/report-detector-%d-%s.csv", dir, d.ID(), r.Name), []byte(r.Content), 0755)
}
}
f.timing["save"] += time.Since(start).Nanoseconds()
log.Println("----------- Timing for second simulation:")
f.timing["report"] += time.Since(start).Nanoseconds()
for k, v := range f.timing {
log.Printf("%s:\t%.1f sec\n", k, float64(v)/float64(1000000000))
}
log.Println("----------------------------------------")
return nil
}
func (s *flowSim) SetOtherSimulator(callableSecondSim sim.SetSimFunc, conf2 sim.Config) {
// not needed
}
#-----------------------NEW FILE--------------------------- sim/kalman/kalman.go
package kalman
import (
"fmt"
"log"
"math"
)
// State holds the KF apriori state
type State struct {
X float64 // real
Y float64 // noisy
P float64
XHat float64 // estimate of real
rHat float64
RHat float64
qHat float64
QHat float64
K float64
xHatTT float64
pTT float64
nextXHat float64
nextP float64
StepAheadStates []float64
}
var negligible = 0.01
func (s *State) String() string {
base := fmt.Sprintf("%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f", s.P, s.K, s.pTT, s.XHat, s.xHatTT,
s.nextXHat, s.nextP, s.rHat, s.RHat, s.qHat, s.QHat)
if len(s.StepAheadStates) == 0 {
return base
}
extra := ""
for _, a := range s.StepAheadStates {
extra = fmt.Sprintf("%s,%.2f", extra, a)
}
return fmt.Sprintf("%s,%s", base, extra)
}
func (s *State) NextXHat() float64 {
return s.nextXHat
}
// Step moves the simulation one step forward
func Step(past []*State, next *State) []*State {
newState := make([]*State, len(past))
copy(newState[:], past[1:])
newState[len(past)-1] = next
return newState
}
// SimplePredict computes the apriori estimate
func SimplePredict(pastStates []*State, N int, Qtt, Rtt float64, stepAheadCount int) State {
```

```
if len(pastStates) != N+1 {
log.Fatal("Need ", N+1, " levels, got ", len(pastStates))
}
curr := pastStates[len(pastStates)-1]
curr.rHat = float64(0)
for i := 1; i < len(pastStates); i++ {
s := pastStates[i]
curr.rHat += s.Y - s.X
}
curr.rHat *= float64(1) / float64(N)
if curr.rHat == 0 {
curr.rHat = negligible
}
curr.RHat = Rtt
curr.K = curr.P * (float64(1) / (curr.P + curr.RHat))
curr.pTT = (1 - curr.K) * curr.P
curr.qHat = float64(0)
for i := 1; i < len(pastStates); i++ {
s := pastStates[i]
st1 := pastStates[i-1]
curr.qHat += s.XHat - st1.XHat
}
curr.qHat *= float64(1) / float64(N)
if curr.qHat == 0 {
curr.qHat = negligible
}
curr.QHat = Qtt
curr.xHatTT = curr.XHat + curr.K*(curr.Y-curr.XHat)
curr.nextP = curr.pTT + curr.QHat
curr.nextXHat = curr.xHatTT + curr.qHat
if curr.nextXHat < 0 {
curr.nextXHat = 0
}
prevXhats := make([]float64, N+1+stepAheadCount)
for i := 0; i < N+1; i++ {
prevXhats[i] = pastStates[i].xHatTT
}
prevXhats[N+1] = curr.nextXHat
curr.StepAheadStates = make([]float64, stepAheadCount+2)
curr.StepAheadStates[0] = curr.xHatTT
curr.StepAheadStates[1] = curr.nextXHat
qHatMulti := float64(0)
for i := N + 1; i < len(prevXhats); i++ {
// calculet new qHat from prevXhats
qHatMultiNew := float64(0)
for j := 0; j < N; j++ {
qHatMultiNew += prevXhats[i-j] - prevXhats[i-j-1]
}
qHatMultiNew *= float64(1) / float64(N)
if qHatMultiNew == 0 {
qHatMultiNew = negligible
}
qHatMulti += qHatMultiNew
curr.StepAheadStates[i-N+1] = curr.nextXHat + qHatMulti
if i+1 != len(prevXhats) {
prevXhats[i+1] = curr.StepAheadStates[i-N+1]
}
}
next := State{
P: curr.nextP,
XHat: curr.nextXHat,
}
return next
}
// MultiStepAheadPredict computes the apriori estimate
func MultiStepAheadPredict(pastStates []*State, N int, stepAheadCount int) State {
if len(pastStates) != N+1 {
log.Fatal("Need ", N+1, " levels, got ", len(pastStates))
}
curr := pastStates[len(pastStates)-1]
curr.rHat = float64(0)
for i := 1; i < len(pastStates); i++ {
s := pastStates[i]
curr.rHat += s.Y - s.X
}
curr.rHat *= float64(1) / float64(N)
if curr.rHat == 0 {
curr.rHat = negligible
}
curr.RHat = float64(0)
for i := 1; i < len(pastStates); i++ {
s := pastStates[i]
curr.RHat += math.Pow((s.Y - s.X - curr.rHat), 2)
//curr.RHat -= ((float64(N-1) / float64(N)) * s.P)
}
curr.RHat *= float64(1) / float64(N-1)
if curr.RHat == 0 {
curr.RHat = negligible
}
curr.K = curr.P * (float64(1) / (curr.P + curr.RHat))
curr.pTT = (1 - curr.K) * curr.P
curr.qHat = float64(0)
for i := 1; i < len(pastStates); i++ {
s := pastStates[i]
st1 := pastStates[i-1]
curr.qHat += s.X - st1.X
}
curr.qHat *= float64(1) / float64(N)
if curr.qHat == 0 {
curr.qHat = negligible
}
curr.QHat = float64(0)
for i := 1; i < len(pastStates); i++ {
s := pastStates[i]
st1 := pastStates[i-1]
curr.QHat += math.Pow((s.X - st1.X - curr.qHat), 2)
}
curr.QHat *= float64(1) / float64(N-1)
if curr.QHat == 0 {
curr.QHat = negligible
}
curr.xHatTT = curr.XHat + curr.K*(curr.Y-curr.XHat)
curr.nextP = curr.pTT + curr.QHat
curr.nextXHat = curr.xHatTT + curr.qHat
if curr.nextXHat < 0 {
curr.nextXHat = 0
}
prevXhats := make([]float64, N+1+stepAheadCount)
for i := 0; i < N+1; i++ {
prevXhats[i] = pastStates[i].xHatTT
}
prevXhats[N+1] = curr.nextXHat
curr.StepAheadStates = make([]float64, stepAheadCount+2)
curr.StepAheadStates[0] = curr.xHatTT
curr.StepAheadStates[1] = curr.nextXHat
qHatMulti := float64(0)
for i := N + 1; i < len(prevXhats); i++ {
// calculet new qHat from prevXhats
qHatMultiNew := float64(0)
for j := 0; j < N; j++ {
qHatMultiNew += prevXhats[i-j] - prevXhats[i-j-1]
}
qHatMultiNew *= float64(1) / float64(N)
if qHatMultiNew == 0 {
qHatMultiNew = negligible
}
qHatMulti += qHatMultiNew
curr.StepAheadStates[i-N+1] = curr.nextXHat + qHatMulti
if i+1 != len(prevXhats) {
prevXhats[i+1] = curr.StepAheadStates[i-N+1]
}
}
next := State{
P: curr.nextP,
XHat: curr.nextXHat,
}
return next
}
#----------------------NEW FILE-------------------------- sim/singleton_seed.go
package sim
var HasPresetSeeds bool = false
var CarSeed int64
var ChangeSeed int64
var GeneratorSeed int64
// controller, detector, kalman, lane, placer, simulator do not have seed
```