Developing Analysis, Modeling, and Simulation Tools for Connected and Automated Vehicle Applications

Algorithm Description Document: Incorporating Connected Manually Driven Vehicle Model and Variable Speed Advisory into an Improved Cooperative Adaptive Cruise Control Algorithm

August 2020

U.S. Department of Transportation
Federal Highway Administration

Research, Development, and Technology
Turner-Fairbank Highway Research Center
6300 Georgetown Pike
McLean, VA 22101-2296

Notice

This document is disseminated under the sponsorship of the U.S. Department of Transportation (USDOT) in the interest of information exchange. The U.S. Government assumes no liability for the use of the information contained in this document.

The U.S. Government does not endorse products or manufacturers. Trademarks or manufacturers' names appear in this report only because they are considered essential to the objective of the document.

Quality Assurance Statement

The Federal Highway Administration (FHWA) provides high-quality information to serve Government, industry, and the public in a manner that promotes public understanding. Standards and policies are used to ensure and maximize the quality, objectivity, utility, and integrity of its information. FHWA periodically reviews quality issues and adjusts its programs and processes to ensure continuous quality improvement.

TECHNICAL REPORT DOCUMENTATION PAGE

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle Developing Analysis, Modeling, and Simulation Tools for Connected and Automated Vehicle Applications Algorithm Description Document: Incorporating Connected Manually Driven Vehicle Model and Variable Speed Advisory into an Improved Cooperative Adaptive Cruise Control Algorithm		**5. Report Date** August 2020 **6. Performing Organization Code:**	
7. Author(s) Hao Liu, Soomin Woo, Xiao-Yun Lu, and Zhitong Huang		**8. Performing Organization Report No.**	
9. Performing Organization Name and Address Leidos Inc. 11251 Roger Bacon Drive Reston, VA 20190		**10. Work Unit No.** **11. Contract or Grant No.** DTFH61-12-D-00030, TO 22	
12. Sponsoring Agency Name and Address Office of Operations Research and Development Federal Highway Administration 6300 Georgetown Pike McLean, VA 22101-2296		**13. Type of Report and Period Covered** **14. Sponsoring Agency Code**	
15. Supplementary Notes The government task managers were John Halkias and Gene McHale.			
16. Abstract This report details the implementation of the proposed connected manually driven vehicles (CMDVs) model in a simulation framework that also has a calibrated human-driven vehicles (HVs), autonomous vehicles (AVs), and connected automated vehicles (CAVs) model. The results detailed in this report were obtained using a microscopic traffic simulation platform. However, the model logics are open source and described in detail in the appendices. The information in this document may be helpful for researchers and analysts to implement the modeling framework in their customized tools of choice.			
17. Key Words Connected and automated vehicle, traffic simulation, microsimulation, cooperative adaptive cruise control, market penetration rate, variable speed advisory		**18. Distribution Statement** No restrictions.	
19. Security Classif. (of this report) Unclassified	**20. Security Classif. (of this page)** Unclassified	**21. No. of Pages** 162	**22. Price** N/A

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized.

SI* (MODERN METRIC) CONVERSION FACTORS

APPROXIMATE CONVERSIONS TO SI UNITS

Symbol	When You Know	Multiply By	To Find	Symbol
		LENGTH		
in	inches	25.4	millimeters	mm
ft	feet	0.305	meters	m
yd	yards	0.914	meters	m
mi	miles	1.61	kilometers	km
		AREA		
in^2	square inches	645.2	square millimeters	mm^2
ft^2	square feet	0.093	square meters	m^2
yd^2	square yard	0.836	square meters	m^2
ac	acres	0.405	hectares	ha
mi^2	square miles	2.59	square kilometers	km^2
		VOLUME		
fl oz	fluid ounces	29.57	milliliters	mL
gal	gallons	3.785	liters	L
ft^3	cubic feet	0.028	cubic meters	m^3
yd^3	cubic yards	0.765	cubic meters	m^3
		NOTE: volumes greater than 1,000 L shall be shown in m^3		
		MASS		
oz	ounces	28.35	grams	g
lb	pounds	0.454	kilograms	kg
T	short tons (2,000 lb)	0.907	megagrams (or "metric ton")	Mg (or "t")
		TEMPERATURE (exact degrees)		
°F	Fahrenheit	5 (F-32)/9 or (F-32)/1.8	Celsius	°C
		ILLUMINATION		
fc	foot-candles	10.76	lux	lx
fl	foot-Lamberts	3.426	candela/m^2	cd/m^2
		FORCE and PRESSURE or STRESS		
lbf	poundforce	4.45	newtons	N
lbf/in^2	poundforce per square inch	6.89	kilopascals	kPa

APPROXIMATE CONVERSIONS FROM SI UNITS

Symbol	When You Know	Multiply By	To Find	Symbol
		LENGTH		
mm	millimeters	0.039	inches	in
m	meters	3.28	feet	ft
m	meters	1.09	yards	yd
km	kilometers	0.621	miles	mi
		AREA		
mm^2	square millimeters	0.0016	square inches	in^2
m^2	square meters	10.764	square feet	ft^2
m^2	square meters	1.195	square yards	yd^2
ha	hectares	2.47	acres	ac
km^2	square kilometers	0.386	square miles	mi^2
		VOLUME		
mL	milliliters	0.034	fluid ounces	fl oz
L	liters	0.264	gallons	gal
m^3	cubic meters	35.314	cubic feet	ft^3
m^3	cubic meters	1.307	cubic yards	yd^3
		MASS		
g	grams	0.035	ounces	oz
kg	kilograms	2.202	pounds	lb
Mg (or "t")	megagrams (or "metric ton")	1.103	short tons (2,000 lb)	T
		TEMPERATURE (exact degrees)		
°C	Celsius	1.8C+32	Fahrenheit	°F
		ILLUMINATION		
lx	lux	0.0929	foot-candles	fc
cd/m^2	candela/m2	0.2919	foot-Lamberts	fl
		FORCE and PRESSURE or STRESS		
N	newtons	2.225	poundforce	lbf
kPa	kilopascals	0.145	poundforce per square inch	lbf/in^2

*SI is the symbol for International System of Units. Appropriate rounding should be made to comply with Section 4 of ASTM E380. (Revised March 2003)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACC	adaptive cruise control
ACF	after lane-changing car-following
ALC	active lane-changing
AMS	analysis, modeling, and simulation
API	Application Programming Interface
ATM	Active Traffic Management
AV	autonomous vehicle
BCF	before lane-changing car-following
BM	behavior model
CACC	cooperative adaptive cruise control
Caltrans	California Department of Transportation
CAV	connected and automated vehicle
CF	car-following
CMS	changeable message sign
CRM	coordinated ramp metering
CV	connected vehicle
DLC	discretionary lane-change
ECDF	empirical cumulative distribution function
FHWA	Federal Highway Administration
GEH	Geoffrey E. Havers (statistic)
HOV	high occupancy vehicle
HV	human-driven vehicle
I2V	infrastructure-to-vehicle
LC	lane-changing
LRRM	local responsive ramp metering
MAPE	mean absolute percentage error
ML	managed lane
MLC	mandatory lane-changing
MOVES	motor vehicle emission simulator
MPG	miles per gallon
MPH	miles per hour

MPR	market penetration rate
MOTUS	microscopic open traffic simulation
NGSIM	Next Generation Simulation
O-D	origin-destination
PATH	California Partners for Advanced Transportation Technology
PeMS	Performance Measurement System
RCF	receiving car-following
RM	ramp meter
SV	simulated vehicle
TMC	traffic management center
V2V	vehicle-to-vehicle
VAD	vehicle awareness device
VHT	vehicle hours traveled
VMT	vehicle miles traveled
VSA	variable speed advisory
VTT	vehicle time traveled
YCF	yielding car-following

EXECUTIVE SUMMARY

In the next few decades, traffic streams might consist of human-driven vehicles (HVs), connected manually driven vehicles (CMDVs), autonomous vehicles (AVs), and connected automated vehicles (CAVs) at the same time. The interaction of various types of vehicle fleets may induce complex traffic flow patterns that have never been observed in the existing transportation system. Such complex traffic is difficult to model by existing traffic simulation and evaluation approaches, thus bringing about great uncertainty as transportation stakeholders attempt to improve system performance via new technologies. This, in turn, leads to difficulties in the development of Active Traffic Management (ATM) strategies. To address these challenges, the Federal Highway Administration (FHWA) supported a research project entitled "Developing Analysis, Modeling, and Simulation (AMS) Tools for Connected Automated Vehicle Applications." Within the project scope, the research team has developed and integrated a CMDV model into its existing AMS framework.

A CMDV is a manually-driven vehicle in which the human driver's behavior could be affected by infrastructure-to-vehicle (I2V) advisory speeds from traffic management centers (TMCs). This research adopts the stimulus-response paradigm to model the behavior adaptation of CMDV drivers in the traffic stream. This modeling approach quantifies the reaction of drivers after the vehicle's on-board system gives the advisory speed generated by the variable speed advisory (VSA) control. The approach incorporates the CMDV-affected speed behavior parameter into a state-of-the-art microscopic car-following model as factors depicting drivers' response sensitivity to traffic stimuli. To demonstrate the effectiveness of a CMDV on the traffic flow, this study implemented the CMDV model with a feedback VSA algorithm. The algorithm generates advisory speeds lower than the posted speed limit to cause vehicles upstream from the bottleneck section to reduce their traveling speeds, thus decreasing the input flow to the bottleneck and leading to the recovery of the bottleneck congestion.

The effectiveness of a CMDV is affected by the driver's compliance with the VSA. The CMDV system cannot impact the traffic flow unless the driver follows its instructions. The ability and willingness to follow instructions vary from driver to driver. In this study, the driver speed compliance and its variation were determined based on field test data and an empirical model. VSA field test data was used for model calibration and validation to allow CMDV drivers' speed adaptations to represent real-world drivers' speed patterns under the influence of VSA control. CMDV drivers' compliance was found to be best modeled with empirical distributions of two speed groups: a low speed group with the VSA under 35 mph, and a high speed group with the VSA above 35 mph. The calibration and validation process adopted distribution fitting and hypothesis test approaches to determine the empirical distribution parameters. The two-sample Kolmogorov-Smirnov test was conducted to show the goodness of fit of the empirical cumulative distribution functions with real-world data. The validated model has been applied to evaluate the effectiveness of CMDV on freeway corridor operations.

It should be noted that the model calibration and validation datasets represent drivers' response to the VSA displayed on the roadside message sign rather than the direct feedback in the vehicle. Admittedly, such feedback will have less influence on the driver than in-vehicle displays. For this reason, this research designed a sensitivity analysis to explore the effectiveness of the CMDV speed adaptation under various CMDV market penetrations and driver compliance

levels. In addition, the sensitivity analysis has identified the impact of using automated speed controllers with the I2V-based VSA algorithm.

The analysis results indicate that the I2V-based VSA control could have substantial effects on the freeway corridor when the CMDV market penetration is 10–40 percent. With the advisory speed, the average speed and variation of the speed remained similar to the no-control case but the vehicle fuel efficiency increased 2–5 percent, depending on the results of different energy models. These results suggest that the speed adaptation of a few connected drivers could substantially change the traffic flow pattern, leading to more energy efficient traffic flow. As the CMDV market penetration further increased, the reduction of the speed variation and the improvement of the fuel efficiency stabilized. When the CMDV market penetration reached 100 percent, the average speed decreased by about 1 percent and the vehicle fuel economy increased 5–6 percent.

The performance of the VSA algorithm was not sensitive to small changes in the driver compliance level. However, the traffic flow patterns changed significantly when the CMDV drivers fully complied with the VSA. The full compliance brought about 2–3 percent extra benefit on vehicle fuel efficiency. If the VSA algorithm could generate advisory speed based on the predicted traffic conditions, the effects of CMDV should become further increased. When the VSA was implemented with CACC, the CACC controller could perfectly adopt the advisory speed as the reference speed. Nonetheless, adding the VSA algorithm to the CACC vehicles did not bring notable benefits to the freeway corridor (e.g., 5–6 percent increase of the energy efficiency) because the VSA controller tended to underuse the bottleneck capacity due to its delayed response to the traffic variations. To address this shortcoming, the team recommends a predictive VSA algorithm or the combined application of VSA and RM.

This report details the implementation of the proposed CMDV model in a simulation framework that also has a calibrated HV, AV, and CAV model. The results detailed in this report were obtained using the Aimsun modeling platform. However, the model logics are open source and described in detail in the appendices. The information in this document may be helpful for researchers and analysts to implement the modeling framework in their customized tools of choice.

This research has been performed under a FHWA project entitled “Developing Analysis, Modeling, and Simulation Tools for Connected and Automated Vehicle Applications” (contract number: DTFH6116D00030-0022). To get more information of this FHWA project, readers are encouraged to reference the final project report of this project (Lu et al, forthcoming). This report is under FHWA publication process and it will be available soon.

CHAPTER 1. PURPOSE OF THIS MODEL

PURPOSE OF THIS DOCUMENT

Connected and automated vehicle (CAV) technologies offer potentially transformative societal impacts, including significant mobility, safety, and environmental benefits. State and local agencies are interested in harnessing the potential benefits of CAVs. However, for agencies to be able to plan beneficial deployments of infrastructure-to-vehicle (I2V) and vehicle-to-vehicle (V2V) technology, it is important to be able to robustly predict the impacts of such deployments and identify which applications best address their unique transportation problems. Traffic analysis, modeling, and simulation (AMS) tools provide an efficient means to evaluate transportation improvement projects before deployment.

However, current AMS tools are not well suited for evaluating CAV applications due to their inability to represent vehicle connectivity and automated driving features. The development of a new generation of tools involves spending a lot of resources and time to develop, calibrate, and validate. Many independent researchers have developed models of CAV systems based on a divergent array of underlying assumptions. As a result, there is little consensus in the literature regarding the most likely impacts of CAV technologies.

Thus, there is a desire for a consistent set of models to produce realistic and believable predictions of CAV impacts. These models can be based on the best available data and include the most accurate possible representations of the behaviors of drivers of conventional vehicles and CAVs. Deployment concepts, strategies, and guidelines are also key for allowing State and local agencies to understand how and where to deploy CAV technologies.

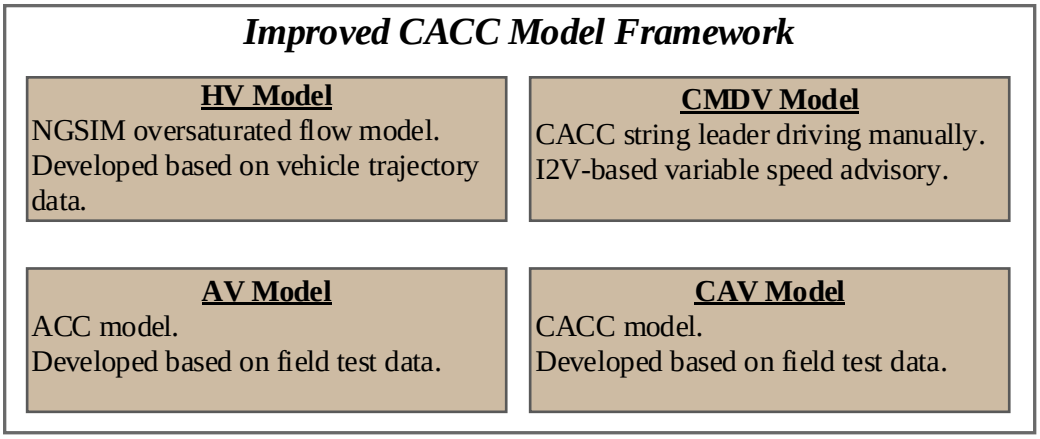
To meet these goals, the Federal Highway Administration (FHWA) sponsored a project entitled 'Developing Analysis, Modeling, and Simulation Tools for Connected and Automated Vehicle Applications' (contract number: DTFH6116D00030-0022). This project aimed to develop AMS models for the most prominent CAV applications and to incorporate these models into existing AMS simulation tools. Three CAV applications were developed under this project: a lane changing (LC) model for light duty CAVs, a combined application model that integrates speed harmonization (SH) and coordinated merge (CM), and an improved cooperative adaptive cruise control (CACC) model for light duty CAVs. The final project report of this project (Lu et al, forthcoming) is under FHWA publication process and it will be available soon.

This document presents an improved CACC model for light duty CAVs, with the objective of providing detailed information on this model to benefit the CAV simulation community. This document is expected to help future users easily adopt and customize the improved CACC model in a traffic simulation tool to meet their simulation needs. To this end, this document provides detailed descriptions of the newly-developed connected manually driven vehicle (CMDV) model, the model calibration and validation, and the sensitivity analysis of the model behaviors. In addition, the model implementation in the microscopic traffic simulation tool is described to help readers implement the model in various simulation platforms. Descriptions and pseudo code of the model functions are also provided as a resource.

PURPOSE OF THIS MODEL

In the next few decades, the traffic stream is likely to be mixed with human-driven vehicles (HVs), connected manually driven vehicles (CMDVs), autonomous vehicles (AVs), and connected and automated vehicles (CAVs) all operating simultaneously. The interaction of various types of vehicle fleets may induce complex traffic flow patterns that do not exist in the existing transportation system. Such complex traffic is difficult to model with existing traffic simulation and evaluation approaches, thus bringing about significant uncertainty as transportation stakeholders attempt to improve the system performance via the new technologies. This lack of understanding of future traffic flow patterns leads to difficulties in the development of active traffic management (ATM) strategies. To address the challenge, the project team developed a CMDV model and integrated the model into its existing AMS framework. Because CV, in a general sense, would include manually driven vehicles and automatically driven vehicles, such as CAV, the research team used CMDV to specifically represent the connected but manually driven vehicle.

To test the CMDV model, the project team adopted a previously developed simulation framework for modeling mixed traffic: the California PATH simulation framework (referred to as the simulation framework or model framework in the remainder of this chapter). This existing framework contained an HV model developed based on the NGSIM oversaturated flow model (Yeo et al. 2008), an AV model developed based on field tests of adaptive cruise control (ACC) vehicles (Milanes and Shladover 2014), and a CAV model developed based on field tests of cooperative adaptive cruise control (CACC) vehicles (Milanes and Shladover 2014). The existing models were calibrated based on field data (Kan et al. 2019). The detailed algorithms of the HV, ACC, and CACC models are open source and documented in an existing report (Liu et al. 2018). One limitation of the existing framework is that it is incapable of capturing the behavior of CMDVs. Thus, the objective of this study was to accurately model CMDV behavior and to implement it in the existing modeling framework. Figure 1 shows the major components of the modeling framework.



Source: FHWA.

AV = autonomous vehicle.
CACC = cooperative adaptive cruise control.
CAV = connected automated vehicle.
CMDV = connected manually driven vehicle.
HV = human-driven vehicle.
I2V = infrastructure-to-vehicle.
NGSIM = Next Generation Simulation.

Figure 1. Diagram. Components of the modeling framework.

A CMDV is a manually driven vehicle. It differs from a conventional HV in two aspects. In this modeling framework, the CMDV model depicts the differences via two major functionalities:

- Serving as the leader of CACC strings while driving manually.
- Affecting human drivers' speed behavior via onboard message display of variable speed limit (VSL) or variable speed advisory (VSA) broadcast via infrastructure-to-vehicle (I2V) communication by traffic management centers (TMC).

The research team developed the first function in a previous effort; for details on the function algorithm, see Liu et al. (2018).

The remainder of this chapter gives a detailed description of the second function. The proposed CMDV model intended to depict the speed adaptation of connected drivers due to the I2V-based speed control and the quantification of the effects of such behavior changes on the mixed traffic. Studies of VSA/VSL compliance confirm that only a small portion of drivers choose to modulate their driving behavior requested by the system. This may be intentional (e.g., a driver can overlook the speed information from roadside signs) or unintentional (e.g., a driver fails to check the current speed against the posted speed limit). However, the research team hypothesizes that the CMDV system may help avoid these human errors by sending warning messages via onboard devices to better motivate drivers to follow the VSL or VSA. For example, in the study of Farah et al. (2012), drivers equipped with CMDV receive warning messages when they drove faster than the speed limit. Test results show that the average speed of the equipped drivers was significantly lower than the non-equipped drivers. The measured free-flow speed of the equipped drivers was almost equal to the speed limit. The same trend was observed in Spyropoulou et al. (2014), where the Intelligent Speed Adaptation system was adopted to promote a uniform speed among drivers in the concerned road segment.

Incorporating the CMDV model into the modeling framework enables users to assess a wide variety of traffic scenarios. Those scenarios reflect the effects of the latest technological

advances in vehicle automation and connectively, such as isolated ACC operation, CACC strings, and CV speed management. Simulating those scenarios in mixed traffic may generate useful insights into the transportation system, thus benefiting the development of future traffic management strategies and infrastructure upgrades. Since the components in the modeling framework have been calibrated and validated with the field data, the modeling outputs reasonably emulate the traffic flow dynamics expected in the real-world system. The research team also provided basic information on the model implementation. With this information, readers can easily adopt the developed models to their studies or transfer the models to different simulation environments.

DOCUMENT OVERVIEW

The next chapter describes the model development and logic in detail. The methodology for model calibration and validation and the corresponding results are presented in Chapter 3. In addition to CMDV modeling, CMDV implementation has been integrated into the existing modeling framework to simulate microscopic mixed traffic, which is described in Chapter 4. Chapter 5 presents the model sensitivity analysis, while Chapter 6 describes the model summary and implementation recommendations. Finally, Appendices A through D provide detailed information regarding the functions and pseudo codes of the modeling framework.

CHAPTER 2. MODEL DEVELOPMENT AND LOGIC

DESCRIPTIONS OF MODEL LOGIC

A CMDV is a manually driven vehicle; the infrastructure influences the human driver's behavior by display of an I2V advisory speed from a TMC. This research adopts the stimulus-response paradigm to model the behavior adaptation of the CMDV drivers in the traffic stream (Liu et al. 2017). This modeling approach quantifies the reaction of drivers (i.e., the response) due to the change of the surrounding traffic conditions (i.e., the stimuli). In traditional traffic flow models, the stimulus can be traffic stimuli such as the speed change of the preceding vehicles and the variation of the space gap between a subject vehicle and the leading vehicles. In the CMDV environment, additional stimuli can come from the onboard system that gives advisory speeds generated by the VSA/VSL control. This research incorporated the I2V-affected speed behavior parameter into a state-of-the-art microscopic car following (CF) model as factors impacting drivers' response sensitivity to traffic stimuli. Liu et al. (2018) details the models adopted to depict human drivers' CF and lane changing (LC) in various traffic conditions. The human driver model uses the desired speed parameter to describe drivers' speed choice when there is no constraint from other road users (Yeo et al. 2008). For a conventional human driver, the desired speed is a constant value drawn from a predetermined desired speed distribution. This study, on the other hand, developed a stochastic model that quantifies CMDV drivers' speed behavior under the influence of the advisory speed. This model was incorporated into the human driver model to replace the desired speed parameter originally used for conventional human drivers.

The CMDV systems can assist drivers such that the drivers' responses would become quicker and more consistent, possibly resulting in fewer traffic disturbances and a more stable traffic stream. The CMDV system effectiveness, however, relies on drivers' compliance with the CMDV information. The CMDV system cannot affect the traffic flow unless the driver follows its instructions. The ability and willingness to follow instructions varies from driver to driver. In this study, the driver speed compliance and its variation were determined based on empirical data sets reported by existing studies (Lu et al. 2019).

MODEL DEVELOPMENT

Speed Adaptation of Connected Drivers

In this study, the research team assumes that a TMC sends VSA instead of VSL to CMDVs. The VSA data samples were used because most U.S. transportation agencies adopt VSA instead of VSL for traffic management due to institutional issues. The research team also assumes that the communication between the TMC and CMDVs takes place in a perfect communication environment in this study. The modeling framework does not capture the communication delay and message packet loss. This assumption is not unreasonable since the VSA was updated every 30 seconds (s). One could send the VSA several times to guarantee it is received. After receiving the VSA via I2V communications from the TMC, the CMDV drivers will adjust their speed behaviors accordingly. Because the drivers' reactions to the VSA are naturally inconsistent, their speed adjustments can follow a random distribution instead of a constant level. For this reason, this study developed an empirical stochastic CMDV model that captured drivers' speed patterns under the VSA influence. The CMDV CF model was developed based on the NGSIM oversaturated flow human driver model calibrated in Kan et al. (2019). Within the model

framework, the same CF and LC logic for human drivers was used for the CMDV drivers because the CMDV drivers still operate their vehicles manually. This study revised the original human driver acceleration model to capture CMDV drivers' speeds under the influence of VSA. Particularly, the model uses the equation in figure 2 to represent a CMDV driver's acceleration:

$$a_{CV} = \min(a_F, a_N, a_G)$$

Figure 2. Equation. Acceleration of a connected manually driven vehicle.

where: a_F (feet per second squared [ft/s^2]) is the free acceleration, which describes the driver's acceleration when the speed choice is not constrained by the preceding vehicles; a_N (ft/s^2) is the Newell acceleration term (Newell 2002), which represents the driver's acceleration when following the preceding vehicle; and a_G (ft/s^2) is the Gipps' acceleration term (Ciuffo et al. 2012), which provides a safety constraint for crash avoidance. The equations for these terms are given in figure 3 through figure 8:

$$a_F = a_{Max}\left[1 - \left(\frac{v(t)}{V_0}\right)^{\gamma}\right]$$

Figure 3. Equation. Free acceleration term used for calculating the acceleration of a manually driven connected vehicle.

$$a_N(t) = \frac{\left(d(t) - d_{jam}\right)/\tau_h - v(t)}{\tau_h/2}$$

Figure 4. Equation. Car-following acceleration term used for calculating the acceleration of a connected manually driven vehicle.

$$a_G(t) = \frac{v_{safe}\,(t + \tau_r) - v(t)}{\tau_r}$$

Figure 5. Equation. Safety acceleration term used for calculating the acceleration of a connected manually driven vehicle.

$$v_{safe}\,(t + \tau_r) = A(t) + \sqrt{A(t)^2 - C(t)}$$

Figure 6. Equation. Safe speed for calculating the safety acceleration.

$$A(t) = -b_f \tau_r$$

Figure 7. Equation. Reaction related term in the safe speed calculation.

$$C(t) = b_f\left[2\left(d(t) - d_{jam}\right) - v(t)\tau_r - v_l(t)^2/(-\hat{b})\right]$$

Figure 8. Equation. Deceleration prediction term in the safe speed calculation.

where: a_{Max} (ft/s^2) is the driver's maximum acceptable acceleration, γ is the model coefficient, $v(t)$ (feet per second [ft/s]) is the current vehicle speed, V_0 (ft/s) is the driver's desired speed, τ_h (s) is the desired headway, d_{jam} (feet) is the jam gap, τ_r (s) is the reaction time, $v(t+\tau_r)$ (ft/s) is the

speed of the subject vehicle after reaction time, v_l (ft/s) is the speed of the preceding vehicle, b_f (ft/s^2) is the most severe braking that the subject driver wishes to undertake, $\hat{b}$ (ft/s^2) is the subject driver's estimate of preceding vehicle's most severe braking capabilities, and *d(t)* (feet [ft]) is clearance gap with regard to the leader at time *t*.

The three terms in figure 2 reflect different aspects of the driver behavior. The free acceleration term a_F is affected by a driver's maximum acceptable acceleration and desired speed; the CF term a_N by the desired headway and jam gap; and the safety term a_G by the maximum acceptable deceleration, reaction time, and jam gap (see figure 3). The VSA is expected to affect drivers' desired speed, while other behavior parameters, such as the reaction time and desired gap, receive little influence from the connected speed control. For this reason, the VSA algorithm can only change a CMDV driver's free acceleration term. The other two acceleration terms remain the same as the terms used in the human driver model. For this reason, V_0 for a connected driver becomes a variable, as shown in Figure 9.

For a normal (i.e., non-connected) human driver, the desired speed V_0 is a constant parameter that is determined based on the (fixed) speed limit of the road segment and the driver's compliance level to the speed limit. For a connected human driver (i.e., a CMDV), V_0 for time *t* is a variable calculated using Figure 9:

$$V_0(t) = V_{l,j}^*(t) + \sigma + \varepsilon$$

Figure 9. Equation. Desired speed of a connected manually driven vehicle.

Where: ε is a random parameter that represents the subject driver's response (compliance) to the advisory speed control, σ is a parameter that represents the random speed fluctuation around a target speed, V^* (ft/s) is the VSA/VSL of the current road section, and *j* is the link ID.

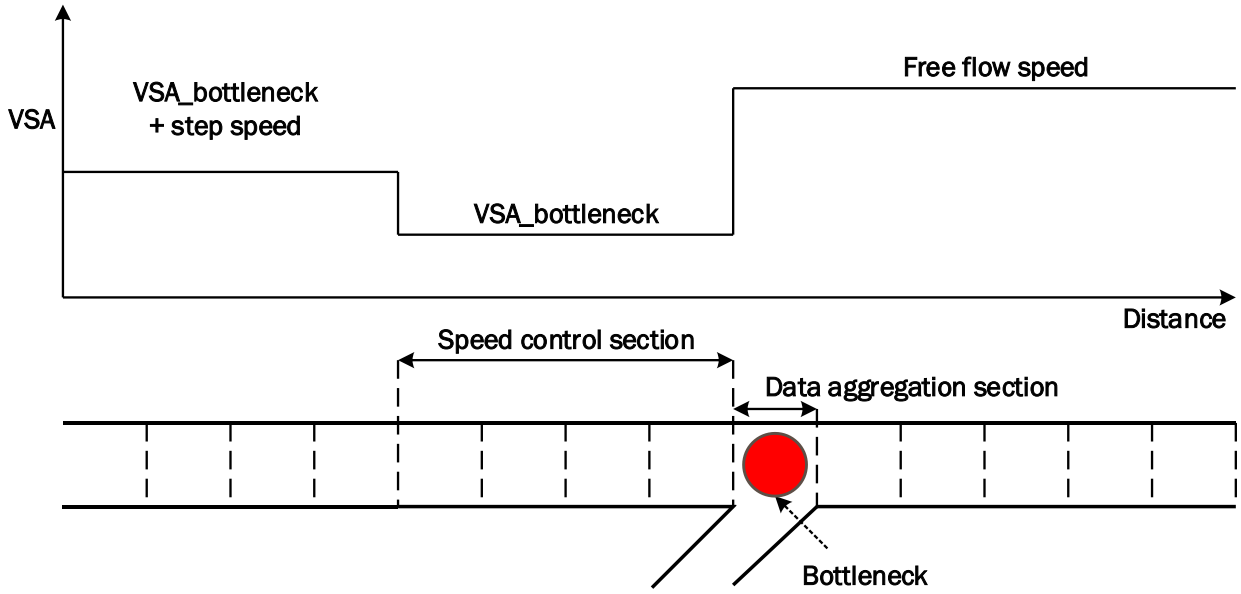
A human driver is not capable of tracking a target speed perfectly. Even if the driver completely complies with the VSL/VSA, the driver cannot always maintain the vehicle speed exactly at the VSL/VSA level. The vehicle speed is expected to fluctuate around the target speed and to be subject to the limit of the front vehicle speed and relative distance. To capture those factors, the equation adopts a stochastic parameter σ that follows a normal distribution with a zero mean and a standard deviation of 6.3 miles per hour (mph). This is identical to human drivers' speed fluctuation around the posted speed limit observed at the test cite (see Model Calibration and Validation). The parameter ε is the major parameter that affects the effectiveness of the CMDV. A value of zero indicates that the driver completely complies with the VSA/VSL from the onboard speed display. In this case, the traffic system functions most effectively because of the high CMDV compliance rate. As the parameter deviates from zero, the effectiveness of the CMDV at smoothing traffic flow will reduce. In this study, the distribution of ε was obtained from the field data. The calibration and validation of ε is discussed later in this chapter.

Traffic Data for Connected Speed Management

The CMDV affects drivers' speed behavior via VSA/VSL. This study uses a VSA algorithm derived based on the study in Lu et al. (2018). The algorithm can provide VSA to be displayed as an advisory speed on roadside variable message signs or directly fed to drivers in CMDVs. In the proposed model, the algorithm generates tailored speed advisories for individual CMDVs based on their distance from the bottlenecks and the traffic congestion conditions of the bottlenecks. To generate such VSAs, it continuously monitors the aggregated speed and traffic occupancy

patterns of the freeway corridor to identify if there are active bottlenecks. If there is no bottleneck, the algorithm intends to harmonize the traffic. If an active bottleneck is found, the algorithm adaptively determines the appropriate advisory speed for vehicles in the subjective section to alleviate the traffic congestion. When multiple bottlenecks are pinpointed, the VSA algorithm handles each bottleneck sequentially from the most upstream one to the most downstream one. It allows the advisory speed derived from a downstream bottleneck to override the advisory speed for the upstream bottlenecks if the downstream advisory speed is lower than the upstream one. This process will give priority to downstream bottlenecks when their congestion level is higher than the upstream counterparts, thus increasing the corridor throughput.

The VSA algorithm calls for aggregated freeway speed and occupancy data as the input for the variable advisory speed computation. Occupancy indicates the percentage of time a traffic detector is occupied by vehicles. To obtain the speed data set, it first segregates the concerned freeway corridor into consecutive data aggregation sections (as figure 10 shows). The speed data from individual CMDVs are averaged within a section over an update interval (e.g., 30 s in this study). The resulting mean speed is a temporal average of CMDVs' space mean speed, which represents the traffic condition for the section over a concerned time period. The computation of the average speed is shown in figure 11. In low CMDV market penetration cases, there might not be any CMDVs in a speed aggregation section during an update interval. In this case, the speed estimations for sections immediately upstream and downstream from this section are used to linearly interpolate the speed of this section. Fixed traffic monitoring stations, such as loop detector stations, provide the occupancy data sets along a concerned freeway corridor. Since the VSA algorithm adopts the critical occupancy as the control reference, the measured occupancy can be used to calculate the error between the control output and the reference. Since the measured occupancy is obtained via point loop sensors, the data collection device might not cover all data aggregation sections within the study freeway corridor. To address the problem, the linear interpolation approach is also adopted to calculate the occupancy of a section based on the occupancy measurements from the closest two loop detector stations.



Source: FHWA. VSA = variable speed advisory.

Figure 10. Plot. Segregation of a freeway corridor.

$$\bar{v}_i(k) = \frac{1}{\sum_{m=1}^{M} N_m} \sum_{m=1}^{M} \sum_{n=1}^{N_m} u_{mn}$$

Figure 11. Equation. Computation of the aggregated speed for a section.

where: $\bar{v}$(ft/s) is the speed of the traffic flow, *i* is the ID of the data aggregation section, *k* is the index of the speed limit update interval, *M* is the number of CMDVs that have traveled in section *i* during the update interval *k*, N_m is the number of speed data samples that vehicle *m* has sent to the speed controller while it was in section *i*, and *u* is the *n*th speed sample of vehicle *m*.

A data aggregation section can be short (e.g., less than 0.3 miles [mi]) compared with the length of the freeway corridor. This ensures that speed data from each section represent variations of the local congestion pattern. When implementing the advisory speed control, on the other hand, it is desirable to combine multiple data aggregation sections (e.g., 1.5 mi) such that they share the same advisory speed. Otherwise, drivers would need to change the speed frequently as they pass each section. This will reduce traffic stability and driving comfort. The combined section is called the speed control section (figure 10). Each speed control section contains multiple data aggregation sections. The VSA algorithm gradually decreases the advisory speed for vehicles upstream from a bottleneck (see the VSA_bottleneck + step speed in Figure 10). The advisory speed reduction takes place discretely at the end point of each speed control section.

Variable Advisory Speed Computation for Isolated Bottlenecks

The VSA algorithm decreases the input traffic flow to the bottleneck by recommending a reduced speed for upstream CMDVs to reduce or eliminate traffic congestion at an isolated bottleneck. In the mixed traffic stream, the scattered CMDVs can still act as actuators of the speed control, leading to the speed reduction of the entire traffic flow. Such a speed change lowers the traffic load to the bottleneck, thus helping the congested area recover from the

breakdown state, eventually bringing the bottleneck flow to the maximum capacity rate. In addition, the VSA algorithm reduces the advisory speed gradually for freeway segments upstream from the bottleneck. This can suppress the development of traffic disturbances that originated from the bottleneck.

If a data aggregation section *i* is an active bottleneck, the advisory speed for the bottleneck section and its downstream sections (i.e., $\forall\, j \in j \geq i$) is the original posted speed limit V_f. The advisory speed for the speed control sections immediately upstream from the bottleneck (i.e., $\forall j \in \lfloor (i-j)/N_c \rfloor = 1$) is V_j, as shown in Figure 12:

$$V_{l,j}(k) = \bar{v}_j(k-1) + \begin{cases} \varsigma_{o1} \cdot \left(O_c - O_j(k)\right) & O_j(k) < O_c \\ \varsigma_{o2} \cdot \left(O_c - O_j(k)\right) & O_j(k) \geq O_c \end{cases}$$

Figure 12. Equation. Variable advisory speed for sections upstream from the bottleneck.

where: N_c is the number of data aggregation sections in each speed control segment, *j* is the section ID, *k* is the timestep index, $\varsigma_{o1}, \varsigma_{o2} > 0$ are the unitless control gains ($\varsigma_{o1} = 0.6$ and $\varsigma_{o2} = 0.6$, respectively [Lu et al. 2018]), O_c is the critical occupancy ($O_c = 20$ percent in this study), and O_j is the measured bottleneck occupancy. The equation in Figure 12 also provides VSA for sections further upstream from the bottleneck (i.e., $\forall j \in \lfloor (i-j)/N_c \rfloor > 1$), with the exception that the measured bottleneck occupancy O_j is replaced by a weighted occupancy calculated by a semi-globally looking ahead algorithm, as shown in Figure 13:

$$O_j(k) = \rho_1 \cdot o_j(k) + \rho_2 \cdot o_{j+N_c}(k) + \rho_3 \cdot o_{j+2N_c}(k)$$

Figure 13. Equation. Weighted occupancy computation.

where: *o* is the measured occupancy and ρ_1, ρ_2, and ρ_3 are weights ($\rho_1 = 0.65$, $\rho_2 = 0.2$, and $\rho_3 = 0.15$, respectively [Lu et al. 2018]). The weight is so selected such that it has lower value when it is farther away in the downstream from the subject section.

The advisory speed of all sections is bounded by the spatial and temporal limits and the maximum and minimum limits, as shown in figure 14:

$$\left|V_{l,j}(k) - V_{l,j+1}(k)\right| \leq V_S$$

$$\left|V_{l,j}(k) - V_{l,j}(k-1)\right| \leq V_S$$

$$V_{l,min} \leq V_{l,j}(k) \leq V_{l,max}$$

Figure 14. Equation. Spatial, temporal, and maximum/minimum bounds for the advisory speed.

where: V_S (ft/s) is a step speed that limits the spatial and temporal change of the VSA. In the implementation, its value could be different from section to section depending on the length. The VSA algorithm has the following effects:

- Vehicles at the bottleneck section and its downstream sections (i.e., $j \geq i$) can travel at the original posted speed limit. This allows queued vehicles to leave the congestion area as fast as possible.

- For vehicles in the speed control section upstream from the bottleneck section, the advisory speed is reduced from the posted speed limit to regulate the traffic flow entering the bottleneck section. A feedback controller is adopted to achieve a stable change of the advisory speed.

- The sections further upstream from the bottleneck adopt a semi-globally looking ahead algorithm to compute the advisory speed. This algorithm allows the traffic stream in upstream sections to also respond to the bottleneck traffic condition, thus improving the overall freeway performance.

- The advisory speed change between two consecutive update intervals and two consecutive speed control segments is less than the step speed V_s. With this constraint, the advisory speed changes gradually over time and space. Especially when the VSA algorithm first becomes active, this prevents the controller from suddenly reducing the advisory speed to a level much lower than the posted speed limit.

- The advisory speed is bounded by the minimum speed limit V_{min} and maximum speed limit V_{max}. For traffic safety consideration, the maximum advisory speed should be no greater than the posted speed limit. The lower bound is associated with the smallest traffic flow the VSA algorithm can generate via the speed control.

Implementation of the Connected Speed Management in Freeway Corridors

For the corridor implementation, the VSA algorithm sequentially handles the bottlenecks from upstream to downstream using the method described in the previous section. The process starts with the identification of the bottleneck sections. Ideally, a bottleneck-searching algorithm considers all sections in a freeway corridor. Since the (recurrent) congestion usually takes place near the freeway merge, diverge, or weaving areas, considering sections outside those regions could increase the computation burden of the controllers. To this end, the algorithm may call for a list of data aggregation sections that contain the merge, diverge, and weaving links. In the list, all data aggregation sections that belong to the same merge, diverge, or weaving segment are mapped to a unique link ID. If a link contains N_l data aggregation sections, a data aggregation section p is identified as a bottleneck if the average speed meets the condition in figure 15:

$$\bar{v}_p(k) \leq V_T$$

Figure 15. Equation. Identification of bottleneck sections.

where: p is the section number ($p \in [1, N_l]$). Section p is upstream from section q, if $p < q$, and V_T (ft/s) is the threshold speed. V_T can be set as a function of the free-flow speed V_f, i.e., $V_T = \beta \cdot V_f$, where the coefficient $0 < \beta < 1$. The bottleneck searching algorithm moves from section 1 to section N_l (i.e., from the upstream section to the downstream section). Once a section in a link is identified as congested, the upstream front of a congested area is located. The remaining data aggregation sections are then disregarded for increasing the computation efficiency. Afterwards, the VSA algorithm uses equations in figure 12, figure 13, and figure 14 to determine the speed limits for all data aggregation sections of the corridor.

The VSA algorithm applies the above bottleneck searching and speed limit computation routine iteratively for each link in the merge, diverge, and weaving link list, from the most upstream link to the most downstream link. The final advisory speed of a data aggregation section *j* is determined as shown in figure 16:

$$V_{l,j}^{*} = \begin{cases} V_{l,j}^{u}, & V_{l,j}^{u} < V_{l,j}^{d} \\ V_{l,j}^{d}, & otherwise \end{cases}$$

Figure 16. Equation. Final advisory speed for a section.

where: V_j^u (ft/s) is the advisory speed computed for the upstream link and V_j^d (ft/s) is the advisory speed computed for the downstream link. With this update scheme, some upstream bottlenecks may no longer take the original posted speed limit as the advisory speed for output sections. Instead, the advisory speed is determined based on the traffic operation of the downstream bottleneck. If the downstream bottleneck is very congested, the algorithm can apply an advisory speed slower than the posted speed limit at the output section of the upstream bottleneck. As a result, the upstream bottleneck will generate a reduced input flow into the downstream bottleneck, helping the recovery of the traffic flow operation at the downstream bottleneck first. This method then coordinates the operation of the upstream and downstream bottlenecks for systematic performance improvement.

CHAPTER 3. MODEL CALIBRATION AND VALIATION

ASSUMPTIONS IN CALIBRATION AND VALIDATION

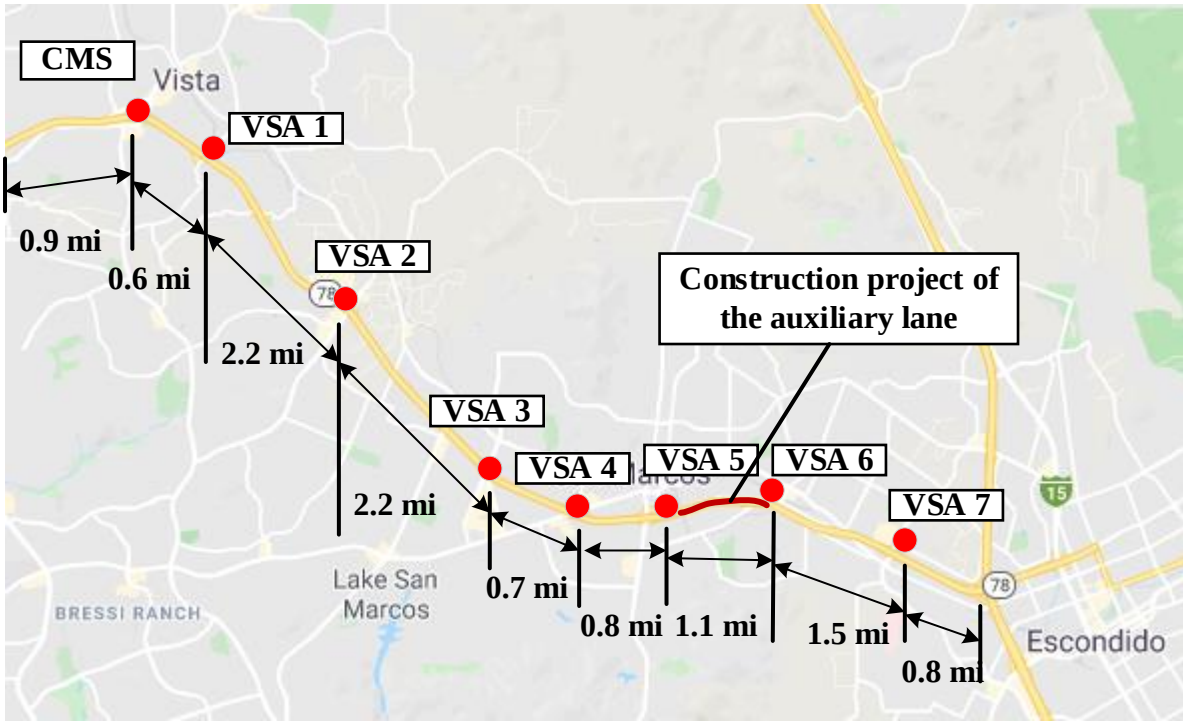
The parameter ε in figure 9 denotes the deviation of a driver's desired speed from a given advisory speed. It describes the driver's compliance level to the speed control. For this reason, the parameter could be calibrated and validated based on speed data sets that depict the behavior of real-world drivers under the influence of an active VSA/VSL control. To determine ε, the research adopts the following assumptions:

- The calibration and validation involve data sets that represent drivers' desired speed. While the desired speed represents the driver's personal preference, it is difficult to directly observe the speed choice with the field speed data. This study assumes that the desired speed pattern is reflected by drivers' speed choices when they have the freedom to select their speed levels. This usually happens in uncongested roads. In congested traffic, the vehicle movement is restricted by the preceding vehicles. The speed data in congested traffic thus cannot depict the desired speed distributions.

- There is no massively implemented VSA/VSL system that connects with drivers via I2V systems. Thus, it is impossible to obtain enough speed data for CMDVs for the model calibration and validation. To address the data limitation, the researchers assumed that the CMDV drivers' speed pattern is similar to normal human drivers' behaviors when those drivers are affected by the VSA/VSL displayed on the roadside changeable message signs. This assumption justifies the use of VSA field test data from San Diego, California, for model calibration and validation.

CALIBRATION AND VALIDATION DATASETS

Once the CMDV model captures the desired speed preference of individual CMDV drivers, the research team integrated the model into microscopic simulation of mixed traffic. Applying the CMDV model in a microsimulation simulation environment enables sensitivity analyses that reveal the impact of CMDV market penetration levels on traffic flow and vehicle energy efficiency. With the above assumptions and discussions, the research team calibrated and validated ε with the field data that describe normal human drivers' free speed choices under the influence of VSA. The calibration and validation data were collected under a California Department of Transportation (Caltrans) project, which included an extensive traffic data collection (Lu et al. 2019). This field test was conducted on State Route 78 eastbound (SR–78E) from Vista Village Drive in the City of Vista to the freeway interchange of SR–78E, and I–15 in the City of Escondido, California, as shown in figure 17. The roadway segment is a three-lane freeway with a posted speed limit of 65 mph, with 10 on-ramps and 10 off-ramps. This freeway corridor contained high-traffic volume in both morning (AM) and evening (PM) peak hours. The data were collected for 5 weeks, from March 30 to May 4, 2018, with VSA activation.



© Lu et al., 2019
CMS = changeable message sign.
mi = mile.
VSA = variable speed advisory.

Figure 17. Plot. Overview of system scope, variable speed advisory sign and changeable message sign locations, and construction area (Lu et al., 2019).

Recurrent bottlenecks were observed daily on weekdays in the study road segment. The morning bottleneck occurred between 6 and 9 a.m. at two locations: one near San Marcos Boulevard (milepost 12.27) and another near interchange of SR–78E to I–15 (milepost 16.6). The evening bottleneck occurred between 2 and 7 p.m. near the interchange of SR–78E to I–15 northbound (I–15 NB) and I–15 southbound (I–15 SB). Vehicle speeds dropped from 65 mph to as low as 15 mph after the onset of the congestion. There was a construction project on the new auxiliary lane in both directions of SR–78, between Twin Oaks Valley Road and Woodland Parkway. A speed limit of 65 mph was posted for that stretch of the roadway.

Seven VSA signs were located at the key locations determined from the analysis on the recurrent bottlenecks. Calculated VSA values were then rounded to multiples of 5 mph and displayed on the VSA signs. The VSA signs were located on the roadside to display the advisory speed, which was updated in real time every 30 s. A changeable message sign (CMS) displaying "FOLLOW ADVISORY SPEED" was placed at the starting point of the test site to instruct drivers to obey the speed posted by the downstream VSA. The posted speed on VSA during morning and evening peak hours was recommended to drivers, but not enforced.

Traffic data were collected from two sources. The traffic flow, speed, and occupancy were collected from loop detectors in the study corridor at 30-s intervals. The data were then fused with real-time speed and speed advisory data, captured every 30 s by radar sensors mounted on the VSA display equipment, at seven different sites along the 10.8-mi section of SR–78E. These two sources of data were then processed for the estimation of the overall traffic state along the

corridor, which was in-turn used to estimate the effectiveness of VSA on drivers' speed behaviors.

The entire data set, including traffic volume, speed, occupancy, and VSA level, provides 5 weeks' data samples. Data samples of 4 weeks were used to determine the distribution of ε. The remaining data samples were adopted for model validation.

CALIBRATION AND VALIDATION METHODOLOGY

The methodology adopted to calibrate and validate the random distribution of the drivers' response to the speed limit control follows a three-step procedure.

Step 1: Data Preparation

Given data on the speed limit and the observed speed, it is possible to calculate ε. In the observed data, VSA levels ranged from 5 to 65 mph in increments of 5 mph. The initial data analysis indicated that the speed deviation ε is a variable with respect to the VSA levels. The deviation tended to grow larger as VSA decreased. Ideally, the distribution fitting of ε should be conducted at individual VSA levels; however, the sample size in the data was small for low VSA levels, especially those lower than 25 mph. On the other hand, the field data showed a similar bimodal pattern for lower VSA levels from 5 to 35 mph with a peak around the VSA level and another peak at a value substantially larger than the VSA level. The second peak depicted driver groups with poor compliance with the VSA. The field observation also showed a similar pattern for higher VSA levels from 35 to 65 mph with a peak near 65 mph, indicating good compliance. For this reason, the data samples were grouped into two VSA levels—low VSA level from 0 to 35 mph and high VSA level from 35 to 65 mph. Note that the VSA feedback to the driver is a multiple of 5 mph. Therefore, such division does not lose the generality.

For each VSA level, the deviation data were divided randomly into a calibration set and a validation set. For each set of speed observation of low VSA level and high VSA level, 70 percent of the data were randomly selected for the calibration data set. The other 30 percent of the data were selected for the validation data set.

Step 2: Distribution Calibration

Since the deviation of the observed speed from the VSA does not follow an analytical distribution (e.g., normal distribution), an empirical distribution was used to describe the deviation distribution. An empirical cumulative distribution function (ECDF) is a cumulative form of the probability distribution function of a given data set. More formally, given the data points of sample size N, Y_1, Y_2, … Y_N are ordered with increasingly. The ECDF is defined as $ECDF(i) = n(i) / N$, where $n(i)$ is the number of data points less than Y_i. Using the calibration set Y, the ECDFs of ε were found for the low VSA level from 0 to 35 mph and the high VSA level from 35 to 65 mph. The difference between the observed speed and VSA for low and high VSA levels are noted as ε_{low} and ε_{high}, respectively. A Python™ package, statsmodels (Seabold and Perktold 2010), provides statistical model estimation functions. This package was used to determine the ECDFs of ε_{low} and ε_{high}, as $F_{c,n,low}(\varepsilon)$ and $F_{c,n,high}(\varepsilon)$, respectively, as shown in figure 18.

$$\varepsilon_{low} \sim F_{c,n,low}(\varepsilon)$$

$$\varepsilon_{high} \sim F_{c,n,high}(\varepsilon)$$

Figure 18. Equation. Empirical cumulative distribution functions for the low and high variable speed advisory levels.

where: ε_{low} and ε_{high} are the data samples in the low and high calibration data set, respectively, and $F_{c,n,low}(\varepsilon)$ and $F_{c,n,high}(\varepsilon)$ are the ECDF of the low and high calibration data set, respectively.

Step 3: Distribution Validation

The two-sample Kolmogorov-Smirnov (KS) test (Massey 1951) is a statistical test to validate if two samples come from an empirical distribution of a continuous random variable. In this work, a KS test is performed to validate the calibrated distribution $F_{c,n,low}(\varepsilon)$ and $F_{c,n,high}(\varepsilon)$. The two-sample KS test is a hypothesis test with the null hypothesis that two data samples for a random variable—in this case, the deviation ε—come from a common distribution. The test statistic $D_{n,m}$ is given as shown in figure 19:

$$D_{n,m} = \sup_{\varepsilon} \left| F_{c,n}(\varepsilon) - F_{v,m}(\varepsilon) \right|$$

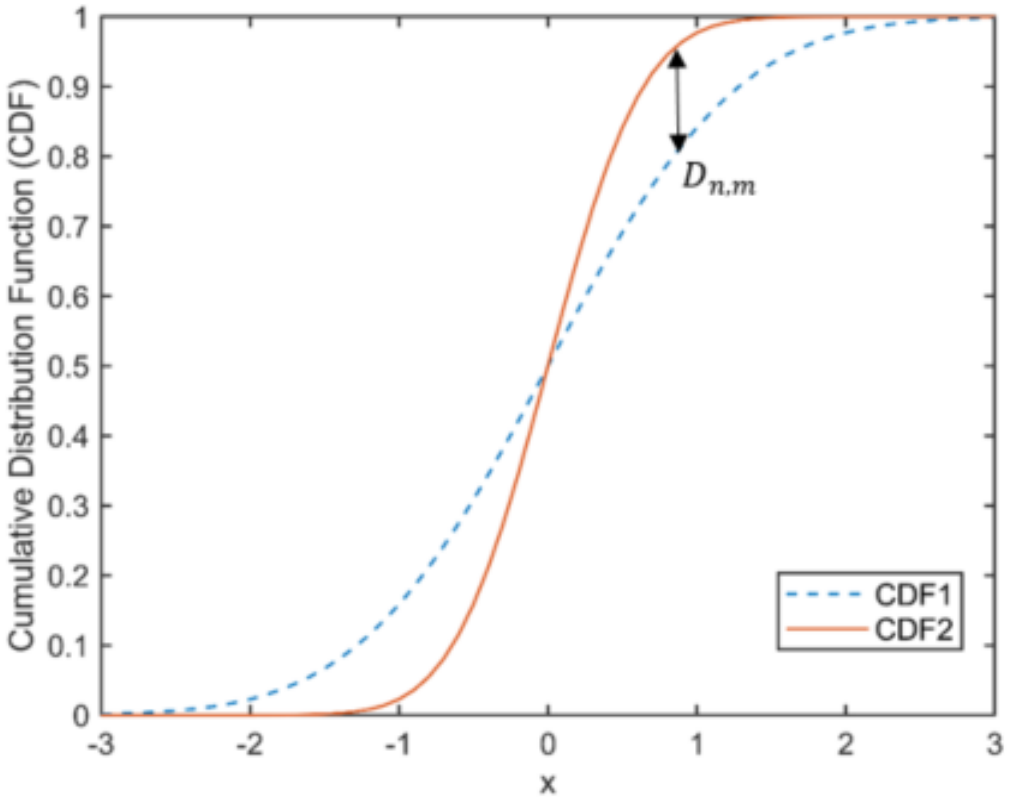
Figure 19. Equation. Kolmogorov-Smirnov test statistic.

where: $F_{c,n}(\varepsilon)$ is the ECDF of the calibration data set of size n and $F_{v,m}(\varepsilon)$ is the ECDF of the validation data set of size m. The test statistic $D_{n,m}$ is the largest difference between the two curves $F_{c,n,low}(\varepsilon)$ and $F_{v,n,high}(\varepsilon)$, as shown in figure 20 and figure 21. The null hypothesis is rejected at a significance level α for large samples, if:

$$D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{nm}}$$

Figure 20. Equation. Criterion for rejecting the null hypothesis.

where the critical value is given as $c(\alpha = 0.05) = 1.36$. The statistician fails to reject the null hypothesis, otherwise.



Source: FHWA.

Figure 21. Diagram. Visualization of the Kolmogorov-Smirnov test statistic.

CALIBRATION AND VALIDATION RESULTS

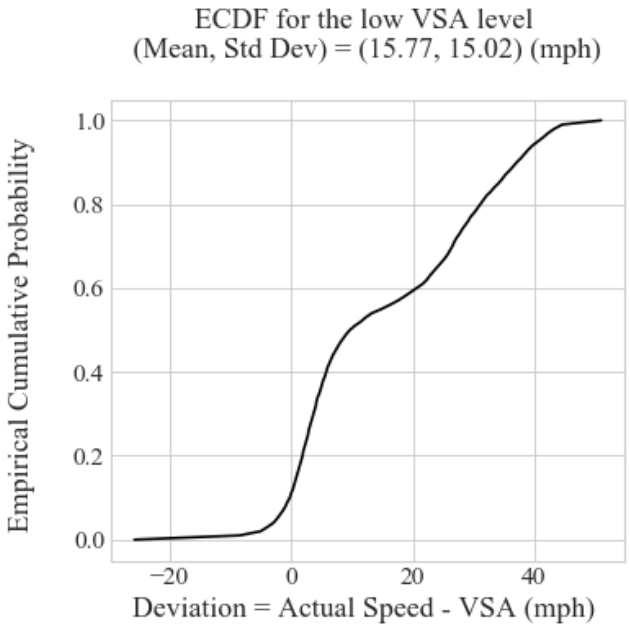
The sample sizes for the calibration and validation data sets for low and high VSA levels are given in table 1. Since the data were gathered from a field observation of drivers' speed on the public road, the sample collection was not performed in a controlled experiment. Though generally a sufficient sample size for a statistical analysis is 30 for well-designed random experiments, this rule may not be suitable for this uncontrolled observation. In addition, ε for each 5 mph VSA level did not seem to have a smooth distribution function even with a sample size in a scale of hundreds. After adding data samples from multiple VSA levels, the deviation for the low VSA and the high VSA seemed to have smooth distribution functions with sample sizes in a scale of thousands.

Table 1. Sample sizes for the low and high variable speed advisory cases.

Sample Sizes	Low Variable Speed Advisory (0–35 mph)	High Variable Speed Advisory (35–65 mph)
Sample size of calibration data set	5,380	73,706
Sample size of validation data set	2,306	31,589
Total sample size	7,686	105,295

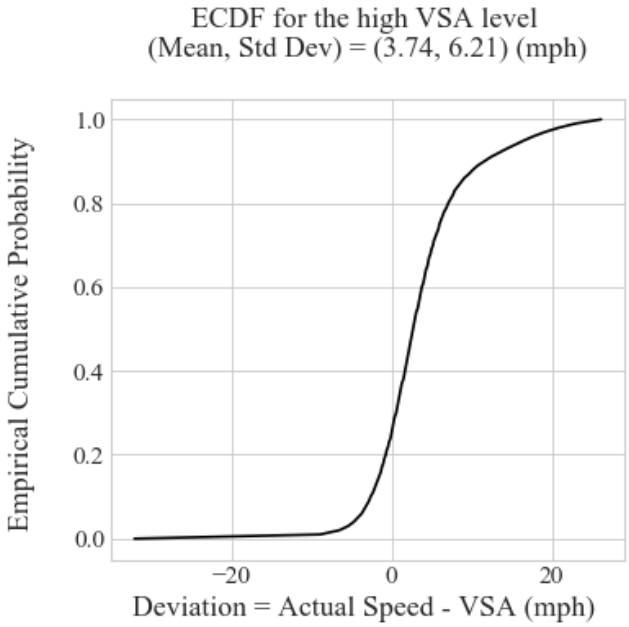
Source: FHWA. mph = miles per hour.

The calibration results of the empirical cumulative distribution functions for the low and high VSA levels are shown in figure 22. The *x*-axis is the deviation in mph. The *y*-axis is the cumulative probability. The mean and the standard deviation values are also given in the graphs. The calibrated distributions have been used in the sensitivity analyses to generate the random numbers that describe the baseline compliance level.



Source: FHWA.
ECDF = empirical cumulative distribution function. Std = standard deviation.
mph = miles per hour. VSA = variable speed advisory.

(a) Low variable speed advisory levels.



Source: FHWA.
ECDF = empirical cumulative distribution function. Std = standard deviation.
mph = miles per hour. VSA = variable speed advisory.

(a) High variable speed advisory levels.

Figure 22. Diagram. Calibrated empirical distributions of ε for the low and high variable speed advisory levels (mean and standard deviation in miles per hour).

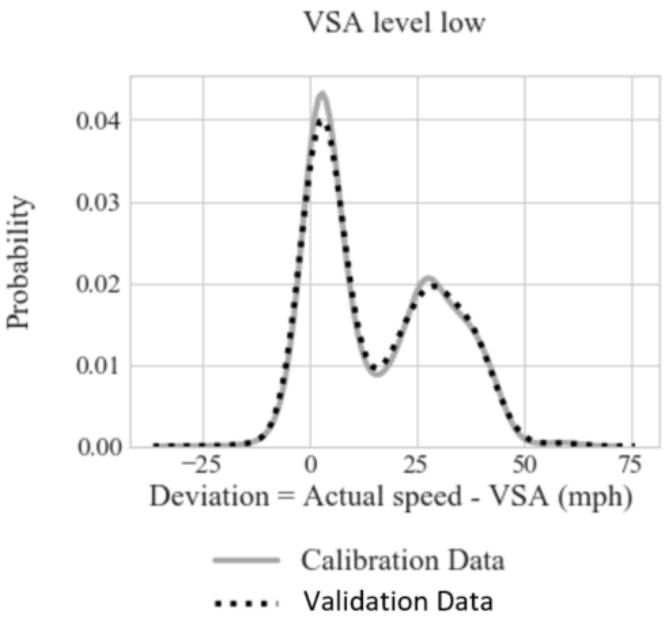
The results of the two-sample KS test for low and high VSA levels are given in table 2. In both cases, the researchers failed to reject the null hypotheses. In other words, the calibration and validation data sets follow a common distribution.

Table 2. Results of the two-sample Kolmogorov-Smirnov test.

Validation Results	Low Variable Speed Advisory (0–35 mph)	High Variable Speed Advisory (35–65 mph)
Null hypothesis	Data sample 1 and 2 follow a common distribution	Data sample 1 and 2 follow a common distribution
Data sample 1	Calibration data sets	Calibration data sets
Data sample 2	Validation data sets	Validation data sets
Level of significance	0.05	0.05
D statistic	0.01168	0.00564
Critical *D* value	0.03385	0.00915
Test result	Since the *D* statistic is smaller than the critical *D* value, fail to reject the null hypothesis	Since the *D* statistic is smaller than the critical *D* value, fail to reject the null hypothesis

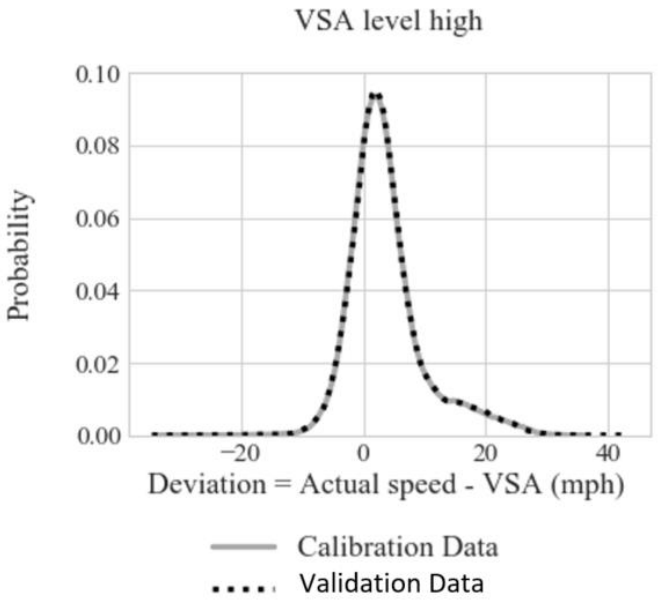
Source: FHWA.
mph = miles per hour.

In addition, the probability density functions for the low and high VSA levels are given to visually inspect the goodness of fit between the calibration and validation data sets in figure 23. The probability density function of the low VSA levels shows a bimodal distribution. The left peak around 0 mph represents driver populations that closely follow the VSA. The right peak is associated with drivers that do not comply with the VSA. This usually happened at freeway sections upstream from the bottleneck where the VSA was intentionally set to levels lower than the traffic speed. In this case, vehicles entering the bottleneck would reduce the speed, thus creating a reduced input traffic flow to the congested area. This could help lower the congestion intensity at the bottleneck. When the VSA was much lower than the traffic speed, some drivers may ignore the VSA because they might think the advisory speed was not credible. Such a behavior leads to the second peak in the probability density function.



Source: FHWA.
mph = miles per hour.
VSA = variable speed advisory.

(a) Low variable speed advisory levels.



Source: FHWA.
mph = miles per hour.
VSA = variable speed advisory.

(b) High variable speed advisory levels.

Figure 23. Diagram. The probability density functions for the calibration and validation datasets.

CHAPTER 4. BASIC INFORMATION ON MODEL IMPLEMENTATION

MODELING PROCESS

Understanding the update process of the simulation tool is important for implementing the proposed CMDV model in the simulation environment. A typical microscopic traffic simulation tool updates the traffic flow dynamics discretely—it computes the new location, speed, and acceleration of individual vehicles at the end of an update interval. While this is a simplification of the physical world, it gives computational advantage to the simulation tool, especially when the tool models thousands of vehicles in a network. In general, the computation process within an update interval can be divided into three states:

Stage 1: Information Gathering

This stage takes place at the very beginning of an update interval. It allows the simulation tool to obtain the latest status of the modeled vehicles, traffic control and monitoring devices, and traffic management strategies. Those pieces of information are the inputs for computing the new system status at the end of the update interval.

Stage 2: New System Status Computation

The simulation tool computes the new system status based on the inputs from the previous stage. In this stage, the tool calculates the new position, speed, and acceleration for individual modeled vehicles, the new state for the traffic signals, and the updated implementation scheme of traffic management strategies (e.g., activate or deactivate a managed lane, calculate new advisory speeds, and turn ramp meters on or off). This stage only computes the new system status, without implementing the updates. All computation processes in this stage are performed based on a common deterministic baseline (i.e., the system status of the previous update interval). In this case, the update of the modeled entities does not rely on the new status of other entities. Such an update method allows the simulation tool to execute the update process without following a specific order (from the most downstream section to the upstream section, for example). The new status computation for individual modeled entities can start simultaneously. This update method is ideal for applying parallel computing to increase the simulation speed.

Stage 3: System Status Update

Once the simulation tool has determined the new status of each modeled entity in the previous stage, it assigns the new status to each entity. The modeled vehicles move to a new position with an updated speed and acceleration. The signal lights change color. The new scheme of the traffic management strategies is executed. This concludes the update interval.

The proposed CMDV algorithm can be easily implemented in the three-stage update framework:

- The algorithm uses CMDV speed and loop detector occupancy as the inputs. Those inputs are obtained in the first stage when the simulation tool gathers the current status of all simulated entities. Usually, a simulation tool with an application programming interface (API) would provide functions to access attributes of simulated entities. Those functions may be called to collect the inputs for the CMDV algorithm.

- The CMDV algorithm computes the advisory speed based on the CMDV speed and loop occupancy inputs. The computation process is implemented in the second stage. If the simulation tool allows the user to deploy the customized system update functions, the computation of the advisory speed can be integrated into the customized update functions.

- The computed advisory speed is sent to individual CMDVs in the traffic stream. This is achieved in the last stage. In this case, the user would develop a function to update the desired speed for only the CMDV type, while keeping the original desired speed for other vehicle types.

CHAPTER 5. USE CASE AND SENSITIVITY STUDY

IMPLEMENTATION OF THE DEVELOPED MODEL INTO A TRAFFIC SIMULATION TOOL

The CMDV model was implemented in the modeling framework by creating a new CMDV vehicle class, in addition to the existing HV, AV, and CAV classes. The logic of the CMDV model is not proprietary and can be implemented into the analyst's choice of microsimulation software. For the purpose of testing the model, the researchers implemented the CF and LC mechanisms for the four vehicle classes in a microscopic simulation tool via its Microscopic Simulator Software Development Kit (microSDK) and API programming tools (Aimsun 2020). The microSDK contains two key classes—a behavior model (BM) class and a simulated vehicle (SV) class. The BM class configures the global parameters (e.g., simulation time, timestep, and origin-demand matrix) of the simulation environment. It controls the process of each simulation run (e.g., start/end of the simulation, creation of new vehicles, and update of vehicle movement). The SV class offers functions to access real-time vehicle data and compute the acceleration, speed, and location of individual vehicles at each update interval. By incorporating customized methods and functions in the two classes, the model framework reproduces the patterns of a traffic flow mixed with regular HVs, CMDVs, ACC vehicles, and CACC vehicles.

The detailed information of the model implementation in Aimsun is given in Appendices A through D. With the function descriptions, users with other simulation tools (e.g., Vissim or SUMO) may find similar functions in their tools. Appendix A, in particular, provides a detailed description of the functions for simulation process control. In each simulation interval, those functions sequentially evaluate algorithms that generate new vehicles and depict vehicle interactions, including CF and LC, the control mode transitions of ACC and CACC vehicles, and the speed behavior adaptation of CVs. The outputs of those functions would serve as the updated traffic state in the next simulation interval. Appendix B describes the vehicle generation functions, while Appendix C presents the CF and LC algorithms for HVs. Appendix D depicts the functions for modeling ACC and CACC control mode transitions, as well as CMDV speed behaviors.

DESIGN OF SIMULATION EXPERIMENTS

The developed CMDV model was tested under various simulation scenarios. Those scenarios tested the model sensitivity to assumptions of the CMDV market penetration rate (MPR) and the human drivers' compliance levels to the VSA from the I2V communication. The MPR increased from 0 to 100 percent, with a 10-percent step. The compliance level increased from the baseline compliance level to the full compliance level. The baseline compliance level was obtained based on the empirical data described in figure 22. A modeled driver's compliance (i.e., ε in figure 9) was determined following the next two steps:

- Step 1: generate a random real number, μ, that follows a uniform distribution between zero and one.
- Step 2: take μ as a cumulative probability, find its corresponding ε in figure 22.

Thus, the driver's speed deviation from the VSA follows the empirical distribution identified in the model calibration. In scenarios where drivers' compliance levels are increased, the random

compliance ε changes based on the increased rate of compliance. If the compliance level increases by θ ($0 < \theta <= 1$), the new compliance is given as shown in figure 24:

$$\varepsilon' = \varepsilon \cdot (1 - \theta)$$

Figure 24. Equation. Determination of the increased compliance level.

When θ is equal to 1, it reaches the full compliance level (i.e., $\varepsilon' = 0$). The simulation scenarios are listed in table 3.

Table 3. Simulation scenarios for connected manually driven vehicles.

ID	Connected Manually Driven Vehicle Market Penetration Rate (Percent)	Compliance Increase (θ)
1	0	N/A
2	10	0 (baseline compliance)
3	20	0
…	…	0
11	100	0
12	10	0.05
13	10	0.10
14	10	0.15
…	…	…
17	10	0.30
18	10	1.00
19	20	0.05
20	20	0.10
21	20	0.15
…	…	…
24	20	0.30
25	20	1.00
26	100	0.05
27	100	0.10
28	100	0.15
…	…	…
31	100	0.30
32	100	1.00

Source: FHWA.
N/A = not applicable.

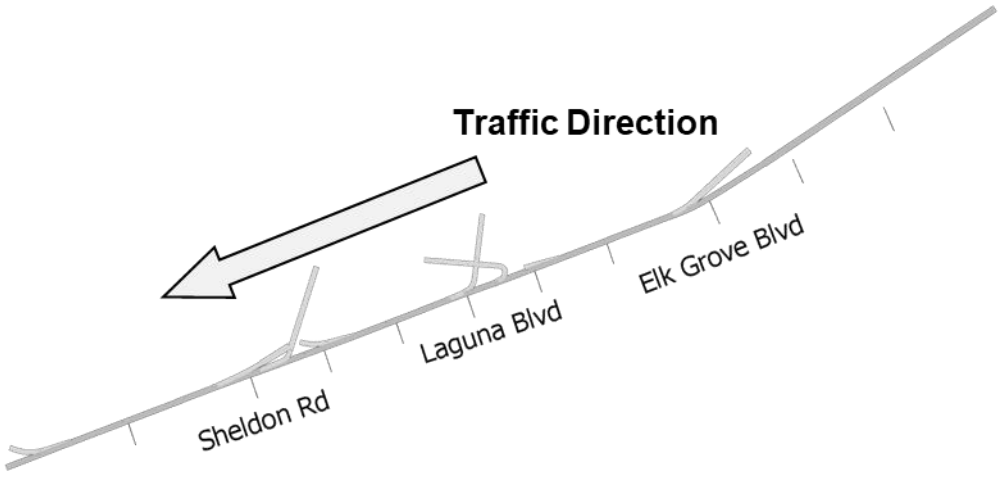
In addition to the CMDVs, the proposed VSA algorithm was applied with the CACC vehicles such that the advisory speed was used as the reference speed of the CACC controller. When an advisory speed was sent to CACC vehicles, it was assumed that the CACC controller would perfectly adopt the advisory speed as the reference speed. There was no random variation in the reference speed. The resulting scenarios can depict the effects of automated controllers on the VSA performance. The CACC scenarios are defined in table 4 as the following:

Table 4. Simulation scenarios for cooperative adaptive cruise control vehicles.

ID	Cooperative Adaptive Cruise Control Market Penetration Rate (Percent)	Variable Speed Advisory On or Off
33	10	Off
34	20	Off
35	30	Off
36	10	On
37	20	On
38	30	On

Source: FHWA.

The simulation scenarios were evaluated on a 4.3 miles freeway corridor (see figure 25). The freeway corridor was coded into the adopted simulation tool based on the real-world SR–99 corridor south from downtown Sacramento, California. The corridor contained three on-ramp bottlenecks at Elk Grove Boulevard interchange, Laguna Boulevard interchange, and Sheldon Road interchange, respectively. The Sheldon Road interchange is the most downstream bottleneck that causes the most severe congestion during the AM peak hours. The simulation runs cover an 8-hour period from 4 a.m. to 12 p.m. The formation and recovery of the bottlenecks can be captured entirely during the simulation period. The simulation inputs were real-world traffic counts obtained from the Performance Measurement System (PeMS) database (Caltrans 2020).



Source: FHWA.

Figure 25. Diagram. Simulated freeway corridor.

SIMULATION RESULTS FOR THE DIFFERENT SCENARIOS

The impacts of CMDV market penetration are depicted by the average traffic speed and vehicle fuel economy. The fuel results were computed using two models: the Virginia Tech (VT) comprehensive power-based fuel consumption model (Rakha et al. 2011) and the MOtor Vehicle Emission Simulator (MOVES) model (Ramezani et al. 2019). Results from the two models

provide a comprehensive picture of the effectiveness of the CMDV speed adaptation. The parameters used in the two fuel models are listed in table 5 and table 6.

Table 5. Fuel model parameters of the MOtor Vehicle Emission Simulator model.

Parameter Name	Parameter Value
Rolling resistance (ft-lb/ft)	35.1
Rotational resistance (FPS/ft^2)	0.137
Aerodynamic drag coefficient (FPS^2/ft^3)	1.03E-2
Vehicle age distribution	California average vehicle age distribution in 2019
Fuel supply	MOVES default
Fuel formulation	MOVES default
Alternative vehicle and fuels technology	MOVES default
Temperature	Average July temperature measured in Sacramento, California
Humidity	Average July humidity measured in Sacramento, California

Source: FHWA.
ft•lb/ft = foot-pound per foot. FPS/ft^2 = foot-pound-second per square foot. FPS^2/ft^3 foot-pound-second squared per cubic foot.

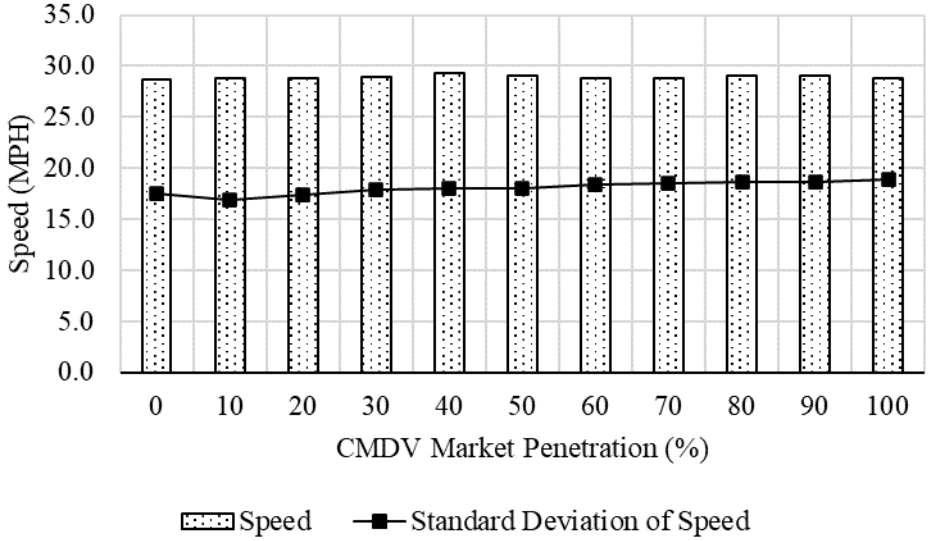
Table 6. Fuel model parameters of the Virginia Tech comprehensive power-based fuel consumption model.

Parameter Name	Parameter Value
Alpha 0	5.92E-4
Alpha 1	4.24E-5
Alpha 2	1.00E-6
Vehicle mass (lb)	3203
Driveline efficiency	0.92
Density of air at sea level at a temperature of 15 °C (lb/ft^3)	7.68E-2
Vehicle drag coefficient	0.30
Correction factor for altitude	1.00
Vehicle frontal area (ft^2)	25.0
Rolling resistance parameters Cr, C1, and C2	1.75, 3.28E-1, 4.58

Source: FHWA.
°C = degree Celsius. ft^2 = square foot. lb = pound. lb/ft^3 = pound per cubic foot.

As figure 26 shows, the average speed of the traffic flow has little change with the CMDV market penetration. However, this does not mean that the CMDV did not change the traffic flow pattern. Essentially, the CMDV had two major effects to the traffic flow. First, the CMDVs upstream from the bottleneck adopted reduced speeds recommended by the VSA algorithm. As CMDVs started to slow down, they made the following vehicles reduce the speed as well. This eventually led to the slowdown of the traffic flow for freeway sections upstream from the bottleneck. On the other hand, the speed reduction helped the bottleneck recover from the congested status, which increased the queue discharging rate. As the queued vehicles could leave

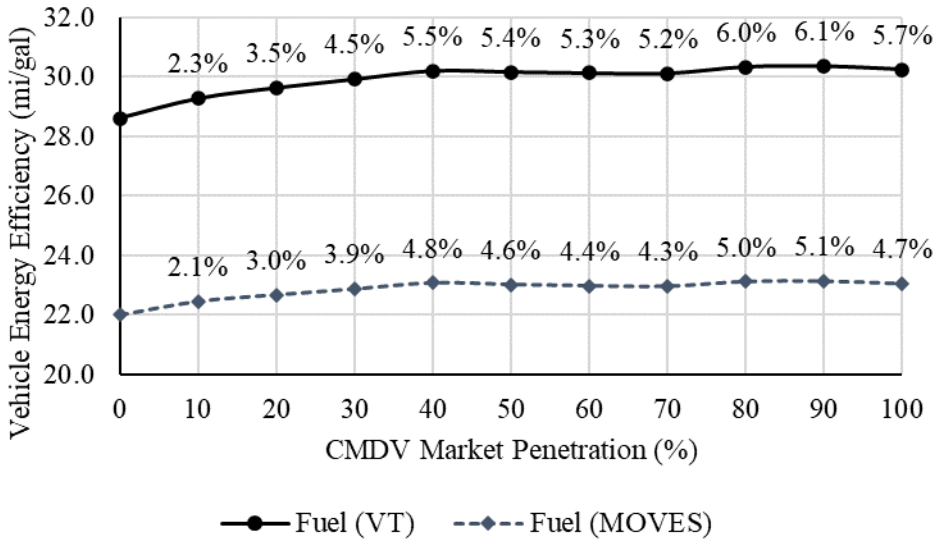
the congested region faster, vehicles spent less time in the low speed status. For this reason, the average speed of the modeled vehicles in the congested bottleneck got improvement. As the two effects worked simultaneously, the average speed of the studied freeway corridor remained identical to the non-CMDV case.



Source: FHWA.
% = percent.
CMDV = connected manually driven vehicle.
mph = miles per hour.

Figure 26. Diagram. Average vehicle speed and the standard deviation of the speed under various connected manually driven vehicle market penetrations.

The two effects of CMDV led to the harmonization of vehicle speed and improvement of the bottleneck traffic flow, which resulted in improved vehicle fuel efficiency. As figure 27 shows, vehicle fuel efficiency increased with the CMDV market penetration. The largest increase was observed when the market penetration changed from 0 to 40 percent. When the CMDV market penetration was 40 percent or higher, the fuel efficiency increase became less rapid. This indicates that 40 percent is a turning point for the CMDV performance when the driver compliance level follows the observed distribution. The speed adaptation of less than half of vehicles could already alter the speed of the traffic flow. Further increasing the CMDV market penetration could not generate additional vehicle energy benefits. The VT model gave a higher estimation than the MOVES model because the VT model used parameters of more recent vehicles than those in the MOVES model. The newer vehicles generally have higher energy performance than the older vehicles. Despite the difference in the magnitude of the results, the two models provide curves with a similar trend.



Source: FHWA.
% = percent.
CMDV = connected manually driven vehicle.
mi/gal = miles per gallon.
MOVES = MOtor Vehicle Emission Simulator.
VT = Virginia Tech model.

Figure 27. Diagram. Average vehicle fuel efficiency under various connected manually driven vehicle market penetrations.

The effects of drivers' compliance are depicted in table 7, table 8, and table 9. The results indicate that a small change in the driver compliance level had little influence on both the average speed and fuel efficiency. The speed and fuel performance of the studied freeway corridor changed less than 2 percent as drivers' compliance level increased from 5 to 30 percent. This trend is consistent when the CMDV market penetration was 10 percent, 20 percent, or 100 percent. It implies that the performance of the VSA is not sensitive to the small increase of drivers' compliance, regardless of the CMDV market penetration levels. On the other hand, when all drivers fully complied with the CMDV, the vehicle energy efficiency became substantially better than the baseline compliance (see the final rows in Tables 9–11). The standard deviation of vehicle speed significantly decreased as well. The results suggest that the effectiveness of CMDV also relies on how drivers react to the CMDV information. When drivers are willing to closely follow the CMDV instruction, the effects of CMDV can obtain an extra 2–3 percent improvement. Given that the energy efficiency improvement ranges from 2 to 6 percent under the baseline compliance (see figure 27), the extra increase of the energy performance is significant.

Table 7. Effects of driver compliance level on the speed and vehicle fuel efficiency at 10 percent connected manually driven vehicle market penetration case.

Compliance Increase (%)	Fuel Efficiency from MOVES (mi/gal)	Fuel Efficiency from VT (mi/gal)	Vehicle Speed (mph)	Std of Vehicle Speed (mph)
0	22.5	29.3	28.9	16.9
5	22.6	29.5	29.3	17.1
10	22.6	29.4	29.1	17.0
15	22.6	29.5	29.1	17.2
20	22.7	29.6	29.2	16.9
25	22.7	29.6	29.2	16.9
30	22.5	29.4	28.8	16.8
100	22.8	29.7	28.9	16.0

Source: FHWA.
% = percent. mi/gal = miles per gallon. MOVES = MOtor Vehicle Emission Simulator. mph = miles per hour. Std = standard deviation. VT = Virginia Tech model.

Table 8. Effects of driver compliance level on the speed and vehicle fuel efficiency at 20 percent connected manually driven vehicle market penetration case.

Compliance Increase (%)	Fuel Efficiency from MOVES (mi/gal)	Fuel Efficiency from VT (mi/gal)	Vehicle Speed (mph)	Std of Vehicle Speed (mph)
0	22.7	29.6	28.8	17.4
5	22.9	29.9	29.1	17.4
10	22.9	29.9	29.2	17.5
15	22.8	29.8	29.0	17.3
20	22.7	29.7	28.6	17.4
25	22.7	29.7	28.6	17.5
30	22.8	29.8	28.8	17.3
100	23.2	30.3	28.8	16.2

Source: FHWA.
% = percent. mi/gal = miles per gallon. MOVES = MOtor Vehicle Emission Simulator. mph = miles per hour. Std = standard deviation. VT = Virginia Tech model.

Table 9. Effects of driver compliance level on the speed and vehicle fuel efficiency at 100 percent connected manually driven vehicle market penetration case.

Compliance Increase (%)	Fuel Efficiency from MOVES (mi/gal)	Fuel Efficiency from VT (mi/gal)	Vehicle Speed (mph)	Std of Vehicle Speed (mph)
0	23.0	30.2	28.8	18.9
5	23.2	30.4	29.0	18.8
10	23.2	30.5	29.0	19.0
15	23.3	30.6	29.0	19.0
20	23.4	30.7	29.3	18.9
25	23.2	30.4	28.6	19.0
30	23.3	30.6	28.9	19.0
100	23.7	31.0	28.6	18.2

Source: FHWA.
% = percent. mi/gal = miles per gallon. MOVES = MOtor Vehicle Emission Simulator. mph = miles per hour. Std = standard deviation. VT = Virginia Tech model.

As the compliance level was further increased to 100 percent, it substantially changed the traffic flow patterns. Figure 28 and figure 29 show the fundamental diagrams of the bottleneck link when the CMDV market penetration was 100 percent. Table 10 shows a comparison of the averaged speed and vehicle fuel efficiency. Figure 28 depicts the results when the baseline compliance level was applied, whereas Figure 29 illustrates the results with 100 percent improvement of the compliance level. Comparing to Figure 28, points in Figure 29 shift left and up toward higher flow regions. This indicates that the CMDV speed compliance improved the traffic flow of the bottleneck.

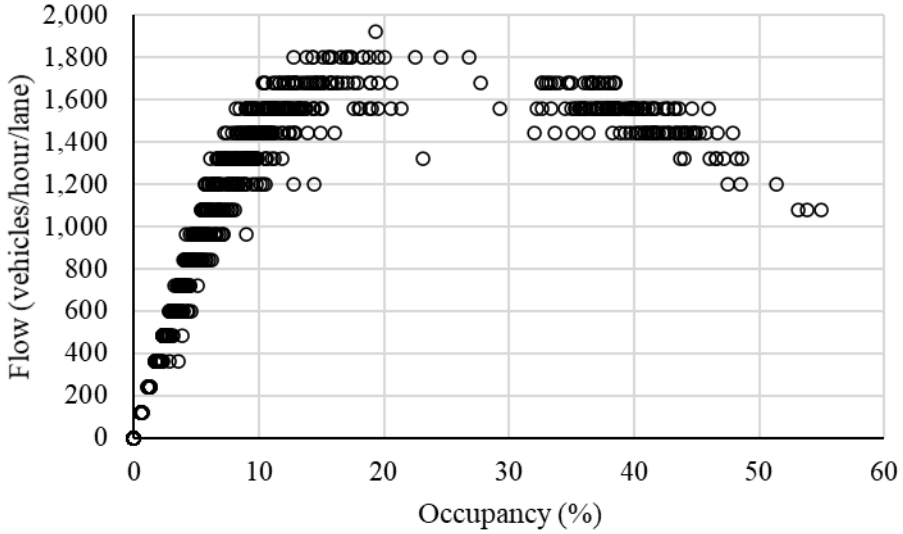
The increase of the compliance level resulted in the increase of the capability for the CMDV to regulate the traffic flow. Consequently, the CMDV speed adaptation could more effectively reduce the input flow to the bottleneck, leading to a decrease in the congestion level and the increase of the bottleneck traffic flow rate. That explained the left and upward shift of the points in the fundamental diagram in figure 29. However, the VSA algorithm implemented in the analysis was a reactive algorithm. It reduced the input traffic flow to the bottleneck when the bottleneck traffic became congested, and it increased the input flow after the congestion disappeared. The delayed response of the VSA algorithm did not always benefit traffic flow. In some cases, the bottleneck was congested, while the on-ramp traffic would be light in the next few minutes. The congestion level should reduce due to the decrease of the on-ramp traffic disturbance. This naturally led to an increase in the bottleneck flow rate. However, the VSA algorithm would continue reducing the input flow to the bottleneck in this case because it still detected the congestion status of the bottleneck. This could overly reduce the input flow, making some of the bottleneck capacity underused (see points in the dashed box in figure 29, where the occupancy is in the middle range, but the flow rate is overly reduced by the VSA). In other cases, the bottleneck congestion had been relieved, but the heavy on-ramp traffic would break down the bottleneck traffic again in the next few minutes. The VSA algorithm would increase the input flow because it only detected the light traffic condition of the bottleneck. This could create much heavier congestion than in the no-control case (see points in the dotted box in figure 29, where the capacity reduction caused by the excessive traffic input is much higher than the rest of the

points in the same occupancy range). As the increase of drivers' compliance levels led to the enhanced VSA control capability, these undesirable control behaviors also happened more frequently. The negative effect of these behaviors could offset the benefit due to the bottleneck flow increase. The above discussion indicates that a predicative algorithm is desired to generate VSAs that match the future traffic conditions. Such an algorithm is expected to enhance the performance of CMDV.

Table 10. Comparison of the baseline compliance and the full compliance level.

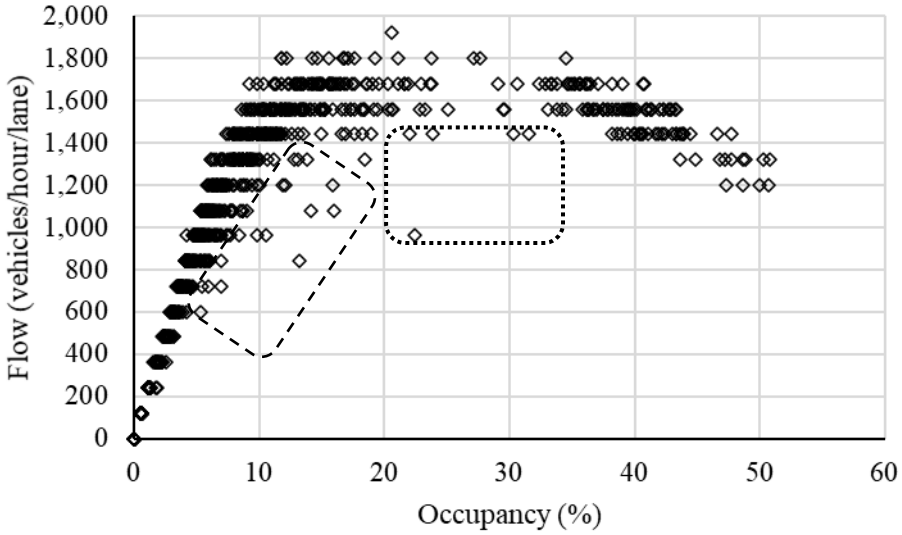
CMDV MPR (%)	Compliance Increase (%)	Fuel Efficiency from MOVES (mi/gal)	Fuel Efficiency from VT (mi/gal)	Vehicle Speed (mph)	Std of Vehicle Speed (mph)
10	0	22.5	29.3	28.9	16.9
10	100	22.8	29.7	28.9	16.0
20	0	22.7	29.6	28.8	17.4
20	100	23.2	30.3	28.8	16.2
100	0	23.0	30.2	28.8	18.9
100	100	23.7	31.0	28.6	18.2

Source: FHWA.
% = percent. CMDV = connected manually driven vehicle. mi/gal = miles per gallon. MOVES = MOtor Vehicle Emission Simulator. mph = miles per hour. MPR = market penetration rate. Std = standard deviation. VT = Virginia Tech model.



Source: FHWA.
% = percent.

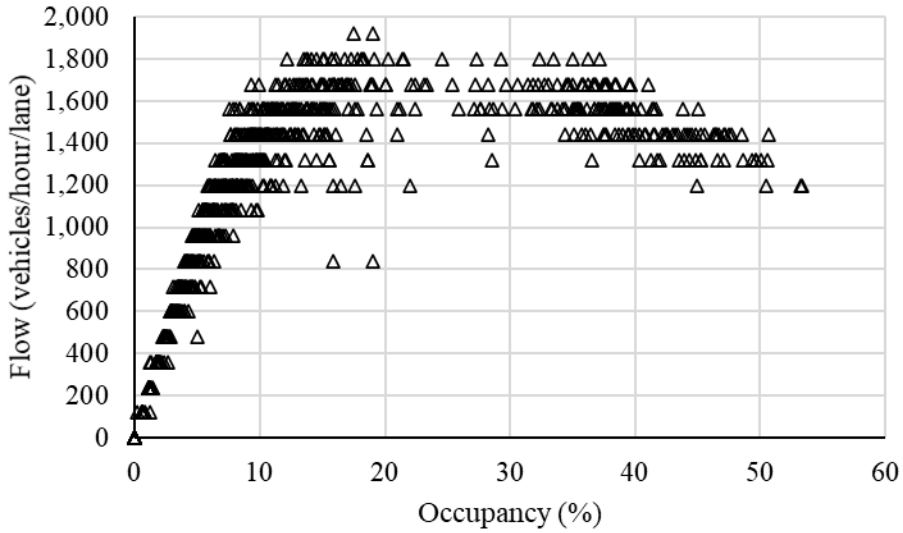
Figure 28. Diagram. Fundamental diagram of 20 percent connected manually driven vehicle market penetration with the baseline compliance level.



Source: FHWA.
% = percent.

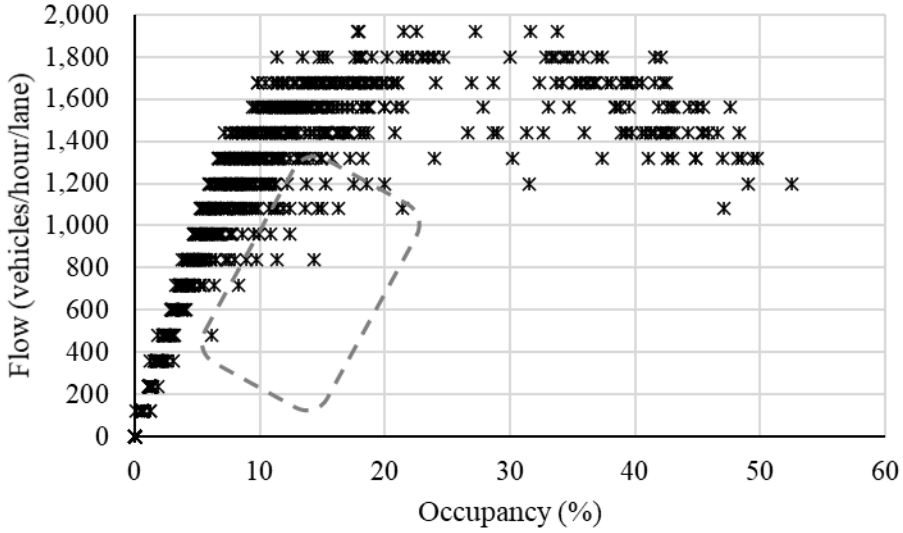
Figure 29. Diagram. Fundamental diagram of 20 percent connected manually driven vehicle market penetration with the full compliance level.

When the VSA algorithm was applied with CACC vehicles, the vehicles perfectly adopted the advisory speed. As a result, the above undesired control behavior became more prominent than in the CMDV case. For the 30 percent CACC market penetration case, the fundamental diagrams with and without the I2V advisory speed are displayed in figure 30 and in figure 31. The comparison of the two plots shows that the congestion at the bottleneck was greatly improved by the VSA control. Nonetheless, there was also a great increase in the cases where the bottleneck capacity was underused (e.g., points in the dashed box in in figure 31). In this case, the queued vehicles upstream from the bottleneck spent more time in the congested condition, reducing the speed and vehicle energy performance of the corridor. The comparison between the traffic speed and vehicle fuel economy with and without VSA is shown in table 11. The vehicle fuel efficiency was improved by 5–6 percent, while the speed remained the same with and without the VSA. The vehicle energy efficiency improvement was in the same range as the CMDV cases with 40 percent or higher market penetration. If the VSA algorithm can eliminate the capacity underuse condition, the performance of the VSA under the CAV condition may be much better.



Source: FHWA.
% = percent.

Figure 30. Diagram. Fundamental diagram of 30 percent cooperative adaptive cruise control vehicle market penetration without the variable speed advisory.



Source: FHWA.
% = percent.

Figure 31. Diagram. Fundamental diagram of 30 percent cooperative adaptive cruise control vehicle market penetration with the variable speed advisory.

Table 11. Comparison of the baseline compliance and the full compliance level.

Comparison	Fuel Efficiency from MOVES (mi/gal)	Fuel Efficiency from VT (mi/gal)	Vehicle Speed (mph)	Std of Vehicle Speed (mph)
No VSA	22.8	29.3	30.8	18.7
VSA	23.9	31.1	30.6	15.2
Percent change	5.1 percent	6.0 percent	-0.9 percent	-18.9 percent

Source: FHWA.
mi/gal = miles per gallon. MOVES = MOtor Vehicle Emission Simulator. mph = miles per hour. Std = standard deviation. VSA = variable speed advisory. VT = Virginia Tech model.

The above results indicate that the reactive nature of the VSA controller impedes the improvement of the SPDHRM and vehicle fuel efficiency, even when the CMDV drivers are willing to fully comply with the advisory speed or the CACC vehicles perfectly adopt the advisory speed. A possible direction for further improvement is to use a predictive VSA algorithm to generate advisory speeds. The predictive algorithm can estimate the short-term disturbances from the on-ramp traffic. It then computes the input flow to the bottleneck that matches the bottleneck traffic condition, leading to a more efficient bottleneck operation. However, the traffic flow changes rapidly within the bottleneck area; a fast and localized speed control is desired to create input flows that maximize the bottleneck throughput. Such speed control might be difficult for human drivers to implement. In this sense, the predictive VSA algorithm is expected to perform best with the automated speed controller, such as CACC. Another direction of improvement combines the on-ramp metering and VSA control. With the RM, the disturbance caused by the on-ramp traffic becomes controllable. The VSA algorithm can generate speed commands based on the metering rate. Since the combined implementation of VSA and RM does not require frequent change of a vehicle's reference speed in the bottleneck area, it is easier for human drivers to follow.

CHAPTER 6. SUMMARY AND RECOMMENDATIONS

This document presented the modeling approach the team used to depict the CMDV drivers' compliance behaviors in response to the I2V-based VSA in mixed traffic. The CMDV drivers' speed adaptation due to VSA was captured by a stimulus-response model, which allowed drivers' desired speed to change based on the VSA level. A random variable was used to model the fluctuation of the desired speed in the traffic flow. The field VSA test data were used for the model calibration and validation so that the CMDV drivers' speed adaptations represent real-world drivers' speed patterns under the influence of VSA control. The CMDV drivers' compliance was best modeled with empirical distributions of two speed groups: a low-speed group when the VSA is under 35 mph and a high-speed group when the VSA is above 35 mph. The calibration and validation process adopted distribution fitting and hypothesis test approaches to determine the parameters of the empirical distribution. The two-sample KS test was conducted to show the goodness of fit of those empirical cumulative distribution functions with the real-world data. The CMDV model then applied the validated compliance distribution to evaluate the effectiveness of CMDVs on the freeway corridor operations.

The model calibration and validation data sets represent drivers' responses to the VSA displayed on the roadside message sign instead of directly fed into the vehicle. Admittedly, such feedback will have less influence on the driver than in-vehicle displays. For this reason, this research designed a sensitivity analysis to explore the effectiveness of the CMDV speed adaptation under various CMDV market penetrations and driver compliance levels. In addition, the sensitivity analysis identified the impact of using automated speed controllers with the I2V-based VSA algorithm.

The analysis results indicate that the I2V-based VSA control could have substantial effects on the freeway corridor when the CMDV market penetration is 10–40 percent. With the advisory speed, the average speed and variation of the speed remained similar to the no-control case but the vehicle fuel efficiency increased 2–5 percent, depending on the results of different energy models. These results suggest that the speed adaptation of a few connected drivers could substantially change the traffic flow pattern, leading to more energy efficient traffic flow. As the CMDV market penetration further increased, the reduction of the speed variation and the improvement of the fuel efficiency stabilized. When the CMDV market penetration reached 100 percent, the average speed decreased by about 1 percent and the vehicle fuel economy increased 5–6 percent.

The performance of the VSA algorithm was not sensitive to small changes in the driver compliance level. However, the traffic flow patterns changed significantly when the CMDV drivers fully complied with the VSA. The full compliance brought about 2–3 percent extra benefit on vehicle fuel efficiency. If the VSA algorithm could generate advisory speed based on the predicted traffic conditions, the effects of CMDV should become further increased. When the VSA was implemented with CACC, the CACC controller could perfectly adopt the advisory speed as the reference speed. Nonetheless, adding the VSA algorithm to the CACC vehicles did not bring notable benefits to the freeway corridor (e.g., 5–6 percent increase of the energy efficiency) because the VSA controller tended to underuse the bottleneck capacity due to its delayed response to the traffic variations. To address this shortcoming, the team recommends a predictive VSA algorithm or the combined application of VSA and RM.

This report details the implementation of the proposed CMDV model in a simulation framework that also has a calibrated HV, AV, and CAV model. The results detailed in this report were obtained using the Aimsun modeling platform. However, the model logics are open source, and the information in this document may be helpful for researchers and analysts to implement the modeling framework in their customized tools of choice.

REFERENCES

Ciuffo, B., Punzo, V., & Montanino, M. (2012). Thirty years of Gipps' car-following model: Applications, developments, and new features. Transportation research record, 2315(1), 89-99.

Farah, H., Koutsopoulos, H. N., Saifuzzaman, M., Kölbl, R., Fuchs, S., & Bankosegger, D. (2012). Evaluation of the effect of cooperative infrastructure-to-vehicle systems on driver behavior. Transportation research part C: emerging technologies, 21(1), 42-56.

Google®. 2017. Google® Earth™, Mountain View, CA, obtained from: https://www.google.com/earth, last accessed [DATE].

Kan, X. D., Xiao, L., Liu, H., Wang, M., Schakel, W. J., Lu, X. Y., ... & Ferlis, R. A. (2019). Cross-comparison and calibration of two microscopic traffic simulation models for complex freeway corridors with dedicated lanes. Journal of Advanced Transportation, 2019.

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

Liu, H. (2018). Using Cooperative Adaptive Cruise Control (CACC) to Form High-Performance Vehicle Streams. Microscopic Traffic Modeling.

Liu, H., Kan, X. D., Shladover, S. E., Lu, X. Y., & Ferlis, R. E. (2018). Modeling impacts of cooperative adaptive cruise control on mixed traffic flow in multi-lane freeway facilities. Transportation Research Part C: Emerging Technologies, 95, 261-279.

Liu, H., Wei, H., Zuo, T., Li, Z., & Yang, Y. J. (2017). Fine-tuning ADAS algorithm parameters for optimizing traffic safety and mobility in connected vehicle environment. Transportation research part C: emerging technologies, 76, 132-149.

Lu, X. Y., Spring, J., Wu, C. J., Nelson, D., & Kan, Y. (2019). Field Experiment of Variable Speed Advisory (VSA) Final Report.

Lu, X., Liu, H., Li, X., Li, Q., Mahmassani, H., Talebpour, A., Hosseini, M., Huang, Z., Hale, D. K., and Shladover, S. E. (Forthcoming). Developing Analysis, Modeling, and Simulation Tools for Connected and Automated Vehicle Applications. FHWA.

Milanés, Vicente, and Steven E. Shladover. (2014). Modeling cooperative and autonomous adaptive cruise control dynamic responses using experimental data. Transportation Research Part C: Emerging Technologies 48: 285-300.

Newell, G. F. (2002). A simplified car-following theory: a lower order model. Transportation Research Part B: Methodological, 36(3), 195-205.

Rakha, H. A., Ahn, K., Moran, K., Saerens, B., & Van den Bulck, E. (2011). Virginia tech comprehensive power-based fuel consumption model: model development and testing. Transportation Research Part D: Transport and Environment, 16(7), 492-503.

Ramezani, H., Lu, X. Y., & Shladover, S. E. (2019). Calibration of Motor Vehicle Emission Simulator (MOVES) to Incorporate Effect of Truck Platooning (No. 19-05831). " Transportation Research Board 98th Annual Meeting. Washington, D.C. January 13-17, 2019.

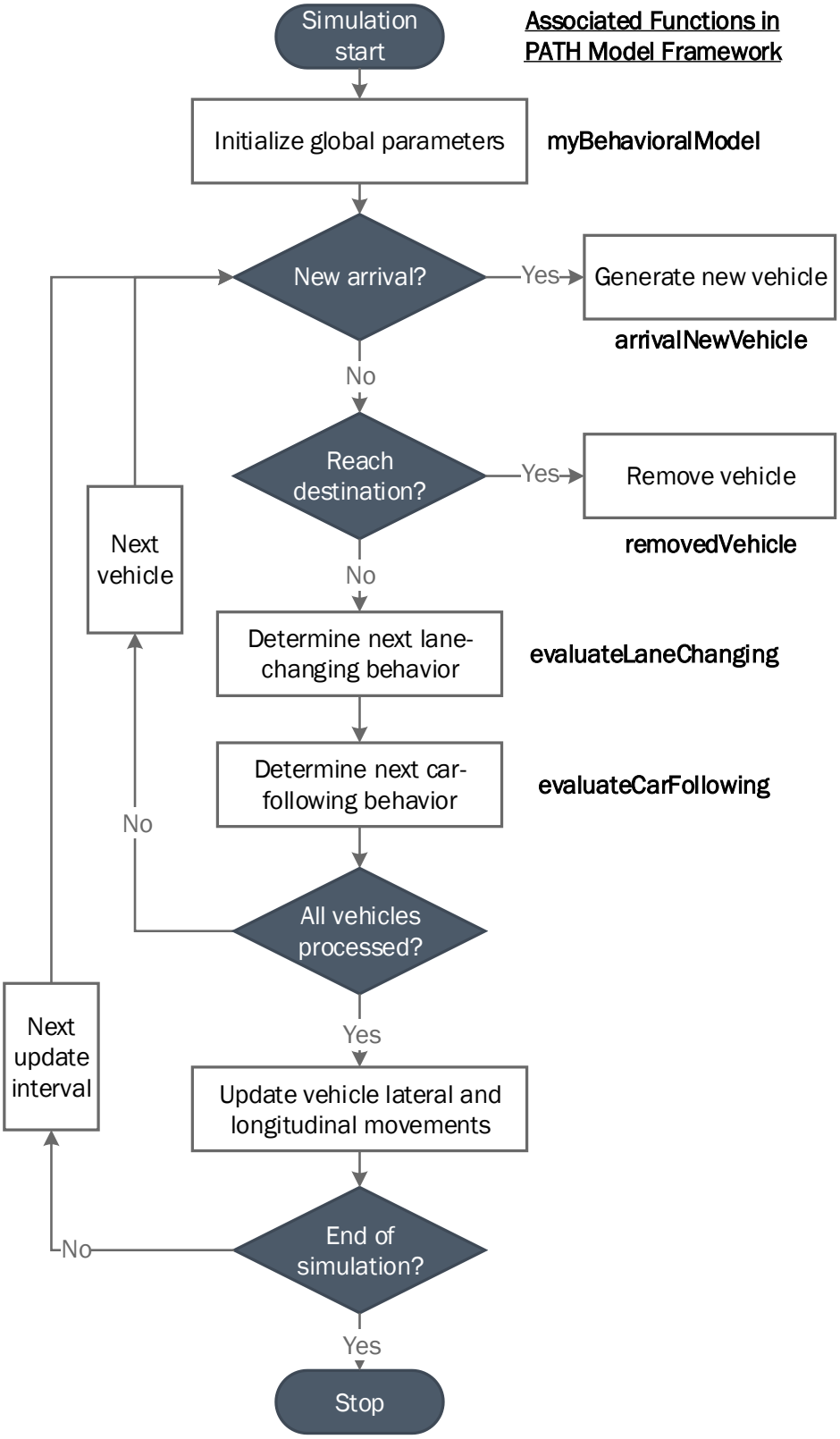
Spyropoulou, I. K., Karlaftis, M. G., & Reed, N. (2014). Intelligent Speed Adaptation and driving speed: Effects of different system HMI functionalities. Transportation research part F: traffic psychology and behaviour, 24, 39-49.

Yeo, H., Skabardonis, A., Halkias, J., Colyar, J., & Alexiadis, V. (2008). Oversaturated freeway flow algorithm for use in next generation simulation. Transportation Research Record, 2088(1), 68-79.

APPENDIX A. FUNCTIONS FOR SIMULATION PROCESS CONTROL

This section and the following sections give detailed descriptions of the simulation functions used by the modeling framework within Aimsun MicroSDK and API. The functions are arranged in alphabetical order to allow users to easily locate the function description with the function name. Each function item provides the function syntax, functionality description, input and output definitions, sub-functions, and pseudo code. The information can help users understand how the functions are implemented in the modeling framework, which is essential for developing similar algorithms in different simulation platforms. The pseudo code section describes the critical decision and computation processes involved in each function while omitting many auxiliary steps. For example, in a lane-changing gap searching process, the pseudo code only provides information on gap estimation and the gap acceptance process, which are the most important components in a gap searching process. It does not, however, include apparent decision steps, such as "a left lane-changing vehicle should not check the right lane," or "the vehicle length needs to be counted in the gap estimation." Such a simplification of the pseudo code is expected to provide clearer logic flow of each function rather than including every step of the function.

The BM class controls the implementation process of the simulation logic. The BM class used in the model is named mybehavioralModel. The logic flow of this class is demonstrated in figure 21. In the figure, the bold notations indicate the model functions that correspond to the component in the logic flow. Section A.1 provides detailed information on these functions, which must use several supportive functions for controlling the simulation flow. Section A.2 gives descriptions of the supportive functions.



Source: FHWA

Figure 32. Diagram. Logic flow of the PATH simulation algorithm.

A1. MAJOR FUNCTIONS FOR SIMULATION PROCESS CONTROL

arrivalNewVehicle

Syntax

myVehicleDef* arrivalNewVehicle(void *handlerVehicle, unsigned short idHandler, bool isFictitiousVeh)

Description

This function generates new vehicles based on exponential distribution. Once a new vehicle is released into the network, the PATH framework will call *AdjustArrivalVehicle_New* to set its initial position and speed. If the vehicle is a CACC vehicle, the PATH framework also determines the CACC string operation status for the vehicle. The detailed information regarding *AdjustArrivalVehicle_New* is given in Appendix B.

Inputs Arguments

handlerVehicle: a pointer to the new vehicle.

idHandler: an ID used by Aimsun to declare a new vehicle object.

isFictitiousVeh: a flagger indicating if the created vehicle is a real vehicle or fictitious vehicle, such as traffic control devices.

Output Arguments

A myVehiclDef object that represents the new arriving vehicle.

evaluateCarFollowing

Syntax

bool mybehavioralModel::evaluateCarFollowing(A2SimVehicle *vehicle, double &newpos, double &newspeed)

Description

This function computes the updated position and speed for a subject vehicle based on the PATH car-following logic. It also calculates lane-changing desire and determines the target lane for the subject vehicle if it needs to make a lane change. The subject vehicle will move to the updated position with the updated speed at the end of the simulation interval. If a lane-changing maneuver is deemed feasible, the PATH model framework will perform the lane change in the next simulation interval with the function *evaluateLaneChanging*. Details of the lane-changing and car-following algorithms is discussed in Appendix C and D.

Inputs Arguments

vehicle: a pointer to the subject vehicle.

newpos: new position of the subject vehicle with respect to the beginning of the current road link.

newspeed: new speed of the subject vehicle.

Output Arguments

True: tells the simulation tool that the car-following maneuvers are handled by the PATH model.

False: tells the simulation tool that the car-following maneuvers are handled by the default models.

Sub-Functions

setPara4NewVeh()

AdjustArrivalVehicle_New()

getSectionInfo()

getLeader()

getAroundSpeed()

getAroundLeaderFollowers()

NGSIMPlusACC_CACC_V2VAHM()

NGSIMPlusACC(false)

RunNGSIM()

Pseudo-Code

If (the subject vehicle just enters the network):

- Set parameters for the new vehicle: setPara4NewVeh().
- Set initial speed and position for the new vehicle: AdjustArrivalVehicle_New().

Else:

- Get road link information (e.g., link ID, length, number of lanes, and ID of the next link): getSectionInfo().
- Get traffic information for the computation of car-following model: getLeader(), getAroundSpeed(), getAroundLeaderFollowers().
- Run the car-following model (including car-following model that describes vehicle behaviors before making a lane-changing maneuver) based on the vehicle type: NGSIMPlusACC_CACC_V2VAHM(), NGSIMPlusACC(false), RunNGSIM().
- Set the new position speed value.

Return true

evaluateLaneChanging

Syntax

bool mybehavioralModel::evaluateLaneChanging(A2SimVehicle *vehicle, int threadId)

Description

This function moves a subject vehicle to the target lane if the vehicle needs to make a lane-changing maneuver and identifies an acceptable gap in the previous simulation interval with the function *evaluateCarFollowing*.

Inputs Arguments

Vehicle: a pointer to the vehicle potentially in need of a lane change.

ThreadId: an ID used by the Aimsun internal algorithm.

Output Arguments

True: lane changes are handled by the PATH model.

False: lane changes are handled by the default model.

mybehavioralModel

Syntax

mybehavioralModel();

Description

This is the constructor function for the mybehavioralModel class. It is executed at the beginning of each simulation run. The function reads user-specified parameters from C:\CACC_Simu_Data\ParameterSet.txt and imports them into the simulation environment.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

ReadExternalParameters()

SetExternalParameters()

ReadHOVSetting()

removedVehicle

Syntax

void mybehavioralModel::removedVehicle(void *handlerVehicle, unsigned short idHandler, A2SimVehicle * a2simVeh)

Description

This function removes vehicles that have reached their destinations in the current update interval.

Inputs Arguments

handlerVehicle: a pointer used by the Aimsun internal algorithm.

idHandler: an ID used by Aimsun to remove a vehicle object.

a2simVeh: a pointer to the vehicle to be removed.

Output Arguments

None.

A2. SUPPORTIVE FUNCTIONS

createFreeFlowSpeed

Syntax

double myVehicleDef::createFreeFlowSpeed(bool ACCCACC)

Description

This function creates free flow speed for a road segment considering the speed friction—a subject driver will not drive much faster than vehicles in the adjacent lanes, even if he or she could do so in the current lane.

Inputs Arguments

ACCCACC: a flagger indicating if there are ACC/CACC vehicles on the concerned road segment.

Output Arguments

Free flow speed for the road segment.

getAroundLeaderFollowers

Syntax

void myVehicleDef::getAroundLeaderFollowers()

Description

This function gets the pointers of the lead and lag vehicles in the adjacent lane(s) to the current lane.

Inputs Arguments

None.

Output Arguments

None.

getAroundSpeed

Syntax

void myVehicleDef::getAroundSpeed()

Description

This function gets the average speeds of traffic in the immediate left and right lanes. The average speeds are computed over a pre-specified road length, among a pre-specified number of vehicles (i.e., n_scan). If more than n_scan vehicles are within the road segment, only the closest n_scan vehicles will be considered. If there are only n vehicles in the road segment (n < n_scan), it is assumed that the other (n_scan-n) vehicles are traveling in the free flow speed. The free flow speed is obtained via *createFreeFlowSpeed*.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

createFreeFlowSpeed ()

getLeader

Syntax

const A2SimVehicle* myVehicleDef::getLeader()

Description

This function returns pointer to the front vehicle of a subject vehicle.

Inputs Arguments

None.

Output Arguments

Pointer to the front vehicle.

getSectionInfo

Syntax

void myVehicleDef::getSectionInfo()

Description

This function gets basic information for a subject vehicle’s current section.

Inputs Arguments

None.

Output Arguments

None.

ReadExternalParameters

Syntax

void mybehavioralModel::ReadExternalParameters()

Description

This function reads global simulation parameters from C:\CACC_Simu_Data\ParameterSet.txt. The user will need to create the file if it does not exist.

Inputs Arguments

None.

Output Arguments

None.

ReadHOVSetting

Syntax

void mybehavioralModel::ReadHOVSetting()

Description

This function reads HOV information from the Aimsun network file. The network file should provide the HOV information when the HOV operation is active. Such information includes the start/end of HOV operation, and the percentage of HOV vehicles. The vehicle generation algorithm (described in Appendix B) will generate HOVs based on the percentage. Upon entering the network, an HOV will be motivated to merge into the HOV lane. The lane-changing algorithm presented in Appendix C captures the behaviors of the HOVs.

Inputs Arguments

None.

Output Arguments

None.

SetExternalParameters

Syntax

void mybehavioralModel::SetExternalParameters()

Description

This function imports the user-specified simulation parameters (i.e., parameters in ParameterSet.txt) into the simulation environment.

Inputs Arguments

None.

Output Arguments

None.

APPENDIX B. FUNCTIONS FOR NEW VEHICLE GENERATION

In the simulation algorithm, the new vehicle creation and LC/CF update rely on functions in the simulated vehicle (SV) class. The SV class in the modeling framework is named myVehicleDef. When a new vehicle is created, the modeling framework first calls SetPara4NewVeh to specify the initial parameters (e.g., desired headway, desired speed, and reaction time) to the vehicle. Afterwards, the simulation model determines its initial speed and location by calling *AdjustArrivalVehicle_New*. Descriptions for functions for new vehicle generation are provided below.

B.1 MAJOR FUNCTIONS FOR VEHICLE GENERATION

AdjustArrivalVehicle_New

Syntax

myVehicleDef::PositionSpeed myVehicleDef::AdjustArrivalVehicle_New()

Description

This function adjusts the speed and position of a new arrival vehicle. The function selects a default starting point (i.e., 300 meters downstream from the start of the entering section) in the absence of a front vehicle. If a front vehicle is present, the function first computes the equilibrium position for the new vehicle, such that the speed of the subject vehicle is equal to the speed of the preceding vehicle and the acceleration is zero. If the subject vehicle is a CACC vehicle, the function also checks the CACC string status. When the preceding vehicle is a CACC or VAD vehicle, the function allows the subject vehicle to join the CACC string.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

GippsDecelerationTerm ()

GetEquPosition ()

NGSIM_Speed ()

Pseudo code

If (leader exists and is not fictitious, leader vehicle is not subject vehicle, leader and subject are in the same section and lane, leader position further than subject vehicle position):

- Set v = min(0, leader's getSpeed(0), freeflowspeed, averge speed ahead)
- If (vehicle type is CACC):
    - If (leader vehicle type is CACC):

        - If (leader vehicle is not in platoon):
            - Make the CF mode of leader and subject vehicle CACC.
        - If (leader vehicle is in a platoon of maximum vehicle numbers):
            - Make the subject vehicle the leader of a new CACC platoon.
        - Else:
            - Make the subject vehicle a follower of leader vehicle's platoon.
    - Else if (leader vehicle type is VAD-equipped):
        - Make the CF mode of leader and subject vehicle CACC.
    - Else:
        - Make the CF mode of subject vehicle CACC, following an ACC vehicle.
        - Compute eq_pos, equilibrium position with minimum(GetEquPosition(), initial starting position).
    - Else if (vehicle type is ACC):
        - Make the CF mode of subject vehicle ACC.
        - Compute eq_pos, equilibrium position with minimum(GetEquPosition(), initial starting position).
- Else:
    - Compute eq_pos, equilibrium position with minimum(GetEquPosition(), initial starting position).
    - With the computed eq_pos, compute CF speed with PATH CF model.
    - If (CF speed < v):
        - The subject vehicle needs to decelerate with the computed eq_pos.
        - Set eq_pos = eq_pos – 0.5 and recalculate CF speed with PATH CF model.
        - Go back to the speed comparison.
    - Else:
        - Obtain eq_pos.

If (eq_pos > 0 and v > 0):

- Return: the position and speed at eq_pos and speed v, respectively.

Else:

- Report error, remove the subject vehicle.

NGSIM_Speed

Syntax

myVehicleDef::PositionSpeed myVehicleDef::NGSIM_Speed(double x, double v)

Description

This function is used to determine the initial position of a new vehicle. It calculates the speed of the new vehicle in the next simulation interval, with the assumption that the subject vehicle's location is at *x* and its current speed is *v*. If the output speed is larger than the current speed of the vehicle, it indicates that the subject vehicle is downstream from the equilibrium position (e.g., the position where the speed is equal to the speed of the preceding vehicle and the acceleration is zero).

Inputs Arguments

x: assumed position of the subject vehicle.

v: assumed current speed of the subject vehicle.

Output Arguments

An estimated speed based on the assumed position and speed.

Pseudo code

- Compute theta = Theta of the Gipps model * reaction time.
- Compute the maximum speed with Gipps safety criterion as v_after_tau with GippsDecelerationTerm(), which is described in Appendix C.
- Compute the safe acceleration: max_a = min(max acceleration, (v_after_tau – current speed) / reaction time.
- Compute the car-following acceleration based on Newell model: newell_a = min(max acceleration, min(min acceleration, (current headway / desired headway – current speed) / (0.5*desired headway)).
- Compute free acceleration with IDM free flow model: min_a = max acceleration * (1-(current speed / free flow speed)^IDM coefficient).
- Compute acc_target = min(min_a, newell_a, max_a).
- Update acc = acc_target.
- Get current acceleration as current_acc.
- If (vehicle is a newly arrived vehicle):
    - Set acceleration level acc = 0.
- Else:
    - acc = (current_acc + (acc_target – current_acc) / acceleration smoothing coefficient.

- Compute Vel = max(0, current speed + acc * timestep).
- Return (Vel + v)/2.

SetPara4NewVeh

Syntax

void myVehicleDef:: SetPara4NewVeh(myVehicleDef* res)

Description

For creating a new vehicle, Aimsun needs to call the arrivalNewVehicle function. Once the function is called, it will execute the sub-functions, listed in table 1. These sub-functions set up parameters required to determine the car-following and lane-changing patterns after the new vehicle enters the simulation network.

Inputs Arguments

res: a pointer to the target vehicle.

Output Arguments

None.

Sub-functions

This function calls a series of sub-functions for specifying the initial parameter levels. The sub-functions are listed in table 11.

Table 12. Sub-Functions executed for creating a new vehicle.

ID	FUNCTIONS	COMMENTS
1	myVehicleDef::setJamGap	The jam gap is a normal distributed random number.
2	myVehicleDef::setMode	Set up driving mode. The initial driving mode is normal car-following (CF).
3	myVehicleDef::setReactionTime	The reaction time is a normal distributed random number.
4	mybehavioralModel:: setHeadwayTime()	The headway is a normal distributed random number.
5	myVehicleDef::setGippsTheta	Theta used in Gipps model. It represents an extra response delay time.
6	myVehicleDef::setEstimateLeaderDec Coeff	A coefficient representing the uncertainty when estimating the max deceleration of the leading vehicle.

ID	FUNCTIONS	COMMENTS
7	myVehicleDef::setAccSmoothCoef	The acceleration smooth coefficient is used when updating vehicle acceleration. It prevents jerks in the acceleration profile.
8	Setup alpha, beta, relaxation, ACF_Steps, and ACF_Step	Parameters for adjusting behavior parameters: alpha for jam gap, beta for reaction time, relaxation for desired headway. ACF_Steps is the total number of time steps of the after lane-changing car-following state. ACF_Step is the elapsed number of steps in the ACF state.
9	myVehicleDef::setE	E and T are parameters used for determining the mandatory LC desire.
10	myVehicleDef::setT	
11	myVehicleDef::setMinTimeBtwLcs	Minimum time between two consecutive lane changes.
12	myVehicleDef::setPoliteness	A parameter to specify if the driver is willing to create a gap for a driver in mandatory LC.
13	myVehicleDef::setRandomPoliteness	Politeness threshold. A driver will yield if his or her politeness is greater than the threshold.
14	myVehicleDef::setPolitenessOptional	A parameter to specify if the driver is willing to create a gap for a driver in non-mandatory LC.
15	myVehicleDef::setRandomPoliteness Optional	Politeness threshold.
16	myVehicleDef::setFrictionCoef	Friction coefficient used for determining the free flow speed in different lanes.
17	myVehicleDef::setGapAcceptanceMo del	ACC based model vs. non-ACC based model. The ACC/CACC vehicles and regular vehicles have different gap acceptance behaviors.
18	myVehicleDef::setDLCScanRange	A scan range for determining the average speed in adjacent lanes.
19	myVehicleDef::setDLCScanNoCars	The number of vehicles scanned for determining the speed of the adjacent lanes.
20	myVehicleDef::setAccExp	Coefficient used in the free flow component of the IDM.
21	myVehicleDef::setLaneChangeDesire Thrd	LC desire threshold. If a driver's LC desire is larger than the threshold, he or she will enter the BLC driving mode.
22	myVehicleDef::setDLCWeight	Weight of the discretionary LC desire when combining the desire of DLC and MLC.

ID	FUNCTIONS	COMMENTS
23	myVehicleDef::setDLCForbidZoneBeforeExit	A road segment before the exit ramp. The DLC is not allowed in the segment.
24	myVehicleDef::setRightDLCCoeff	Coefficient to adjust the right DLC. It makes right DLC smaller.
25	myVehicleDef::setLCGapReductionFactor	A parameter used for depicting the relaxation behavior after the completion of a lane change.
26	myVehicleDef::SetUnsequentialMerging	A parameter used for determining CF behaviors before on-ramp merging maneuvers.
27	myVehicleDef::setOffRampE	E and T used for off-ramp exit LC desire.
28	myVehicleDef::setOffRampT	T value used in off-ramp lane changes.
29	myVehicleDef::setPenaltyDLCNoExitLane	Not used in simulation.
30	myVehicleDef::setComfDecDLC	Comfortable deceleration in DLC.
31	myVehicleDef::setComfDecRampLC	Comfortable deceleration in on-/off-ramp LC.
32	myVehicleDef::setRelaxationTime	The time of the ACF state.
33	myVehicleDef::setForwardGapReductionOnRamp	A parameter used for depicting the relaxation behavior after the completion of a lane change.
34	myVehicleDef::setForwardGapReductionDLC	Same as above.
35	myVehicleDef::setForwardGapReductionOffRamp	Same as above.
36	myVehicleDef::setBackwardGapReductionOnRamp	Same as above.
37	myVehicleDef::setBackwardGapReductionOffRamp	Same as above.
38	myVehicleDef::setBackwardGapReductionDLC	Same as above.
39	myVehicleDef::setIncreaseDLCCloseRamp	Not used in simulation.
40	myVehicleDef::setOffRampOverpassAcc	A default of 1 m/s^2 is used as drivers overpass a gap before making LC toward the off-ramps.
41	myVehicleDef::setHOVStart	Start time of the HOV operation.
42	myVehicleDef::setHOVEnd	End time of the HOV operation.
43	myVehicleDef::setHOVIncluded	A flagger indicating if the HOV is implemented.

ID	FUNCTIONS	COMMENTS
44	myVehicleDef::setHOV	A flagger indicating if the new vehicle can use the HOV lane.
45	myVehicleDef::Setup delta	Delta used in the IDM.
46	myVehicleDef::setSourceSection	A flagger indicating if the new vehicle is in the source section.
47	myVehicleDef::setVehID	
48	myVehicleDef::setEarlyLaneKeepDis	2 miles in default. A distance before off-ramp in which a leaving driver will actively look for gaps in the right lane.
49	myVehicleDef::setInitialLeaderId	The initial leader is either -1 or the vehicle itself.
50	myVehicleDef::setNewArrivalAdjust	Indicates that the vehicle is a new arrival vehicle and, thus, requires position and speed adjustment.
51	myVehicleDef::SetVehTypeIDs	Store the vehicle type ID in the vehicle object.

B.2 SUPPORTIVE FUNCTIONS

GetEquPosition

Syntax

double myVehicleDef::GetEquPosition(double leader_pos, double leader_l, double v)

Description

This function obtains the equilibrium position for a subject vehicle based on the front vehicle information. The equilibrium position is a car-following position, in which the subject vehicle and the preceding vehicle have the same speed and the subject vehicle has a zero acceleration.

Inputs Arguments

leader_pos: position of the front vehicle.

leader_l: length of the front vehicle.

v: speed of the subject vehicle.

Output Arguments

Equilibrium position for the subject vehicle.

UpdateLatestArrival

Syntax

int mybehavioralModel::UpdateLatestArrival(int vid, int secid, int lane_id)

Description

This function identifies the ID of the first vehicle in a lane of a section, and is used to determine the leader of a new arrival vehicle.

Inputs Arguments

Vid: ID of a new arrival vehicle that will become the first vehicle on a road section.

secid: ID of the section that the new arrival vehicle enters.

lane_id: lane ID of the new arrival vehicle.

Output Arguments

The ID of the vehicle that leads the new arrival vehicle.

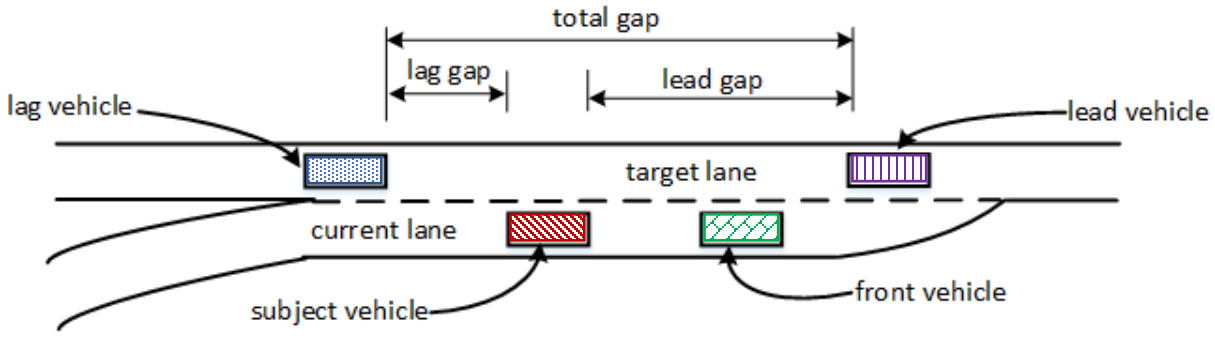
-1, otherwise.

APPENDIX C: VEHICLE CAR-FOLLOWING AND LANE-CHANGING FUNCTIONS

This section describes the CF and LC algorithms of the manually-driven vehicles. The terminologies used in the CF and LC interactions are defined in Figure 33. The overall logic flow of the CF and LC algorithms is shown in Figure 34. In the figure, the italic notations are the function names corresponding to individual boxes in the plot. The CF and LC behaviors are modeled via different driving modes, which are defined as follows:

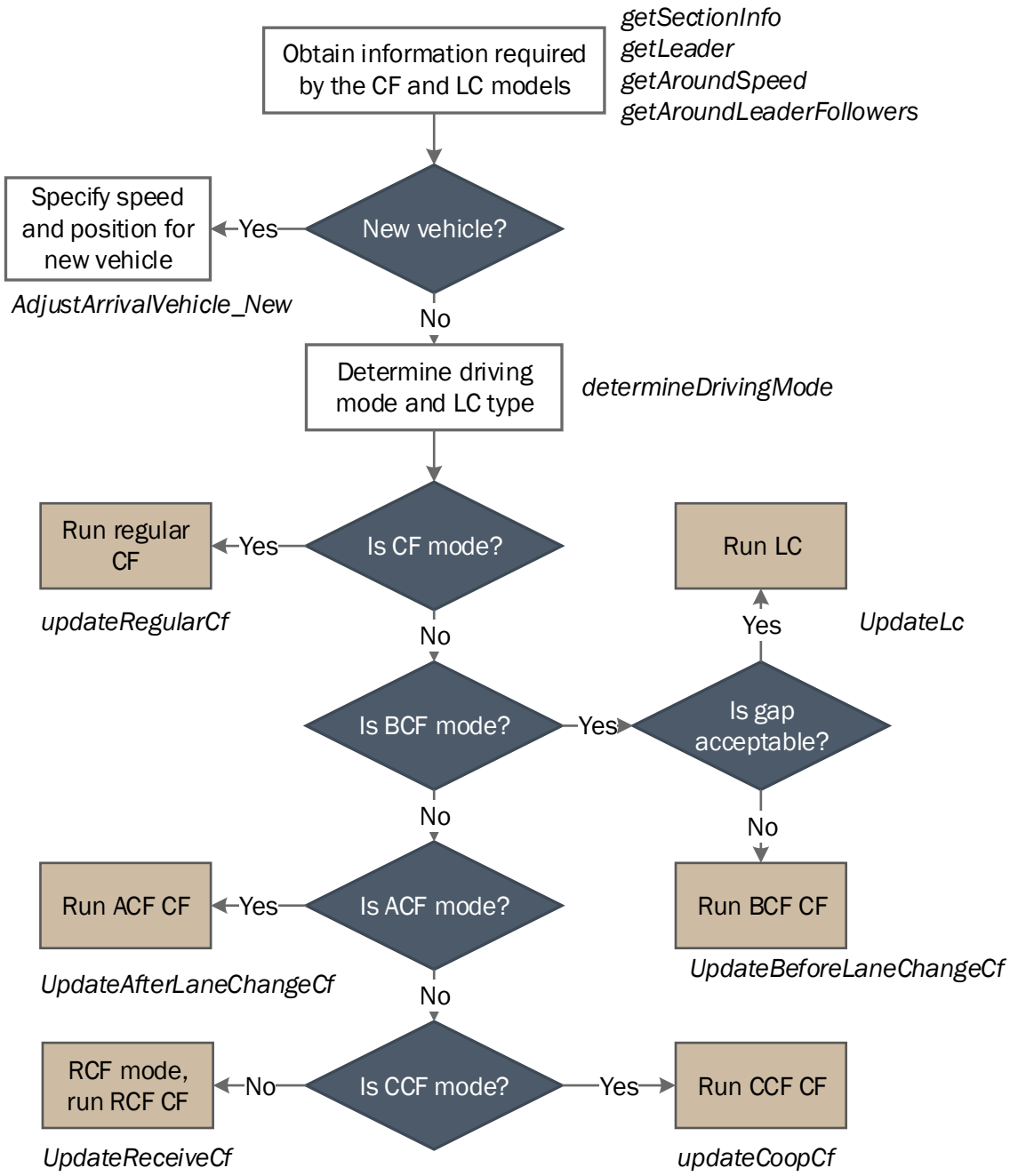
- CF: Regular car following mode.
- LC: Lane change mode, which includes discretionary lane change (DLC), anticipatory lane change (ALC) and mandatory lane change (MLC).
- ACF: After lane changing car following mode (a driver temporarily adopts a short gap after a lane change maneuver).
- BCF: Before lane changing car following mode (a driver speeds up or slows down to align with an acceptable gap in the target lane).
- RCF: Receiving car following mode (a driver temporarily adopts a short gap after a vehicle from the adjacent lane merges in front).
- CCF: Cooperative car following mode.

In Figure 34, the box "determining driving mode" and the processes associated with the BCF mode require more detailed descriptions, as they contain many sub-functions and processes. Figure 35 presents the details regarding the driving mode determination. Under the BCF mode, a modeled driver is actively searching for opportunities to make a lane change. The driver will continue performing the car following if he or she cannot find an acceptable gap in the target lane. The CF behavior varies when the driver's intended LC type is DLC, ALC, or MLC. The logic of the behavior is shown in Figure 36 through 38. Figure 36, in particular, displays the CF functions when the modeled driver intends to make an ALC or MLC towards the freeway exit. Figure 37 shows the CF logic for drivers that make the MLC on an on-ramp. Figure 38 demonstrates the CF logic for drivers that make DLCs. Sections C.1 and C.2 present detailed descriptions for functions in figures 34 through 38.



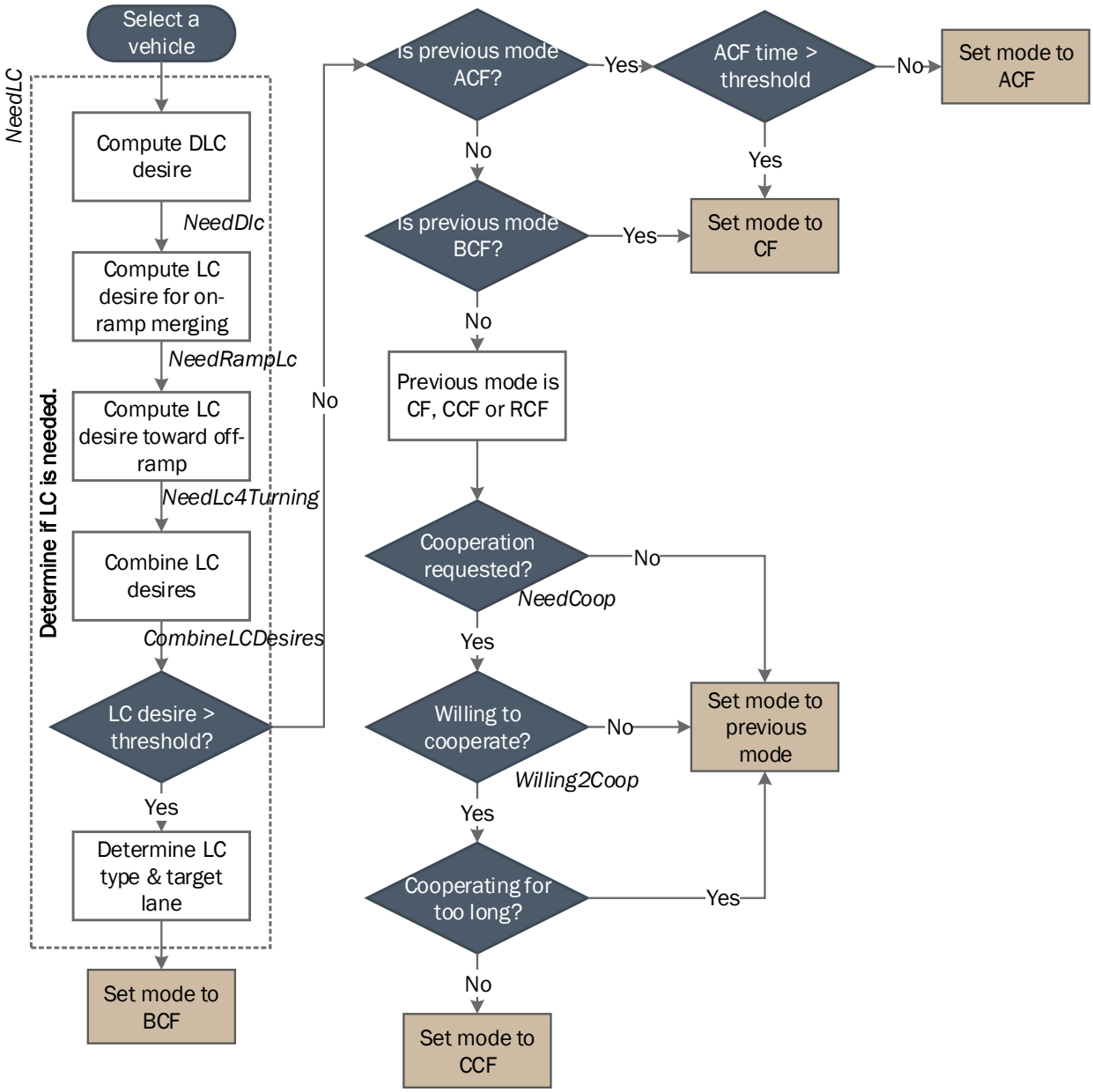
Source: FHWA

Figure 33. Diagram. Vehicles involved in the CF and LC interactions.



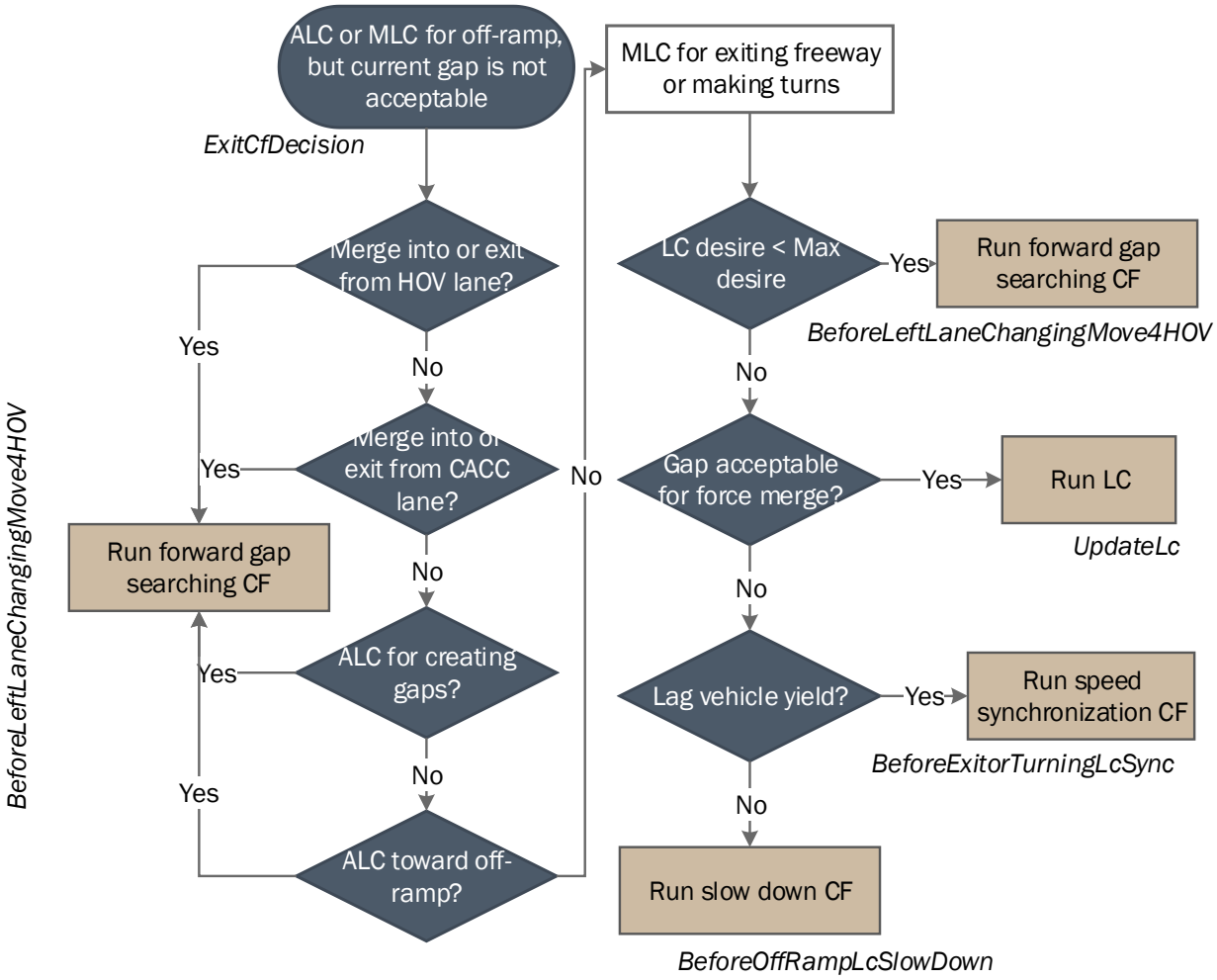
Source: FHWA

Figure 34. Diagram. CF and LC logic for manually-driven vehicles.



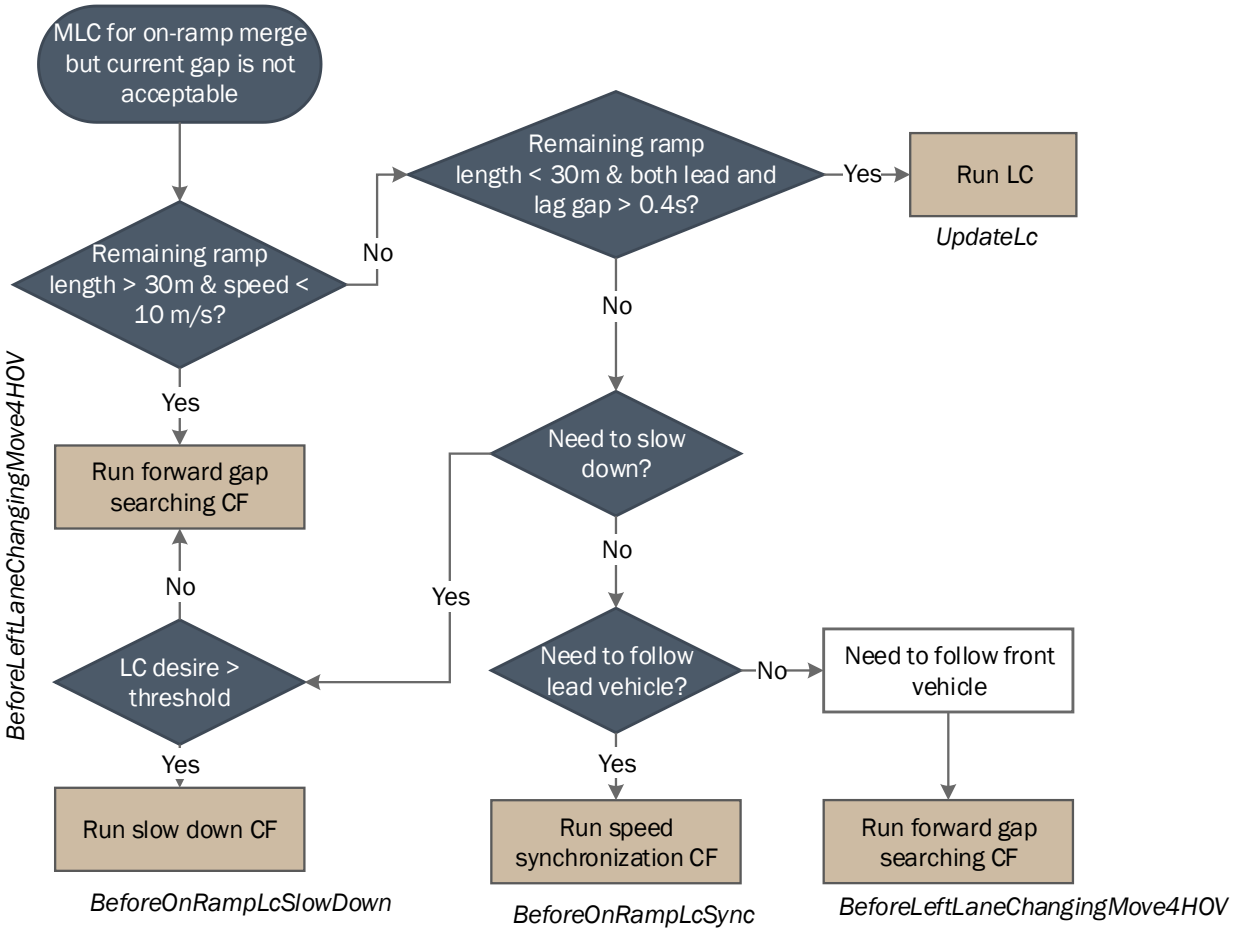
Source: FHWA

Figure 35. Diagram. Driving mode determination for manually-driven vehicles.



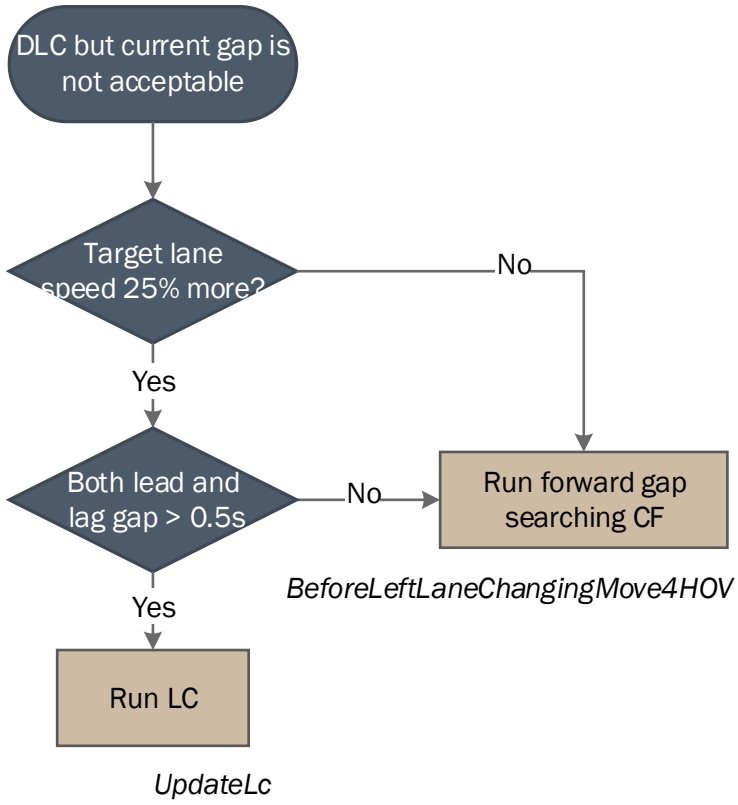
Source: FHWA

Figure 36. Diagram. BCF logic flow for ALC and MLC (off-ramp exiting maneuver).



Source: FHWA

Figure 37. Diagram. BCF logic flow for MLC (on-ramp merging maneuver).



Source: FHWA

Figure 38. Diagram. BCF logic flow for DLC.

C.1 MAJOR FUNCTIONS FOR CAR-FOLLOWING AND LANE-CHANGING ALGORITHMS

AccGapAccepted

Syntax

bool myVehicleDef::AccGapAccepted(double a_L, double a_U, double tau, double headway, double jamGap, double d_leader, double l_leader, double vf, double v, double x, double x_leader, double x_leader_steps_early, double lead_v, double min_headway, double Gap_AC_Thrd, double desire)

Description

This function computes an anticipated acceleration for a subject driver if he or she wants to merge into a gap. If the anticipated acceleration is smaller than the threshold, it indicates the underlying gap is not acceptable.

Inputs Arguments

a_L: maximum deceleration of the subject vehicle.

a_U: maximum acceleration of the subject vehicle.

tau: reaction time.

headway: current spacing headway.

jamGap: the spacing gap between two vehicles in jam traffic.

d_leader: not used.

l_leader: length of the front vehicle.

vf: free flow speed.

v: current speed.

x: current position.

x_leader: position of the front vehicle.

x_leader_steps_early: not used.

lead_v: speed of the front vehicle.

min_headway: desired headway used in the Newell model.

Gap_AC_Thrd: an acceleration threshold to be compared with the anticipated acceleration.

desire: lane change desire

Output Arguments

True: gap is acceptable

False: gap is not acceptable.

Sub-functions

BaseCfModel ().

Pseudo code

Compute new position with a base CF model (BaseCfModel()).

Compute new speed = (new position - current position) / timestep - current speed.

Compute acceleration = (new speed - current speed) / timestep.

If (acceleration is less than acceleration threshold):

- Return False.

Else:

- Return True.

AnticipatedAcc

Syntax

double myVehicleDef::AnticipatedAcc(double a_L, double a_U, double tau, double headway, double jamGap, double d_leader, double l_leader, double vf, double v, double x, double

x_leader, double x_leader_steps_early, double lead_v, double min_headway, double Gap_AC_Thrd, double desire)

Description

This function computes the anticipated acceleration of a subject vehicle after the current update interval.

Inputs Arguments

a_L: maximum deceleration of the subject vehicle.

a_U: maximum acceleration of the subject vehicle.

tau: reaction time.

headway: current spacing headway.

jamGap: the spacing gap between two vehicles in jam traffic.

d_leader: not used.

l_leader: length of the front vehicle.

vf: free flow speed.

v: current speed.

x: current position.

x_leader: position of the front vehicle.

x_leader_steps_early: not used.

lead_v: speed of the front vehicle.

min_headway: desired headway used in the Newell model.

Gap_AC_Thrd: not used.

desire: not used.

Output Arguments

Anticipated acceleration.

Sub-functions

BaseCfModel ().

Pseudo code

- Compute pos with BaseCfModel().
- Compute new speed = 2* (pos - current position) / timestep - current speed.
- Compute acceleration, acc = (new speed - current speed) / timestamp.
- Return acc.
- Set the target lane as last LC target with setLastLCTarget().

- Apply lane changing through its aimsun function.
- Set the current simulation time as the last LC time with setLastLCTime().
- Set this LC type as the last LC type with setLastLCType().

BaseCfModel

Syntax

double myVehicleDef::BaseCfModel(double a_L, double a_U, double tau, double headway, double jamGap, double d_leader, double l_leader, double vf, double v, double x, double x_leader, double x_leader_steps_early, double lead_v, double min_headway)

double myVehicleDef::BaseCfModel(double a_L, double a_U, double tau, double headway, double jamGap, double d_leader, double l_leader, double vf, double v, double x, double x_leader, double x_leader_steps_early, double lead_v, double min_headway, double &target_pos)

Description

This function computes the new position for a subject vehicle using the NGSIM CF model.

Inputs Arguments

a_L: maximum deceleration of the subject vehicle.

a_U: maximum acceleration of the subject vehicle.

tau: reaction time.

headway: current spacing headway.

jamGap: the spacing gap between two vehicles in jam traffic.

d_leader: not used.

l_leader: length of the front vehicle.

vf: free flow speed.

v: current speed.

x: current position.

x_leader: position of the front vehicle.

x_leader_steps_early: not used.

lead_v: speed of the front vehicle.

min_headway: desired headway used in the Newell model.

target_pos: the position that the subject vehicle should reach in the current update interval under optimum conditions (the vehicle might not reach it as the required acceleration/deceleration exceeds the limit or the speed becomes less than 0).

Output Arguments

New position for the subject driver.

Sub-functions

GippsDecelerationTerm ()

Pseudo code

Compute theta = Theta of the Gipps model * reaction time.

Compute the maximum speed with Gipps safety criterion as v_after_tau with GippsDecelerationTerm(), which is described in Appendix C.

Compute the safe acceleration: max_a = min(max acceleration, (v_after_tau – current speed) / reaction time.

Compute the car-following acceleration based on Newell model: newell_a = min(max acceleration, min(min acceleration, (current headway / desired headway – current speed) / (0.5*desired headway)).

Compute free acceleration with IDM free flow model: min_a = max acceleration * (1-(current speed / free flow speed)^IDM coefficient).

Compute acc_target = min(min_a, newell_a, max_a).

Update acc = acc_target.

Get current acceleration as current_acc.

If (vehicle is a newly-arrived vehicle):

- Set acceleration level acc = 0.

Else:

- acc = (current_acc + (acc_target - current_acc) / acceleration smoothing coefficient.

Compute Vel = max(0, current speed + acc * timestep).

Compute x_CF = x + timestep * (Vel + current speed) / 2.

Compute target_v = current speed + acc * timestep.

BeforeExitorTurningLcSlowDown

Syntax

void myVehicleDef::BeforeExitorTurningLcSlowDown()

Description

This function handles the CF movements for a subject driver that needs to make a mandatory lane change toward the off-ramp and decides to slow down for gap searching.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ()

PosCfSkipGap ()

Pseudo code

If (target lane is left):

- Set follower vehicle vehUp = left follower.
- Set leader vehicle vehDown = left leader.

Else:

- Set follow vehicle vehUp = right follower.
- Set leader vehicle vehDown = right leader.

Compute the position when slowing down to skip the current gap, posSlow with PosCfSkipGap() with vehUp.

If (current speed < minimum speed for slowing down to off-ramp):

- Update posSlow = current position + current speed * delta t.

Else:

- Update posSlow = max(posSlow, current position + timestep * (current speed + min speed for slowing down to off-ramp) / 2.

If (there is a leader in the current lane):

- Compute the position with current leader as, posFollowCurrentLeader with PosCF() with leader.
- Update posSlow = min(posFollowCurrentLeader, posSlow).

Set new position as posSlow.

Set new speed as 2 * (posSlow - current position) / timestamp - current speed.

BeforeExitorTurningLcSync

Syntax

myVehicleDef::PositionSpeed myVehicleDef::BeforeExitorTurningLcSync()

Description

This function handles the CF movements for a subject driver that needs to make a mandatory lane change towards the off-ramp and decides to synchronize speed with the lead vehicle.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ()

Pseudo code

If (target lane is left):

- Set follower vehicle vehUp = left follower.
- Set leader vehicle vehDown = left leader.

Else:

- Set follow vehicle vehUp = right follower.
- Set leader vehicle vehDown = right leader.

Compute the position when slowing down to skip the current gap, x_CF_Sync with PosCf() with vehDown.

If (vehDown exists and not fictitious, and current speed < (speed of vehDown – some speed difference threshold)):

- Update X_CF_Sync = max(x_CF_Sync, current position + current speed * delta t).

If (no leader in the current lane):

- Compute position without any front barrier, X_CF_NoSync = current position + current speed * delta t.

Else:

- Compute position with respect to the current leader, X_CF_NoSync with PosCf() with leader in the current lane.

Set new position in pos_speed struct as min(x_CF_NoSync, x_CF_Sync).

Set new speed in pos_speed struct as 2 * (min(x_CF_NoSync - x_CF_Sync) - current position) / timestep - current speed.

Return pos_speed.

BeforeLeftLaneChangingMove4HOV

Syntax

myVehicleDef::PositionSpeed myVehicleDef::BeforeLeftLaneChangingMove4HOV()

Description

This function handles the CF movements for a subject driver that needs to make a lane change towards the HOV lane, but the current gap is not acceptable. With the CF movements of the function, the subject driver will increase its speed as much as possible and actively examine the downstream gaps in the target lane. Such a gap searching behavior is also used for drivers trying to merge into the CACC managed lane, and drivers making lane changes for avoiding conflicts with the merge traffic.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ()

MinCollisionAvoidancePos()

Pseudo code

Get overpassing acceleration acc, with getOffRampOverpassAcc().

Compute speed = min(current speed + acc * delta t, free flow speed).

Compute overpassing position posOverpass = current position + timestep * (current speed + speed) / 2.

If (no leader in the current lane):

- Compute a regular CF position as posCurrentLeader with PosCf().

Else:

- Compute a position that avoids collision with leader as posCurrentLeader with MinCollisionAvoidancePos().

Set new position in pos_speed struct as min(posOverpass, posCurrentLeader).

Set new speed in pos_speed struct as 2 * (min(posOverpass, posCurrentLeader) - current position) / timestep - current speed.

Return pos_speed.

BeforeOffRampLcSlowDown

Syntax

myVehicleDef::PositionSpeed myVehicleDef::BeforeOffRampLcSlowDown()

Description

This function handles the CF movements for a subject driver that needs to make a mandatory lane change toward the off-ramp and decides to slow down for gap seeking.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ()

PosCfSkipGap()

PosCf2EndofRamp()

Pseudo code

If (target lane is left):

- Set follower vehicle vehUp = left follower.
- Set leader vehicle vehDown = left leader.

Else:

- Set follow vehicle vehUp = right follower.
- Set leader vehicle vehDown = right leader.

Compute the position when slowing down to skip the current gap, posSlow with PosCfSkipGap() with vehUp.

If (no leader in the current lane):

- Compute position with respect to the end of the ramp, posCurrentLeader with PosCf2EndofRamp.

Else:

- Compute position with respect to the current leader, posCurrentLeader with PosCf() with leader in the current lane.

Set new position in pos_speed struct as min(posSlow, posCurrentLeader).

Set new speed in pos_speed struct as 2 * (min(posCurrentLeader, posSlow) - posCurrentLeader) - current position) / timestep - current speed.

Return pos_speed.

BeforeOnRampLcSlowDown

Syntax

myVehicleDef::PositionSpeed myVehicleDef::BeforeOnRampLcSlowDown()

Description

This function handles the CF movements for a subject driver that needs to make a mandatory lane change from the on-ramp and decides to slow down for gap seeking.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ();

Pseudo code

If (target lane is left):

- Set follower vehicle vehUp = left follower.
- Set leader vehicle vehDown = left leader.

Else:

- Set follow vehicle vehUp = right follower.
- Set leader vehicle vehDown = right leader.

Compute the position when slowing down to skip the current gap, posSlow with PosCfSkipGap() with vehUp.

If (no leader in the current lane):

- Compute position with respect to the end of the ramp, posCurrentLeader with PosCf2EndofRamp.

Else:

- Compute position with respect to the current leader, posCurrentLeader with PosCf() with leader in the current lane.

Set new position in pos_speed struct as min(posSlow, posCurrentLeader).

Set new speed in pos_speed struct as 2 * (min(posCurrentLeader, posSlow) - posCurrentLeader) - current position) / timestep - current speed.

Return pos_speed.

BeforeOnRampLcSync

Syntax

myVehicleDef::PositionSpeed myVehicleDef::BeforeOnRampLcSync()

Description

This function handles the CF movements for a subject driver that needs to make a mandatory lane change from the on-ramp and decides to synchronize speed with the lead vehicle in the target lane.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ();

Pseudo code

If (target lane is left):

- Set follower vehicle vehUp = left follower.
- Set leader vehicle vehDown = left leader.

Else:

- Set follow vehicle vehUp = right follower.
- Set leader vehicle vehDown = right leader.

Compute the position when slowing down to skip the current gap, x_CF_Sync with PosCf() with vehDown.

Compute speed = max(0, speed + delta t* comfortable deceleration threshold for ramp).

Update speed = min (speed, synchronize speed threshold).

Compute limit of position for synchronization, x_CF_Sync = max(x_CF_sync, current position + delta t * 0.5 * (current speed + speed)).

If (no leader in the current lane):

- Compute position with respect to the end of the ramp, X_CF_NoSync with PosCf2EndofRamp().

Else:

- Compute position with respect to the current leader, X_CF_NoSync with PosCf() with leader in the current lane.

Set new position in pos_speed struct as min(x_CF_NoSync, x_CF_Sync).

Set new speed in pos_speed struct as 2 * (min(x_CF_NoSync, x_CF_Sync) - current position) / delta t - current speed.

Return pos_speed.

CalculateDesireForce

Syntax

double myVehicleDef::CalculateDesireForce(int n_lc, double d_exit, double speed, bool is_for_on_ramp)

Description

This function computes the desire of the mandatory lane change based on the number of lanes it needs to cross, the distance to the end of the on-/off-ramp, and the current speed.

Inputs Arguments

n_lc: number of lanes to cross.

d_exit: distance to the end of the on/off-ramp.

speed: current speed.

is_for_on_ramp: a flagger indicating if the subject vehicle is in an on-ramp.

Output Arguments

The desire for the mandatory lane change.

Sub-functions

DesireEquation()

Pseudo code

If (is_for_on_ramp):

- Calculate distance and time dependent desires with DesireEquation() and parameters related to the on-ramp geometry. DesireEquation() returns para1 and para 2, respectively.

Else:

- Calculate distance and time dependent desires with DesireEquation() and parameters related to the off-ramp geometry. DesireEquation() returns para1 and para 2, respectively.

Return max(para1, para2).

CombineLCDesires

Syntax

bool myVehicleDef::CombineLCDesires()

Description

This function computes the LC desire by combining the desires for both discretionary and mandatory lane change. It also determines the actual LC type and target lane if a lane change is desirable.

Inputs Arguments

None.

Output Arguments

True—a lane change is desirable; false—a lane change is not desirable.

Sub-functions

isLaneChangingPossible ()

DLCDesire ()

Pseudo code

If (currently in node):

- Set LC type to 0.
- Set lane change desire to 0.
- Set target lane to 0.
- Set mandatory type to 0.
- Return False.

If (all LC desire; i.e., mandatory and optional LC desire to left and right, sums to zero):

- Set LC type to 0.
- Set lane change desire to 0.
- Set target lane to 0.
- Set mandatory type to 0.
- Return False.

Else:

- Compute left desire = mandatory desire to left + optional desire to left * weight to optional LC.
- Compute right desire = mandatory desire to right + optional desire to right * weight to optional LC.
- If (mandatory desire to left >0):
    - Update right desire = 0.
- Else if (mandatory desire to left >0):

    - Update left desire = 0.
- Compute desire = max(left desire, right desire).
- Compute target lane = LEFT * (left desire > right desire) + RIGHT * (right desire > left desire).
- If (lane change is not possible; e.g., target lane not exist or target lane has restricted access):
    - Set LC type to 0.
    - Set lane change desire to 0.
    - Set target lane to 0.
    - Set mandatory type to 0.
    - Return False.
- If (currently on CACC and mandatory desire is 0 for both left and right):
    - Desire threshold = max(1, CACC optional LC threshold).
- Else:
    - Desire threshold = LC threshold.
- If (desire > desire threshold):
    - Set type as mandatory.
    - If (left desire > right desire AND left mandatory desire == 0).
        - Set type as discretionary.
    - If (right desire > left desire AND right mandatory desire == 0).
        - Set type as discretionary.
    - If (desire < 0.9, time since last LC <= minimum time between LCs, and the last LC lane is either current or target lane) or (time since last LC <= minimum time between LCs * 3 and the vehicle is targeting LC for its last lane):
        - Set LC type to 0.
        - Set lane change desire to 0.
        - Set target lane to 0.
        - Set mandatory type to 0.
        - Return False.
    - Set LC type as the computed type.
    - Set target lane as the computed target lane.
    - Set lane change desire as the computed desire.
    - Return True.

- Else:
    - Set LC type to 0.
        - Set lane change desire to 0.
        - Set target lane to 0.
        - Set mandatory type to 0.
        - Return False.

DesireEquation

Syntax

void myVehicleDef::DesireEquation(double& para1, double& para2, double dis2End, double time2End, int n_lc, double minE, double minT, double E, double T)

Description

This function determines the LC desire for mandatory lane changes.

Inputs Arguments

dis2End: distance to the end of the on-/off-ramp.

time2End: time to the end of the on-/off-ramp.

n_lc: number of lanes to cross.

minE: minimum distance parameter.

minT: minimum time parameter.

E: maximum distance parameter.

T: maximum time parameter.

Output Arguments

para1: output desire computed based on the distance parameter.

para2: output desire computed based on the time parameter.

Pseudo code

If (dis2End < E or time2End < T):

- If (time2End < minT or dis2End < minE):
    - Para2 = 1.
    - Para1 = 1.
- Else:
    - Para2 = 1-(time2End - minT)/(T-minT).
    - Para1 = 1-(dis2End - minE)/(E_minE).

Else:

- Para1 = Para2 = 0.

Determine2ndLcAfterLc

Syntax

int myVehicleDef::Determine2ndLcAfterLc()

Description

This is an umbrella function that calls *NeedCoop*, *NeedLC* and *isAfterLaneChangeFinish* to determine the driving mode for a subject vehicle when it completes a LC (i.e., the vehicle is currently on ACF mode).

Inputs Arguments

None.

Output Arguments

An integer indicating the driving mode that the subject driver should take.

Sub-functions

NeedCoop()

NeedLC()

isAfterLaneChangeFinish ()

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else:

- If (isAfterLaneChagneFinish()).
  - Set mode as CF and return.
- Else.
  - Return current mode.

determineCoopOrLc

Syntax

int myVehicleDef::determineCoopOrLc()

Description

This is an umbrella function that calls *NeedCoop and NeedLC* to determine if a subject vehicle needs to yield to other lane changers.

Inputs Arguments

None.

Output Arguments

An integer indicating the driving mode that the subject driver should take.

Sub-functions

NeedDlc ()

NeedCoop ()

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else:

- Set mode as CF and return.

determineDrivingMode

Syntax

int myVehicleDef::determineDrivingMode()

Description

This function determines the driving mode of a human driver at the current update interval (see figure C3 for the logic flow). The PATH framework models five driving modes: regular car-following (CF), car-following mode before lane-changing (BCF), car-following mode after lane-changing (ACF), cooperative car-following mode (CCF), and receiving car-following mode (RCF). While the first three modes are self-explanatory, CCF stands for the mode that the subject driver is willing to actively create a gap for the lane changer. RCF mode describes the CF behavior of the driver if a lane changer just merges in front of him or her. Once the driving mode is determined, the simulation framework will use the corresponding CF and LC mechanisms to update the subject driver's movements. A driver's driving mode is affected by his or her driving mode in the previous update interval, the LC motivation, and the (potential) need to yield to other lane changers. For this reason, this function calculates the LC desire, LC type, target lane, and cooperation request for the driver.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

DetermineLcOrMergeOrCoop()

Determine2ndLcAfterLc()

determineCoopOrLc()

DetermineReceiveOrLcOrCoop()

determineGapOrGiveup()

Pseudo code

If (the subject vehicle is within a node, e.g., intersection):

- If previous mode is BCF:
  - Set mode to CF.
- Else:
  - Set mode to previous mode.

Else

- If (current mode is CF):
  - Return: DetermineLcOrMergeOrCoop().
- Else if (current mode is ACF):
  - Return: Determine2ndLcAfterLc().
- Else if (current mode is CCF):
  - Return: determineCoopOrLc().
- Else if (current mode is RCF):
  - Return: DetermineReceiveOrLcOrCoop().
- Else if (current mode is BCF):
  - Return: determineGapOrGiveup().

Else

- Return current mode.

determineGapOrGiveup

Syntax

int myVehicleDef::determineGapOrGiveup()

Description

This function checks if a subject driver still needs LC if his or her driving mode in the previous update interval is BCF.

Inputs Arguments

None.

Output Arguments

An integer indicating the driving mode that the subject driver should take.

Sub-functions

NeedLC ()

NeedCoop()

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else:

- Set mode as CF and return.

DetermineLcOrMergeOrCoop

Syntax

int myVehicleDef::DetermineLcOrMergeOrCoop()

Description

This is an umbrella function that calls *NeedLC* and *NeedCoop* to determine if a subject vehicle needs to make a lane change or yield to other lane changers.

Inputs Arguments

None.

Output Arguments

An integer indicating the driving mode that the subject driver should take.

Sub-functions

NeedDlc ()

NeedCoop ()

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else:

- Return current mode.

DetermineReceiveOrLcOrCoop

Syntax

int myVehicleDef::DetermineReceiveOrLcOrCoop()

Description

This is an umbrella function that calls *NeedLC* and *NeedCoop* to determine if a subject vehicle under the RCF mode needs to make a lane change or yield to other lane changers.

Inputs Arguments

None.

Output Arguments

An integer indicating the driving mode that the subject driver should take.

Sub-functions

NeedLC ()

NeedCoop ().

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else:

- Return current mode.

DisGapAccepted

Syntax

bool myVehicleDef::DisGapAccepted(double a_L, double a_U, double tau, double headway, double jamGap, double d_leader, double l_leader, double vf, double v, double x, double x_leader, double x_leader_steps_early, double lead_v, double min_headway, double Gap_AC_Thrd, double desire, bool on_ramp, bool forward, double acc_self)

Description

This function determines if a gap is acceptable to a subject driver. It calls the Gipps gap acceptance function *GippsGap()*.

Inputs Arguments

a_L: maximum deceleration of the subject vehicle.

a_U: maximum acceleration of the subject vehicle.

tau: reaction time.

headway: current spacing headway.

jamGap: the spacing gap between two vehicles in jam traffic.

d_leader: not used.

l_leader: length of the front vehicle.

vf: free flow speed.

v: current speed.

x: current position.

x_leader: position of the front vehicle.

x_leader_steps_early: not used.

lead_v: speed of the front vehicle.

min_headway: desired headway used in the Newell model.

Gap_AC_Thrd: not used.

desire: not used.

on_ramp: a flagger indicating if the subject vehicle is in an on-ramp.

Forward: a flagger indicatingwhether to check lead gap.

acc_self: acceleration of the subject vehicle.

Output Arguments

True: gap is acceptable.

False: gap is not acceptable.

Sub-functions

GippsGap ().

Pseudo code

Compute Theta = tau * Gipps Theta.

Compute B_estimate = a_L * estimated coefficient for leader deceleration.

Return Gipps gap with GippsGap(), theta and b_estimate.

DLCCFDecision

Syntax

int myVehicleDef::DLCCFDecision()

Description

This function determines the CF behavior (e.g., slow down, synchronize speed or cruise) for a subject driver that wants to make a discretionary lane change.

Inputs Arguments

None.

Output Arguments

RAMP_LANE_CHANGE_FEASIBLE

RAMP_DECISION_FOLLOW

RAMP_DECISION_SLOW_DOWN.

Sub-functions

GapAcceptDecision_Sync_First ();

DLCDesire

Syntax

double myVehicleDef::DLCDesire(int target_lane)

Description

This function computes the LC desire for a subject driver, with the assumption that the driver will make a discretionary lane change towards a target lane.

Inputs Arguments

target_lane: the target lane of the assumed discretionary lane change.

Output Arguments

The LC desire.

Sub-functions

isLaneChangingPossible ();

Pseudo code

If (lane change is not possible or lane is on ramp):

- Return 0.

If (target lane will make vehicle depart from the route):

- Return 0.

If (target lane is left):

- Anticipated speed Ant_speed = left average speed ahead.
- If (left leader exists):
    - Update Ant_speed = min (left leader speed, ant_speed).

Else:

- Ant_speed = right average speed ahead.

Compute speed = max(average speed ahead, minimum optional LC speed).

If (vehicle is not HOV, or HOV lane is not active, or no HOV lane):

- If (Ant_speed < speed):
    - Return 0.
- Else:
    - Desire = max(0, min(1, (ant_speed - speed) / speed)).
    - If (target lane is right):
        - Update desire=desire * right optional LC coefficient.
    - Return desire.

Else:

- If (lane change to left is impossible):
    - Return 0.
- Else
    - If (ant_speed < speed):
        - Return 0.
    - Else:
        - Desire = max(0, min(1, (ant_speed - speed)/speed)).
        - If (target lane is right).
        - Update desire=desire * right optional LC coefficient.

- Return desire.

ExitCfDecision

Syntax

int myVehicleDef::ExitCfDecision()

Description

This function determines the CF behavior for a subject driver that wants to make a lane change toward the off-ramp. It calls GapAcceptDecision_Sync_First() to compute the CF behavior mode.

Inputs Arguments

None.

Output Arguments

0: lane change feasible; other: lane change not feasible.

Sub-functions

GapAcceptDecision_Sync_First()

GapAcceptDecision_Sync_First

Syntax

int myVehicleDef::GapAcceptDecision_Sync_First()

Description

This function checks if the lead and lag gaps are acceptable for a driver trying to perform an MLC (i.e., a LC toward off-ramp or a LC from on-ramp) or ALC. If both the lead and lag gaps are acceptable, the LC is feasible. If the lag gap is acceptable but the lead gap is not, the driver will try to follow the lead vehicle. If the lag gap is not acceptable, the driver will slow down.

Inputs Arguments

None.

Output Arguments

0: RAMP_LANE_CHANGE_FEASIBLE

1: RAMP_DECISION_FOLLOW

-1: RAMP_DECISION_SLOW_DOWN.

Sub-functions

AccGapAccepted ()

DisGapAccepted ()

AnticipatedAcc ()

Pseudo code

If (target lane is left):

- Set follower vehicle vehUp = left follower.
- Set leader vehicle vehDown = left leader.

Else:

- Set follow vehicle vehUp = right follower.
- Set leader vehicle vehDown = right leader.

If (no leader or follower vehicles in the target lane):

- Return 0; i.e., ramp lane change feasible.

Compute downstream_gap_acceptable = True.

Compute upstream_gap_acceptable = True.

If (there is a leader in the target lane):

- Compute max deceleration of target lane leader as d_leader = -1 * (target lane lead vehicle speed)^2 / (2 * min acceleration) * relaxation coefficient.
- If (headway to target lane leader - target lane leader vehicle length <=0):
    - Update downstream_gap_acceptable = False.
- Else:
    - If (gap acceptance model is ACC model):
        - If (AccGapAccepted() is False).
            - Update downstream_gap_acceptable = False.
    - Else:
        - If (Gipps model, DisGapAccepted() is false).
            - Update downstream_gap_acceptable = False.
        - Else:
            - Compute acceleration of subject vehicle, acc_self.
            - If (LC type is optional and acc_self < comfortable deceleration for optional LC).
            - Update downstream_gap_acceptable = False.

If (there is a follower in the target lane):

- Compute max deceleration of target lane leader as d_leader = = -1 * (target lane lead vehicle speed)^2 / (2 * min acceleration) * relaxation coefficient.
- If (headway with the target lane follower - subject vehicle length <=0)

        - Update upstream_gap_acceptable = False.
- Else if (target lane follower speed <=0):
    - Update upstream_gap_acceptable = False.
- Else:
    - If (gap acceptance model is ACC model):
        - If (ACCGapAccepted is False):
            - Update upstream_gap_acceptable = False.
    - Else:
        - If (Gipps model, DisGapAccepted is false):
            - Update upstream_gap_acceptable = False.
        - Else:
            - Compute target lane follower deceleration, follower_d.
            - If (LC type is optional and follower_d < comfortable deceleration for optional LC/2).
                - Update upstream_gap_acceptable = False.

If (upstream_gap_acceptable):

- If (Downstream_gap_acceptable):
    - Return 0; i.e., lane change feasible.
- Else:
    - Return 1; i.e., decision to follow.

Else:

- Return -1; i.e., decision to slow down.

GippsGap

Syntax

bool myVehicleDef::GippsGap(double maxDec, double reaction_time, double theta, double x_leader, double x, double jamGap, double l_leader, double v, double lead_v, double b_estimate)

bool myVehicleDef::GippsGap(double maxDec, double reaction_time, double theta, double x_leader, double x, double jamGap, double l_leader, double v, double lead_v, double b_estimate, bool on_ramp, bool forward, double self_acc)

Description

This function determines if the gap between a subject vehicle and the front vehicle is safe.

Inputs Arguments

maxDec: maximum deceleration.

reaction_time: reaction time of the subject driver.

theta: a coefficient to adjust the reaction time.

x: current position.

x_leader: position of the front vehicle.

jamGap: the spacing gap between two vehicles in jam traffic.

l_leader: length of the front vehicle.

v: current speed.

lead_v: speed of the front vehicle.

b_estimate: a coefficient in the Gipps model.

on_ramp: a flagger indicating if the subject vehicle is in an on-ramp.

forward: a flagger indicating whether to consider the lead gap.

self_acc: acceleration of the subject vehicle.

Output Arguments

True: gap is safe.

False: gap is not safe.

Sub-functions

None.

Pseudo code

Compute leader_stop_x = lead position - (leader speed)^2 / (2 * b_estimate).

Compute follower_stop_x = current position - (current speed)^2 / (2* (min acceleration * current speed * (reaction time + Gipps theta).

If (leader_stop_x – follower_stop_x > (l_leader +jamGap)):

- Return True.

Else:

- Return False.

GippsDecelerationTerm

Syntax

double myVehicleDef::GippsDecelerationTerm (double maxDec, double reaction_time, double theta, double x_leader, double x, double jamGap, double l_leader, double v, double lead_v, double b_estimate)

Description

This function computes the speed for a subject vehicle by using Gipps deceleration term.

Inputs Arguments

maxDec: maximum deceleration.

reaction_time: reaction time of the subject driver.

theta: a coefficient to adjust the reaction time.

x: current position.

x_leader: position of the front vehicle.

jamGap: the spacing gap between two vehicles in jam traffic.

l_leader: length of the front vehicle.

v: current speed.

lead_v: speed of the front vehicle.

b_estimate: a coefficient in the Gipps model.

Output Arguments

Updated speed for the subject vehicle.

Sub-functions

None.

Pseudo code

Compute sq_val = (min acceleration * (reaction time / 2 + Gipps theta))^2 - 2 * min acceleration * current gap - (current speed * reaction time - (leader speed)^2 / b_estimate).

If (sq_val >0):

- Compute V_after_tau = max deceleration * (0.5 * reaction time + theta) + sq_val^0.5.

Else:

- Comopute V_after_tau = 0.

Return V_after_tau.

isLaneChangingPossible

Syntax

bool myVehicleDef::isLaneChangingPossible(int target_lane)

Description

This determines if it is possible for a subject vehicle to merge into the target lane.

Inputs Arguments

target_lane: target lane to merge into.

Output Arguments

True: lane change is possible.

False: lane change is not possible.

Sub-functions

None.

Pseudo code

If (subject vehicle is not HOV, and road section contains HOV and HOV is active, and if the target lane is the left-most lane):

- Return False.

If (subject vehicle at on-ramp):

- If (one lane in this section and currently at no lane changing zone):
    - Return False.
- Else if (two lanes in this section and target lane is right):
    - Return False.

Return Aimsun function A2SimVehicle::isLaneChangePossible(target_lane).

MinCollisionAvoidancePos

Syntax

double myVehicleDef::MinCollisionAvoidancePos(const A2SimVehicle* leader, int shortGap, double beta, double alpha, double Relaxation)

Description

This function determines the minimum CF distance a vehicle should keep for collision avoidance.

Inputs Arguments

leader: pointer to the front vehicle.

shortGap: indicating if shortGap mode should be applied.

beta: reduction factor for the reaction time.

alpha: reduction factor for the jam gap.

Relaxation: reduction factor for the relaxation time.

Output Arguments

The furthest position the subject vehicle can safely travel to at this update interval.

Pseudo code

Compute headway to the leader vehicle as Ref_pos_front.

Compute speed and position of the leader as v leader and x leader, respectively.

Compute speed and position of the subject vehicle as v and x, respectively.

If (leader is not in same road section or |x leader - (ref_pos_front+x)| > 0.1):

- X leader = ref_pos_front + x.

If (shortGap =1):

- r = Relaxation.

Else:

- r = 1.

Compute deceleration of leader as d_leader= -1 * (leader speed)^2 / (2 * min acceleration) * r.

Compute theta = Gipps theta * reaction time * beta.

Compute v_after_tau with GippsDecelerationTerm().

Compute val = (current speed + v_after_tau) / 2 * timestep + current position.

Return val.

NeedCoop

Syntax

bool myVehicleDef::NeedCoop()

Description

This function determines if a subject driver needs to perform the cooperative CF maneuver.

Inputs Arguments

None.

Output Arguments

True: the subject vehicle needs to perform the CCF.

False: the subject vehicle does not need to perform CCF.

Sub-functions

Willing2Coop ()

Pseudo code

If (there is a left leader, and the headway to the left leader > 5, and left leader is in BCF mode to change lane to right and its lane change type is mandatory, and its lane change desire >=0.8):

- If (subject vehicle is willing to cooperate, Willing2Coop() = true with the left leader):
    - Update subject vehicle's CoopRequester as left leader.

If (there is a right leader, and the headway to the left leader > 5, and left leader is in BCF mode to change lane to left and its lane change type is mandatory, and its lane change desire >=0.8, and the right leader's distance to obstacle <60):

- If (subject vehicle is willing to cooperate, Willing2Coop() = true with the right leader):
    - Update subject vehicle's CoopRequester as right leader.

If (there is no CoopRequester, but the last CoopRequester is still in BCF mode for a mandatory LC, and the headway with the last CoopRequester is between 5 and 10):

- If (subject vehicle is willing to cooperate, Willing2Coop() = true with the last CoopRequester):
    - Update subject vehicle's CoopRequest as the last CoopRequester.

If (there is a CoopRequester):

- If (CoopRequester is the last CoopRequester, and its target lane is left, and current simulation time – last cooperation time > max cooperation time):
    - Return False.
- Else if (CoopRequester is last CoopRequester, and its target lane is right, and current simulation time – last cooperation time > max cooperation time):
    - Return False.
- Else:
    - If (CoopRequester is not the last CoopRequester).
        - Set LastCoopRequester as CoopRequester.
        - Set LastCoopTime as current time.
    - Return True.

Else:

- Set CoopRequester as Null.
- Set LastCoopRequester as Null.
- Return False.

NeedDlc

Syntax

bool myVehicleDef::NeedDlc()

Description

This function determines if a subject driver needs to make a discretionary lane change.

Inputs Arguments

None.

Output Arguments

True: the subject driver needs to perform a discretionary lane change.

False: the subject driver does not need to perform a discretionary lane change.

Subfunctions

DLCDesire ()

setLaneChangeDesireOption ()

Pseudo code

If (subject vehicle in CACC mode):

- Return False, no DLC for vehicles in CACC strings.

Else if (subject vehicle just completed a DLC):

- Return False.

Else if (subject vehicle is less than 10 m from the end of the current section):

- Return False.

Compute DLC desire for the left lane with DLCDesire(LEFT).

If (CACC managed lane is active, but the current section does not have access to the managed lane):

- If (subject vehicle is CACC, its current lane is right to the managed lane, and left DLC desire > threshold):
    - Return False, not allowing the vehicle to enter the managed lane.

If (subject vehicle is not a HOV and current section contains a HOV lane):

- If (subject vehicle is in the HOV lane):
    - Set right DLC desire = max desire, making the subject vehicle exit the HOV lane.
- Else if (subject vehicle's current lane is right to the HOV lane and the left DLC desire > threshold):
    - Return False, not allowing the subject vehicle to enter the HOV lane.

Else if (subject vehicle is CACC, and it is in the CACC managed lane):

- Set right DLC desire = DLCDesire(RIGHT)*0.68, decreasing the right LC desire so that the subject vehicle will stay in the managed lane.

Else if (subject vehicle is not CACC, its current lane is right to the CACC managed lane):

- Set left DLC desire = 0 and right DLC desire = DLCDesire(RIGHT), preventing the vehicle from entering the managed lane.

Else if (subject vehicle's next section has an on-ramp):

- Set right DLC = 0. The subject vehicle will not make right lane changes to avoid conflicts with the merge traffic.

Else if (subject vehicle is in a source section):

- Return False, no lane changes in the source section.

If (Max(left DLC desire, right DLC desire) > threshold):

- Return True.

Else:

- Return False.

NeedLC

Syntax

bool myVehicleDef::NeedLC()

Description

This function computes the LC desires for DLC, ALC, and MLC, and combines the LC desires for those LC types. A subject driver would start the gap searching if the calculated LC desire is larger than a threshold.

Inputs Arguments

None.

Output Arguments

True: the subject driver decides to make a LC.

False: the subject driver decides not to make a LC.

Sub-functions

NeedDlc ()

NeedRampLc()

NeedLc4Turning()

CombineLCDesires()

Pseudo code

ResetDesires().

If (currently in node):

- Return False.

If (at on-ramp (not the merge lane) or at source section):

- Return False.

Compute DLC desire with NeedDlc().

Compute On-ramp MLC desire NeedRampLc().

If (NeedRampLc() is False):

- Compute ALC and MLC (off-ramp) desire with NeedLc4Turning().

Return CombineLCDesires().

NeedLc4Turning

Syntax

bool myVehicleDef::NeedLc4Turning()

Description

This function determines if a subject driver needs to make an MLC toward the off-ramp. It also computes the ALC desire for a driver if he or she wants to merge into the HOV lane or the CACC managed lane.

Inputs Arguments

None.

Output Arguments

True: the subject driver needs to perform a mandatory lane change from the on-ramp.

False: the subject driver does not need to perform a mandatory lane change from the on-ramp.

Sub-functions

CalculateDesireForce()

Pseudo code

Determine the LC direction (left or right, depending on the location of the off-ramp or exit).

If (current distance to the next turn > threshold distance):

- If (vehicle is HOV, and there is HOV lane in this section and HOV is active):
    - If (left lane speed higher and subject vehicle is in right lane next to the HOV lane):
        - Set left ALC desire as a uniform distributed random number between [0, 1].
        - Return True.
    - Else if (right lane speed higher and subject vehicle is in the HOV lane):
        - Set right ALC desire as a uniform distributed random number between [0, 1].
        - Return True.
    - Else:
        - Set left ALC desire as a uniform distributed random number between [0, 1].
        - Return True.
- Else if (CACC managed lane is active, vehicle is CACC, vehicle's distance to off-ramp <0, vehicle is not in CACC managed lane, vehicle is at freeway mainline, vehicle is not in acceleration lane, and vehicle is currently not in a CACC string):
    - If (left lane speed higher and subject vehicle is in non-CACC managed lane):
        - Set left ALC desire equal to the current DLC desire.
        - Return True.

- Else if (CACC managed lane is active, vehicle is not connected, and vehicle is in the CACC managed lane):
    - Set right ALC desire as a uniform distributed random number between [0, 1].
    - Return True.
- Else if (this road section has an on-ramp, vehicle is on mainline, vehicle is not in the leftmost lane, left lane is faster than average speed ahead, acceleration lane is congested, vehicle is next to the acceleration lane, and vehicle is not in CACC string):
    - Set left ALC desire as a uniform distributed random number between [0, 1].
    - Return True.
- Else if (next section has an on-ramp, vehicle is close to the next section, vehicle is in mainline, vehicle is not in the left-most lane, left lane is faster than the average speed ahead, and vehicle is not in CACC string):
    - Set left ALC desire as a uniform distributed random number between [0, 1].
    - Return True.
- Else:
    - Return False.

Else:

- Calculate MLC desire = CalculateDesireForce().
- IF (subject vehicle in CACC string, current gap not acceptable, and distance to the next turn is larger than a threshold distance):
    - Return False, letting subject vehicle staying in the CACC string.
- Else:
    - Return True.

NeedRampLc

Syntax

bool myVehicleDef::NeedRampLc()

Description

This function determines if a subject driver needs to make a mandatory lane change from the on-ramp acceleration lane. It also computes the MLC desire for the driver.

Inputs Arguments

None.

Output Arguments

True: the subject driver needs to perform a mandatory lane change from the on-ramp.

False: the subject driver does not need to perform a mandatory lane change from the on-ramp.

Sub-functions

CalculateDesireForce()

Pseudo code

If (currently on an on-ramp acceleration lane):

- Calculate MLC desire with CalculateDesireForce().
- If (current gap is acceptable):
    - Set MCL desire = max desire, allowing the subject vehicle to take the gap.
- Return True.

Else if (lane drop in next section):

- Calculate MLC desire with CalculateDesireForce().
- If (current gap is acceptable):
    - Set MCL desire = max desire, allowing the subject vehicle to take the gap.
- Return True.

Else:

- Return False.

PosCf

Syntax

double myVehicleDef::PosCf(const A2SimVehicle* leader, int shortGap, double beta, double alpha, double relaxation)

Description

This function calls *BaseCfModel()* and computes the new position for a subject vehicle. Parameters needed to run *BaseCfModel()* are directly read from the *myVehicleDef* object.

Inputs Arguments

leader: pointer to the front vehicle.

shortGap: a flagger indicating if adjustment coefficients should be applied to the reaction time, desired headway and the jam gap.

alpha: adjustment coefficient for jam gap.

beta: adjustment coefficient for reaction time.

relaxation: adjustment coefficient for desired headway.

Output Arguments

New position for the subject vehicle.

Sub-functions

BaseCfModel ()

PosCf2EndofExitTurning

Syntax

double myVehicleDef::PosCf2EndofExitTurning()

Description

This function calls *BaseCfModel* and computes the new position for a subject vehicle, assuming that the vehicle needs a lane change toward the off-ramp and is close to the end of the ramp.

Inputs Arguments

None.

Output Arguments

New position for the subject vehicle.

Sub-functions

BaseCfModel ()

PosCf2EndofRamp

Syntax

double myVehicleDef::PosCf2EndofRamp()

Description

This function updates the position for a subject vehicle if the vehicle in the acceleration lane but does not have a preceding vehicle. The position is computed based on the distance between the subject vehicle and the end of the acceleration lane. The new position is calculated using *BaseCfModel()*.

Inputs Arguments

None.

Output Arguments

New position of the subject vehicle.

Sub-functions

BaseCfModel ()

PosCfSkipGap

Syntax

double myVehicleDef::PosCfSkipGap(const A2SimVehicle* potential_leader, bool apply_creep_speed)

Description

This function computes the new position for a lane-changing driver if the driver decides to slow down to skip the current gap and check the next gap upstream.

Inputs Arguments

potential_leader: pointer to the lead vehicle.

apply_creep_speed: a flagger indicating if the subject driver should apply a small creep speed.

Output Arguments

New position for the subject driver.

Sub-functions

BaseCfModel()

Pseudo-code

If (lead vehicle exists in target lane):

- If (MLC for on-ramp vehicles):
    - Compute speed = current speed + comfortable deceleration*update interval.
    - Set target speed as max(speed, current speed, and lowest acceptable speed).
- Else if (ALC or MLC for off-ramp and lag vehicle yields):
    - Compute desired_acc that ensures collision free had the subject vehicle moves in front of the lag vehicle.
    - Compute desired_speed based on the current speed and desired_acc.
    - Compute exit_speed with BaseCfModel.
    - Update speed as min(desired_speed, exit_speed).
- Return new position based on the calculated speed.

Else:

- Set speed as max(3, current speed + (-1)*update interval.
- Return new position based on the calculated speed.

RampCfDecision

Syntax

int myVehicleDef::RampCfDecision()

Description

This function determines the CF and LC movements that a subject driver should take if he or she travels on an on-ramp.

Inputs Arguments

None.

Output Arguments

RAMP_LANE_CHANGE_FEASIBLE

RAMP_DECISION_FOLLOW

RAMP_DECISION_NORMAL_FOLLOW

RAMP_DECISION_SLOW_DOWN.

Sub-functions

GapAcceptDecision_Sync_First ()

Pseudo-code

If (there is no leader):

- Return RAMP_DECISION_NORMAL_FOLLOW.

Else:

- Return GapAcceptDecision_Sync_First().

RunNGSIM

Syntax

myVehicleDef::PositionSpeed myVehicleDef::RunNGSIM(bool mode_predetermined)

Description

This is an umbrella function that first calls *determineDrivingMode()* and then executes the CF and LC functions associated with the identified driving mode. The function only updates the new position and speed for manually vehicles.

Inputs Arguments

mode_predetermined: a flagger indicating if the driving mode of the subject vehicle is pre-specified by other functions. True—driving mode is determined by other functions; false—driving mode needs to be determined by calling *determineDrivingMode*.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

determineDrivingMode ()

updateRegularCf ()

UpdateBeforeLaneChangeCf ()

UpdateAfterLaneChangeCf ()

updateCoopCf ()

UpdateReceiveCf()

UpdateAfterLaneChangeCf

Syntax

myVehicleDef::PositionSpeed myVehicleDef::UpdateAfterLaneChangeCf()

Description

This function handles the CF movements for a subject vehicle under the ACF mode. It calls *PosCf*() to calculates the new speed and position. The reduced headway and gap are used in the calculation.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ();

UpdateBeforeLaneChangeCf

Syntax

myVehicleDef::PositionSpeed myVehicleDef::UpdateBeforeLaneChangeCf()

Description

This is an umbrella function that handles the CF and LC movements for a subject vehicle under the BCF mode (see figures C4 through C6 for the detailed logic flow).

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

SetRiskyRelax()

ExitCfDecision()

UpdateLc()

BeforeLeftLaneChangingMove4HOV()

BeforeOffRampLcSlowDown()

BeforeExitorTurningLcSync()

DLCCFDecision()

RampCfDecision()

BeforeOnRampLcSync()

BeforeOnRampLcSlowDown()

Pseudo code

If (time in BCF mode > time threshold):

- Set a reduced reaction time, desired headway, and jam gap for the subject vehicle, because it has been waiting for the LC for a long time.

If (LC type is ALC or MLC toward off-ramp):

- If (current gap acceptable):
  - Execute LC maneuver with UpdateLc().
- Else if (subject vehicle merge or exit from the HOV lane):
  - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.
- Else if (subject vehicle merge or exit from the CACC managed lane):
  - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.
- Else if (subject vehicle create gap for the merge traffic):
  - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.
- Else if (subject vehicle make anticipatory lane changes for exiting the freeway):
  - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.
- Else:
  - If (LC desire < max desire):
    - If (lag vehicle yield):
      - Update CF with BeforeExitorTurningLcSync(), waiting the lag vehicle to create an acceptable gap.
    - Else:
      - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.
  - Else if (LC desire = max desire):
    - If (lead and lag gap after LC > jam gap):
      - Execute LC maneuver with UpdateLc(), creating a force merge maneuver.
    - Else if (lag vehicle yield):
      - Update CF with BeforeExitorTurningLcSync(), waiting the lag vehicle to create an acceptable gap.
    - Else:
      - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.

Else if (LC type is DLC):

- If (current gap acceptable):
    - Execute LC maneuver with UpdateLc().
- Else if (lead and lag headway after the LC maneuver > 0.5 s, and target lane speed > 125 percent of current lane speed):
    - Execute LC maneuver with UpdateLc(), creating an aggressive DLC to the faster lane.
- Else:
    - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.

Else if (LC type is MLC for on-ramp merge vehicles):

- If (current gap acceptable):
    - Execute LC maneuver with UpdateLc().
- Else if (remaining ramp length > 30 m and current speed < 10 m/s):
    - Update CF with BeforeLeftLaneChangingMove4HOV(), searching forward gaps for possible merge.
- Else if (remaining ramp length <= 30 m and lead and lag headway after the LC maneuver > 0.4 s):
    - Execute LC maneuver with UpdateLc(), creating a force merge maneuver.
- Else if (subject vehicle needs slow down for waiting acceptable gaps):
    - Update CF with BeforeOnRampLcSlowDown(), waiting for acceptable gaps.
- Else if (subject vehicle needs to synchronize speed with the lag vehicle):
    - Update CF with BeforeOnRampLcSync(), waiting for the lag vehicle to create a gap.

updateCoopCf

Syntax

myVehicleDef::PositionSpeed myVehicleDef::updateCoopCf()

Description

This function handles the CF movements for a subject vehicle under the CCF mode. It calls *PosCf()* to calculate the updated speed and position. The function is called twice, taking either the cooperation requester or the preceding vehicle as the leading vehicle. The function returns the lower speed and position level from the two computations.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ()

UpdateLc

Syntax

myVehicleDef::PositionSpeed myVehicleDef::UpdateLc()

Description

This function updates a subject vehicle's longitudinal status (i.e., speed and position) when the vehicle is performing the LC maneuver.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf()

Pseudo code

Update vehicle position and speed with PosCf(), following the lead vehicle in the target lane.

If (lag vehicle exists):

- Set the driving mode of the lag vehicle as RCF.

If (LC completes):

- Set current driving mode as ACF.

UpdateReceiveCf

Syntax

myVehicleDef::PositionSpeed myVehicleDef::UpdateReceiveCf()

Description

This function handles the CF movements for a subject vehicle under the RCF mode. The function calls *PosCf()* to calculate the updated position and speed based on a reduced headway and reaction time.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf ();

updateRegularCf

Syntax

myVehicleDef::PositionSpeed myVehicleDef::updateRegularCf()

Description

This function calls *PosCf()* to compute the new position and speed of an HV after an update interval.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

PosCf()

Willing2Coop

Syntax

bool myVehicleDef::Willing2Coop(myVehicleDef *coop_veh)

Description

This function determines if a subject driver is willing to yield to make a gap for the vehicle that wants to merge ahead. The subject driver will yield if his or her politeness level is greater than a politeness threshold and the time of yielding is smaller than a threshold time.

Inputs Arguments

coop_veh: a pointer to the vehicle that requires the subject vehicle to yield.

Output Arguments

True: the subject driver is willing to yield.

False: the subject driver is not willing to yield.

C.2 SUPPORTIVE FUNCTIONS

ApplyNGSIMModel

Syntax

bool myVehicleDef::ApplyNGSIMModel()

Description

This function tells Aimsun if NGSIM model should be applied to model the movements of a subject vehicle.

Inputs Arguments

None.

Output Arguments

True: apply NGSIM model

False: do not apply NGSIM model.

Sub-functions

None.

BoostOnrampIncentive

Syntax

void myVehicleDef::BoostOnrampIncentive()

Description

This function makes the LC desire 1 for an on-ramp merging lane change if the gap is accepted.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

GapAcceptDecision_Sync_First ()

CrashAvoidancePosition

Syntax

void myVehicleDef::CrashAvoidancePosition(double& velocity, double& pos)

Description

This function checks if a subject vehicle is going to collide with the front vehicle after moving to the input position. If a collision will occur, the function makes the subject vehicle stop.

Inputs Arguments

pos: position of the subject vehicle.

velocity: speed of the subject vehicle.

Output Arguments

Processed position of the subject vehicle. The function does not affect the speed.

Sub-functions

RecordCrashInformation ()

distance2EndAccLane

Syntax

double myVehicleDef::distance2EndAccLane()

Description

This function gets the distance to the end of the on-ramp acceleration lane.

Inputs Arguments

None.

Output Arguments

Distance to the end of the acceleration lane.

Sub-functions

None.

EliminateDlcDesireOutSideRouteLanes

Syntax

void myVehicleDef::EliminateDlcDesireOutSideRouteLanes(int fromlane, int tolane)

Description

This function eliminates the discretionary LC desires for a subject driver if the target lane is outside the feasible lane for exiting.

Inputs Arguments

fromlane: current lane.

tolane: target lane.

Output Arguments

None.

Sub-functions

None.

GenerateHeadway4Type

Syntax

double myVehicleDef::GenerateHeadway4Type(int vehTypeId)

Description

This function creates desired time headway for a subject vehicle based on its type.

Inputs Arguments

None.

Output Arguments

Desired time headway of the subject vehicle.

Sub-functions

None.

getAccExp

Syntax

double myVehicleDef::getAccExp()

Description

This function gets the coefficient used in the IDM's free flow component.

Inputs Arguments

None.

Output Arguments

Coefficient of the IDM.

Sub-functions

None.

getComfDecDLC

Syntax

double myVehicleDef::getComfDecDLC()

Description

This function gets the comfortable deceleration rate used by a subject vehicle in DLC.

Inputs Arguments

None.

Output Arguments

Deceleration rate.

Sub-functions

None.

getComfDecRampLC

Syntax

double myVehicleDef::getComfDecRampLC()

Description

This function gets the comfortable deceleration rate on ramps for a subject vehicle.

Inputs Arguments

None.

Output Arguments

Deceleration rate.

Sub-functions

None.

getDesireHeadway

Syntax

double myVehicleDef::getDesireHeadway()

Description

This function obtains desired time headway for a subject vehicle based on its type.

Inputs Arguments

None.

Output Arguments

Desired time headway of the subject vehicle.

Sub-functions

None.

getEarlyLaneKeepDis

Syntax

double myVehicleDef::getEarlyLaneKeepDis()

Description

This function gets the user-specified early lane keep distance. A driver who wants to exit the freeway is encouraged to make discretionary lane change toward the off-ramp within the distance.

Inputs Arguments

None.

Output Arguments

The distance from the off-ramp.

Sub-functions

None.

getFreeFlowSpeed

Syntax

double myVehicleDef::getFreeFlowSpeed()

Description

This function returns the free flow speed for a subject driver.

Inputs Arguments

None.

Output Arguments

Free flow speed for the subject driver.

Sub-functions

None.

getFrictionCoef

Syntax

double myVehicleDef::getFrictionCoef()

Description

This function obtains the friction coefficient for reducing the speed difference among lanes.

Inputs Arguments

None.

Output Arguments

Friction coefficient.

Sub-functions

None.

getGapHeadwayLeader

Syntax

void myVehicleDef::getGapHeadwayLeader(double& gap, double& headway, double& l_leader, double& ref_pos_front)

Description

This function obtains the spacing gap, spacing headway, and length and position of the front vehicle.

Inputs Arguments

The inputs are passed by reference, which means they will store the output values.

Output Arguments

gap: spacing gap.

headway: spacing headway.

l_leader: length of the front vehicle.

ref_pos_front: position of the front vehicle.

Sub-functions

None.

getHOVIncluded

Syntax

bool myVehicleDef::getHOVIncluded()

Description

This function determines whether an HOV lane is included in the simulation.

Inputs Arguments

None.

Output Arguments

True: HOV lane is included.

False: HOV lane is not included.

Sub-functions

None.

getLastAdaptiveMode

Syntax

int myVehicleDef::getLastAdaptiveMode()

Description

This function gets the mode (i.e., ACC/CACC or manual) for ACC/CACC vehicles at the last update interval.

Inputs Arguments

None.

Output Arguments

Mode for the subject vehicle.

Sub-functions

None.

getLastLCType

Syntax

int myVehicleDef::getLastLCType()

Description

This function gets the previous LC type for a subject vehicle.

Inputs Arguments

None.

Output Arguments

Type value.

Sub-functions

None.

getLengthCACC

Syntax

double myVehicleDef::getLengthCACC()

Description

This function obtains the anticipated length of a CACC vehicle. The length is assumed to be 1.4 times larger than its real length. This function is called by a subject CACC vehicle to estimate the length of the front vehicle when the front vehicle is also equipped with CACC. The adjusted length must be fed to the subject CACC vehicle. Otherwise, it will stop for unrealistically long terms in the stop-and-go traffic.

Inputs Arguments

None.

Output Arguments

Length of the subject vehicle.

Sub-functions

None.

getMAXacc

Syntax

double myVehicleDef::getMAXacc()

Description

This function returns the maximum acceleration capability of a subject vehicle.

Inputs Arguments

None.

Output Arguments

The maximum acceleration.

Sub-functions

None.

getMAXdec

Syntax

double myVehicleDef::getMAXdec()

Description

This function returns the maximum deceleration capability of a subject vehicle.

Inputs Arguments

None.

Output Arguments

The maximum deceleration.

Sub-functions

None.

getMaxDecInSync

Syntax

double myVehicleDef::getMaxDecInSync()

Description

This function gets the maximum deceleration that a subject driver is willing to apply during speed synchronization CF.

Inputs Arguments

None.

Output Arguments

Maximum deceleration.

Sub-functions

None.

getMinTimeBtwLcs4DLC

Syntax

double myVehicleDef::getMinTimeBtwLcs4DLC()

Description

This function gets the minimum time a subject driver should wait between two discretionary lane changes.

Inputs Arguments

None.

Output Arguments

Minimum time between two discretionary lane changes.

Sub-functions

None.

getNextSectionRampType

Syntax

int myVehicleDef::getNextSectionRampType (int& next_sec_center_lanes)

Description

This function determines if there is a ramp in the next section of a subject vehicle's route.

Inputs Arguments

next_sec_center_lanes: to store the number of freeway center lanes (e.g., lanes for freeway mainline traffic, not for on/off-ramps) of the next section.

Output Arguments

0: no ramp.

1: with on-ramp.

2: with off-ramp.

Sub-functions

None.

GetOnAccLaneFlow

Syntax

int myVehicleDef::GetOnAccLaneFlow(int next_sec)

Description

This function obtains the number of vehicles on a user-specified acceleration lane.

Inputs Arguments

next_sec: ID of the section that contains the acceleration lane.

Output Arguments

Number of vehicles in the concerned acceleration lane.

Sub-functions

None.

getOnRampVehCount

Syntax

int myVehicleDef:: getOnRampVehCount(int next_sec, double *ramp_length)

Description

This function obtains the number of vehicles on a user-specified on-ramp.

Inputs Arguments

next_sec: ID of the section downstream of the on-ramp. It is used to determine the target on-ramp.

ramp_length: pointer to the length of the on-ramp. It does not affect the output of the function.

Output Arguments

Number of vehicles in the concerned on-ramp.

Sub-functions

None.

GetPastPos

Syntax

double myVehicleDef::GetPastPos(double reaction_time)

Description

This function gets the position of a subject vehicle at an earlier time. The time is equal to the subject driver's reaction time.

Inputs Arguments

reaction_time: reaction time of the subject driver.

Output Arguments

None.

Sub-functions

None.

getPastPositionReferenceVehs

Syntax

double myVehicleDef::getPastPositionReferenceVehs (double reaction_time_ref, myVehicleDef* ref_veh, double reaction_time_this)

double myVehicleDef::getPositionReferenceVeh (myVehicleDef* ref_veh)

Description

This function gets the distance between a subject vehicle and a reference vehicle. The position of the subject vehicle is measured at reaction_time_this seconds earlier, whereas position of the reference vehicle is measured at reaction_time_ref seconds earlier. If the reaction_time_this and reaction_time_ref are not specified, the function returns the current distance between the two vehicles.

Inputs Arguments

reaction_time_ref: the length of time to look back for the position of the reference vehicle.

reaction_time_this: the length of time to look back for the position of the subject vehicle.

ref_veh: pointer to the reference vehicle.

Output Arguments

The distance between a subject vehicle and a reference vehicle.

Sub-functions

None.

GetPositionRelativeFake

Syntax

double myVehicleDef::GetPositionRelativeFake(myVehicleDef* fake_veh, double reaction_time_fake, bool downstream)

Description

This function gets the position of a user-specified fictitious vehicle.

Inputs Arguments

fake_veh: pointer to the fictitious vehicle.

reaction_time_fake: the length of time to look back. The function can return the position at an earlier time.

downstream: a flagger indicating if the fictitious vehicle is downstream from the subject vehicle or not.

Output Arguments

The position of the fictitious vehicle.

Sub-functions

getPositionReferenceVeh ()

getPoliteness

Syntax

double myVehicleDef::getPoliteness()

Description

This function obtains the politeness parameter for a subject vehicle. The parameter is used to determine if the vehicle is willing to yield to a lane changer.

Inputs Arguments

None.

Output Arguments

Politeness parameter.

Sub-functions

None.

getPosition

Syntax

double myVehicleDef::getPosition(int state)

double myVehicleDef::getPosition()

Description

This function returns the position of a subject vehicle at a given update interval. In the absence of an input argument, the function returns the current position of the subject vehicle.

Inputs Arguments

state: number of update intervals. If state = n, the function will give the position of the vehicle n update intervals earlier.

Output Arguments

The position of the subject vehicle.

Sub-functions

None.

getRampDecision

Syntax

int myVehicleDef::getRampDecision()

Description

This function gets the LC and CF maneuvers for a subject vehicle that travels on a ramp.

Inputs Arguments

None.

Output Arguments

An integer indicating the LC and CF maneuvers.

Sub-functions

None.

getRampLCSlowDownDesire

Syntax

double myVehicleDef::getRampLCSlowDownDesire()

Description

This function gets the LC desire for a vehicle deciding to slow down in an on-ramp.

Inputs Arguments

None.

Output Arguments

LC desire.

Sub-functions

None.

GetRampType

Syntax

int myVehicleDef::GetRampType(int sec_id)

Description

This function gets type of the ramp (i.e., on-ramp or off-ramp) for a given section.

Inputs Arguments

sec_id: ID of the target section.

Output Arguments

An integer indicating the ramp type.

Sub-functions

None.

GetSectionHOVLane

Syntax

int myVehicleDef::GetSectionHOVLane()

Description

This function gets the lane ID of HOV lane in the current section.

Inputs Arguments

None.

Output Arguments

Lane ID of the HOV lane.

Sub-functions

None.

GetSectionOfframpLanes

Syntax

int myVehicleDef::GetSectionOfframpLanes(int sec_id)

Description

This function gets the number of off-ramp lanes on a section.

Inputs Arguments

sec_id: ID of the target section.

Output Arguments

Number of off-ramp lanes.

Sub-functions

None.

getSectionSyncCoef

Syntax

double myVehicleDef::getSectionSyncCoef()

Description

This function gets speed synchronization coefficient for a section.

Inputs Arguments

None.

Output Arguments

Speed synchronization coefficient.

Sub-functions

None.

getSpeed

Syntax

double myVehicleDef::getSpeed()

double myVehicleDef::getSpeed(int state)

Description

This function returns the speed of a subject vehicle. In the absence of an input argument, the function returns the current speed of the subject vehicle.

Inputs Arguments

State:

- 0: before update.
- 1: after update.

Output Arguments

Speed of the subject vehicle.

Sub-functions

None.

InitialCACCACCMode

Syntax

void myVehicleDef::InitialCACCACCMode()

Description

This function sets ACC/CACC mode for ACC/CACC vehicles when they first enter the network.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

isAfterLaneChangeFinish

Syntax

bool myVehicleDef::isAfterLaneChangeFinish()

Description

This function checks if the ACF mode concludes by comparing the time elapsed since the last LC and a threshold time.

Inputs Arguments

None.

Output Arguments

True: the ACF concludes.

False: the ACF mode does not conclude.

IsCoopEffectMuch

Syntax

bool myVehicleDef::IsCoopEffectMuch(myVehicleDef *coop_veh)

Description

This function determines if a subject vehicle is too close to a vehicle intending to make a lane change. If the subject vehicle is too close (i.e., spacing headway between the two vehicles < length of the lane changer), the subject vehicle does not yield to the lane changer.

Inputs Arguments

coop_veh: pointer to the lane changer.

Output Arguments

True: the subject vehicle does not yield.

False: the subject vehicle yields.

Sub-functions

None.

isHOVActive

Syntax

bool myVehicleDef::isHOVActive()

Description

This function determines if the HOV lane is active in the simulation.

Inputs Arguments

None.

Output Arguments

True: HOV lane is active.

False: HOV lane is not active.

Sub-functions

None.

IsSectionSource

Syntax

bool myVehicleDef::IsSectionSource(int sec_id)

Description

This function determines if a target section is a source section.

Inputs Arguments

sec_id: ID of the target section.

Output Arguments

True: source section.

False: non-source section.

Sub-functions

None.

getLaneChangeDesire

Syntax

double myVehicleDef::getLaneChangeDesire()

Description

This function obtains the LC desire for a subject driver.

Inputs Arguments

None.

Output Arguments

LC desire (between 0 and 1).

Sub-functions

None.

MissTurns

Syntax

bool myVehicleDef::MissTurns()

Description

This function determines if the current destination of a subject vehicle matches the original destination.

Inputs Arguments

None.

Output Arguments

True: same destination.

False: different destination.

Sub-functions

None.

NextSecContainMerge

Syntax

bool myVehicleDef::NextSecContainMerge()

Description

This function determines if the next section contains an on-ramp.

Inputs Arguments

None.

Output Arguments

True: next section contains an on-ramp.

False: next section does not contain an on-ramp.

Sub-functions

None.

OnRampAddCoef

Syntax

double myVehicleDef::OnRampAddCoef(int num_lane_2_rightmost)

Description

This function computes a coefficient to adjust the estimated speed of a lane if the lane is adjacent to on-ramp(s).

Inputs Arguments

num_lane_2_rightmost: number of lanes between the target lane and the rightmost lane.

Output Arguments

Computed coefficent.

Sub-functions

None.

posEndAccLane

Syntax

double myVehicleDef::posEndAccLane()

Description

This function gets the length of the on-ramp acceleration lane.

Inputs Arguments

None.

Output Arguments

Length of the acceleration lane.

Sub-functions

None.

PreventSimultaneousLC

Syntax

bool myVehicleDef::PreventSimultaneousLC()

Description

This function determines if the simulation algorithm should prevent a subject vehicle and a lead vehicle from performing lane change at the same time.

Inputs Arguments

None.

Output Arguments

True: need to prevent simultaneous lane change.

False: no need to prevent.

Sub-functions

None.

RecordACC2Manual

Syntax

void myVehicleDef::RecordACC2Manual()

Description

This function records the events that an ACC vehicle switches to manual control from automated control.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

RecordMissTurnLog

Syntax

void myVehicleDef::RecordMissTurnLog()

Description

This function records the information for vehicles that miss their turns.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

RecordCrashInformation

Syntax

void myVehicleDef::RecordCrashInformation()

Description

This function records the crash information for a subject vehicle.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

ResetDesires

Syntax

void myVehicleDef::ResetDesires()

Description

This function sets both discretionary and mandatory LC desires to zero.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

ResetRelax

Syntax

void myVehicleDef::ResetRelax()

Description

This function sets behavior parameters for a driver that just completes a lane change.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

ResumeAutomatic

Syntax

bool myVehicleDef::ResumeAutomatic()

Description

This function checks if a manually-driven ACC/CACC vehicle can resume automatic driving. The subject vehicle can resume automatic driving if:

1. The subject vehicle's speed is smaller than a safe speed.
2. The subject vehicle's speed is larger than a minimum speed threshold.
3. The subject vehicle is slower than the front vehicle.

Inputs Arguments

None.

Output Arguments

True: the subject vehicle can resume automatic driving.

False: the subject vehicle cannot resume automatic driving.

Sub-functions

getSpeed (); Safe_Speed ()

ResumeManual

Syntax

bool myVehicleDef::ResumeManual()

Description

This function checks if an ACC/CACC vehicle needs to switch to manual driving from automatic driving. The subject vehicle needs to switch to manual driving if its speed is larger than a safe speed.

Inputs Arguments

None.

Output Arguments

True: the subject vehicle needs to switch to the manual driving.

False: the subject vehicle can continue driving automatically.

Sub-functions

Safe_Speed ()

Return2Manual

Syntax

void myVehicleDef::Return2Manual()

Description

This function checks if an ACC/CACC vehicle switches to manual control from automated control.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

Safe_Speed

Syntax

double myVehicleDef::Safe_Speed()

Description

This function computes the maximum speed that a subject vehicle can travel without colliding with the front vehicle.

Inputs Arguments

None.

Output Arguments

The safe speed.

Sub-functions

getSpeed (); getMAXdec(); getPosition(); getLength()

setAccExp

Syntax

void myVehicleDef::setAccExp(double param1)

Description

This function sets the coefficient used in the IDM's free flow component.

Inputs Arguments

param1: value to be set.

Output Arguments

None.

Sub-functions

None.

setComfDecDLC

Syntax

void myVehicleDef::setComfDecDLC(double param)

Description

This function sets the comfortable deceleration rate used by a subject vehicle in DLC.

Inputs Arguments

param: value to be set.

Output Arguments

None.

Sub-functions

None.

setComfDecRampLC

Syntax

void myVehicleDef::setComfDecRampLC(double param)

Description

This function sets the comfortable deceleration rate on ramps for a subject vehicle.

Inputs Arguments

param: value to be set.

Output Arguments

None.

Sub-functions

None.

setExtraDesire4FeasibleGapOfframp

Syntax

void myVehicleDef::setExtraDesire4FeasibleGapOfframp(int targetlane)

Description

This function sets the LC desire to 1 if a subject driver wants to make a lane change towards the off-ramp and the lane change is feasible.

Inputs Arguments

targetlane: target lane of the lane change.

Output Arguments

None.

Sub-functions

GapAcceptDecision_Sync_First ()

SetExtraHeadways

Syntax

void myVehicleDef::SetExtraHeadways()

Description

This function sets extra spacing headway for ACC/CACC vehicles when they first arrive at the network.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

setFrictionCoef

Syntax

void myVehicleDef::setFrictionCoef(double val)

Description

This function sets up the friction coefficient for a subject vehicle.

Inputs Arguments

val: value to be set.

Output Arguments

None.

Sub-functions

None.

SetInitialVal

Syntax

void myVehicleDef::SetInitialVal()

Description

This function sets initial parameters to a new arrival vehicle.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

setLastAdaptiveMode

Syntax

void myVehicleDef::setLastAdaptiveMode(int mode)

Description

This function sets the mode (i.e., ACC/CACC or manual) for an ACC/CACC vehicle at the last update interval.

Inputs Arguments

mode: mode to be set.

Output Arguments

None.

Sub-functions

None.

setMode

Syntax

int myVehicleDef::setMode(int avalue)

Description

This function sets up the driving mode for a subject vehicle.

Inputs Arguments

val: value to be set.

Output Arguments

None.

Sub-functions

None.

setNewPosition

Syntax

myVehicleDef::PositionSpeed myVehicleDef::setNewPosition(double pos, double velocity)

Description

This function stores historical positions of a subject vehicle in a queue. It also checks the input position and speed for the subject vehicle. It prevents the speed of the subject vehicle from being less than 0. It makes the subject vehicle stop if it is going to collide with the front vehicle after the position update.

Inputs Arguments

pos: position of the subject vehicle.

velocity: speed of the subject vehicle.

Output Arguments

A PositionSpeed struct that stores the processed position and speed of the subject vehicle.

Sub-functions

CrashAvoidancePosition ()

setLaneChangeDesire

Syntax

void myVehicleDef::setLaneChangeDesire(double incentive)

Description

This function sets up the LC desire for a subject driver.

Inputs Arguments

incentive: LC desire to be set up.

Output Arguments

None.

Sub-functions

None.

setLaneChangeDesireForce

Syntax

void myVehicleDef::setLaneChangeDesireForce(double incentive_left, double incentive_right)

Description

This function sets up the mandatory LC desire for a subject driver.

Inputs Arguments

incentive_left: LC desire to the left lane.

incentive_right: LC desire to the right lane.

Output Arguments

None.

Sub-functions

None.

setLaneChangeDesireThrd

Syntax

void myVehicleDef::setLaneChangeDesireThrd(double val)

Description

This function sets the threshold to be compared with a subject driver's LC desire.

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

None.

setLastLCType

Syntax

void myVehicleDef::setLastLCType(int type)

Description

This function sets the previous LC type for a subject vehicle.

Inputs Arguments

type: type value to be set.

Output Arguments

None.

Sub-functions

None.

setRampDecision

Syntax

void myVehicleDef::setRampDecision(int ramp_lc_decision)

Description

This function sets the LC and CF maneuvers for a subject vehicle that travels on a ramp.

Inputs Arguments

ramp_lc_decision: an integer indicating the LC and CF maneuvers.

Output Arguments

None.

Sub-functions

None.

setReactionTime

Syntax

void myVehicleDef::setReactionTime(double val)

Description

This function sets up the reaction time for a subject driver.

Inputs Arguments

val: value to be set.

Output Arguments

None.

Sub-functions

None.

setRightDLCCoeff

Syntax

void myVehicleDef::setRightDLCCoeff(double val)

Description

This function sets a coefficient to adjust a driver's DLC desire to the right lane.

Inputs Arguments

val: value to be set.

Output Arguments

None.

Sub-functions

None.

setRelaxationTime

Syntax

void myVehicleDef::setRelaxationTime(double param)

Description

This function sets the relaxation time after a lane change.

Inputs Arguments

param: value to be set.

Output Arguments

None.

Sub-functions

None.

SetRiskyRelax

Syntax

void myVehicleDef::SetRiskyRelax()

Description

This function sets behavior parameters for a driver such that he or she will take very aggressive CF and LC maneuvers.

Inputs Arguments

None.

Output Arguments

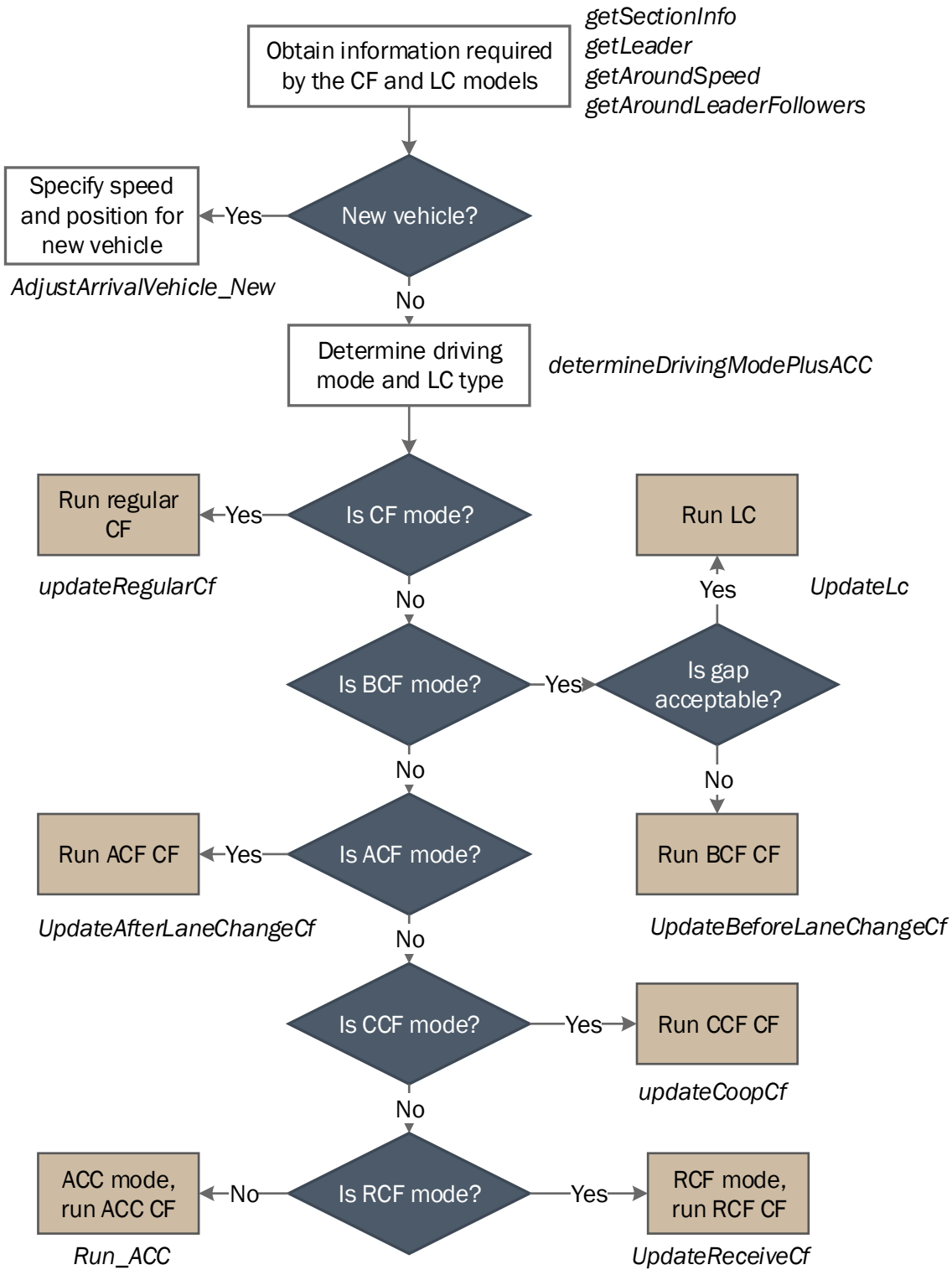
None.

APPENDIX D: FUNCTIONS FOR CONNECTED VEHICLES, AUTONOMOUS VEHICLES, AND CONNECTED AUTONOMOUS VEHICLES

This section gives a detailed account on the functions for CV, AV, and CAV modeling. In particular, the AV component includes the ACC model, the CAV component includes the CACC model, and the CV component includes the CV model with connected variable speed limit.

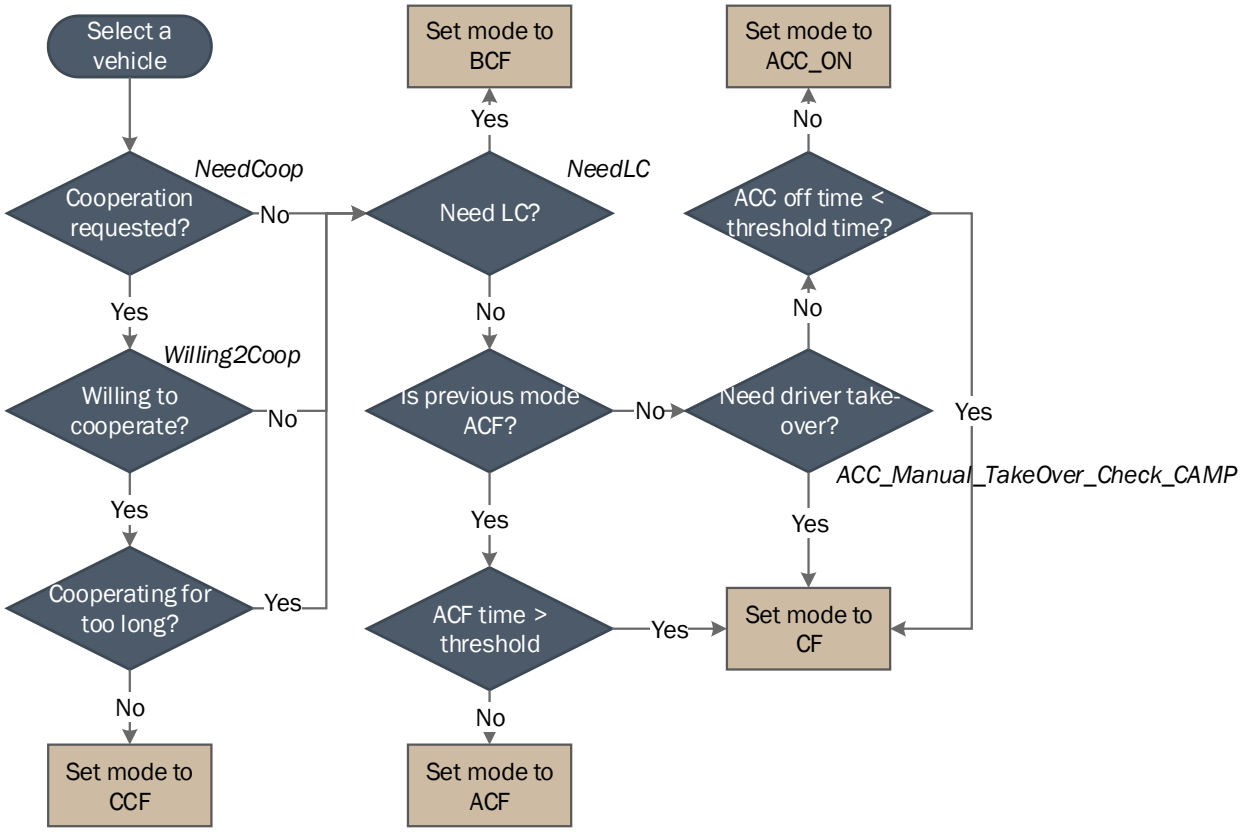
D.1 ADAPTIVE CRUISE CONTROL FUNCTIONS

Figure 39 shows the logic flow of the ACC algorithm. All processes in the figure are embedded in an umbrella function named *NGSIMPlusACC*. For an ACC vehicle, the PATH framework first determines the driving mode for the subject vehicle with the function *determineDrivingModePlusACC,* which identifies the driving mode based on the vehicle status in the current step (Figure 40). The driving mode is associated with various CF models to be implemented later to update the vehicle status. The driving mode identification process first checks if it needs to switch to the manually-driven mode. The mode switch takes place if the subject vehicle is willing to yield to a lane changer, if the vehicle attempts to make a lane change, or if there is a collision risk. Since the driver performs the yielding and lane-changing maneuvers manually, the modeling framework adopts the human driver functions *NeedCoop* and *NeedLC* (see Appendix C) for ACC (as well as CACC and CV). The collision risk is determined based on the CAMP algorithm *ACC_Manual_TakeOver_Check_CAMP,* which is described below. When a collision risk exists, the ACC and CACC driver will cut off the automated control and brake manually to avoid a crash. When there is no need for a mode switch, the ACC vehicle will adopt the ACC controller to update the vehicle speed and position. This automated longitudinal movement update uses *Run_ACC* function.



Source: FHWA

Figure 39. Diagram. Model logic flow for ACC vehicles.



Source: FHWA

Figure 40. Diagram. Car-following mode determination for ACC vehicles.

ACC_Manual_TakeOver_Check_CAMP

Syntax

bool myVehicleDef::ACC_Manual_TakeOver_Check_CAMP(double& v_des)

Description

This function checks if a CACC/ACC vehicle needs to switch to manual driving because of an imminent collision risk. The mode switch occurs if the CACC/ACC controller cannot avoid an upcoming crash, even by applying the maximum allowed deceleration. The crash scenarios are determined based on the criteria developed by the Crash Avoidance Metrics Partnership (CAMP).

Inputs Arguments

v_des: argument to store the updated speed after switching to the manual driving.

Output Arguments

True: switch needed.

False: switch not needed.

Sub-functions

CollisionScenario()

Pseudo code

If (leader_speed < 3 m/s):

- Return False, not dealing with low speed conditions.

Set POV_moving = 1 if the preceding vehicle speed > 0; otherwise POV_Moving = 0.

Compute deceleration required to avoid the collision, decREQ = 9.8 * (-0.165 + 0.685*leader acceleration + 0.080*POV_moving - 0.00894776*(current speed)^2). This is an empirical model obtained from CAMP.

If (decREQ > = 0):

- Return False, no deceleration needed for avoiding the collision.

Compute R = (current speed)^2 / (-2 * decREQ), which represents the distance traveled by the subject vehicle before stopping.

If (POV_moving > 0):

- Compute R1 = MAX(0, (current speed - leader speed)^2) / (-2 * (decREQ - leader acceleration))), which represents the distance traveled by the subject vehicle when colliding with the preceding vehicle and the preceding vehicle is still moving.
- Compute R2 = MAX(0, (current speed)^2 / (-2 * decREQ) - (leader speed)^2 / (-2 * leader_acc)), which represents the distance traveled by the subject vehicle when colliding with the preceding vehicle and the preceding vehicle stops.
- Compute collision scenario CSR if the subject vehicle and the preceding vehicle maintain the current speed and acceleration. CSR is calculated with CollisionScenario().
- If (CSR = no collision):
    - Return False.
- Else if (CSR = 1):
    - Set R = R1.
- Else if (CSR = 2):
    - Set R = R2.

If (current following distance < R):

- Return True.

Else:

- Return False.

CollisionScenario

Syntax

int myVehicleDef::CollisionScenario(double spacing, double leader_speed, double follower_speed, double leader_acc, double decREQ)

Description

This function estimates the relative speed, location, and acceleration between a subject vehicle and its preceding vehicle in next 50 timesteps (if the subject vehicle is on the source link, 200 timesteps). Based on the calculation results, the function identifies if the subject vehicle will collide with the preceding vehicle and specifies the collision scenarios.

Inputs Arguments

spacing: current car-following distance.

leader_speed: speed of the preceding vehicle.

follower_speed: speed of the subject vehicle.

leader_acc: acceleration of the preceding vehicle.

double decREQ: acceleration to be implemented by the subject vehicle.

Output Arguments

0: collision impossible.

1: collision scenario 1, collision occurs before leader stops.

2: collision scenario 2, collision occurs after leader stops.

Pseudo code

For (next 50 or 200 timestamps):

- Compute speed and location of the preceding vehicle as leader_speed_next and leaderpos, based on the current speed and acceleration of the preceding vehicle.
- Set leader_speed = leader_speed_next.
- Compute speed and location of the subject vehicle as follower_speed_next and followerpos, based on the current speed and acceleration of the subject vehicle.
- Set follower_speed = follower_speed_next.
- If (followerpos > leaderpos):
    - Collision will happen.
    - If (leader speed > 0):
        - Return 1, collision happens before leader stops.
    - Else:
        - Return 2, collision happens after leader stops.
- Else if (leader_speed > follower_speed AND leader_acc > decREQ):

    - Return 0, collision not possible if leader runs faster and accelerates faster than the follower.
- Else:
    - Return 0, collision not possible.

determineDrivingModePlusACC

Syntax

int myVehicleDef::determineDrivingModePlusACC()

Description

This function determines the driving mode of an ACC vehicle at the current update interval (see figure 4 for the logic flow).

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

DetermineLcOrMergeOrCoop()

Determine2ndLcAfterLc()

determineCoopOrLc()

DetermineReceiveOrLcOrCoop()

determineGapOrGiveup()

DetermineLcOrMergeOrCoopOrACC()

Pseudo code

If (the subject vehicle is within a node, e.g., intersection):

- If previous mode is BCF:
    - Set mode to CF.
- Else:
    - Set mode to previous mode.

Else:

- If (current mode is CF):
    - Return DetermineLcOrMergeOrCoop().
- Else if (current mode is ACF):
    - Return Determine2ndLcAfterLc().
- Else if (current mode is CCF):

    - Return determineCoopOrLc().
- Else if (current mode is RCF):
    - Return DetermineReceiveOrLcOrCoop().
- Else if (current mode is BCF):
    - Return determineGapOrGiveup().
- Else if (current mode is ACC_ON):
    - Return DetermineLcOrMergeOrCoopOrACC().

Else:

- Return current mode.

DetermineLcOrMergeOrCoopOrACC

Syntax

int myVehicleDef:: DetermineLcOrMergeOrCoopOrACC ()

Description

This function determines the driving mode of an ACC vehicle if the current mode is CF or ACC_ON.

Inputs Arguments

None.

Output Arguments

An integer representing the driving mode.

Sub-functions

NeedCoop ()

NeedLC ()

ACC_Manual_TakeOver_Check_CAMP ()

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else if (there is a collision risk determined with ACC_Manual_TakeOver_Check_CAMP):

- Set mode as CF and return.

Else:

- If (current time – last ACC off time < a threshold recovery time):

- Set mode as CF and return.

- Else:
  - Set mode as ACC_ON and return.

NGSIMPlusACC

Syntax

myVehicleDef::PositionSpeed myVehicleDef::NGSIMPlusACC(bool mode_predetermined)

Description

This is an umbrella function that updates and driving mode and determines the new lane, speed, and location for a subject ACC vehicle. It first determines the driving mode with *determineDrivingModePlusACC.* Based on the driving mode, the function calls various sub-functions to calculate the updated speed and location.

Inputs Arguments

mode_predetermined: a bool indicating if the driving mode is predetermined or not.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

determineDrivingModePlusACC ()

updateRegularCf ()

UpdateBeforeLaneChangeCf ()

UpdateAfterLaneChangeCf ()

updateCoopCf ()

UpdateReceiveCf ()

Run_ACC ()

Run_ACC

Syntax

double myVehicleDef::Run_ACC()

Description

This function runs CF model for ACC vehicles.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Pseudo code

Compute ACC gap = max(1.5 m, ACC constant time gap*current speed).

Compute reference speed:

- If (this is no preceding vehicle):
    - Set reference speed v_ref = free flow speed.
    - Set preceding vehicle's speed leader_spd = large number.
    - Set following distance from the preceding vehicle = large number.
    - Set length of preceding vehicle = 0.
- Else:
    - Set reference speed based on preceding vehicle's speed and car-following distance (a linear interpolation method is used to get the reference speed):
        - Compute a lower distance bound = max(3, ACC gap).
        - Compute a higher distance bound = max(12, 4 * ACC gap).
        - If (car-following distance <= lower bound):
            - v_ref = leader_spd.
        - Else if (car-following distance > higher bound):
            - v_ref = free flow speed.
        - Else:
            - v_ref = leader_spd + (car following distance - lower bound)*(free flow speed - leader_spd) / (higher bound - lower bound).

Compute acceleration using the ACC gap regulation model:

- des_a = 0.07*(leader_spd - current speed) + 0.23*(car-following distance - ACC gap).
- Apply the acceleration bounds des_a = max(min_acc, min(des_a, max_acc)).
- Compute updated speed v_new = current speed + des_a*timestep.

If (v_new > v_ref):

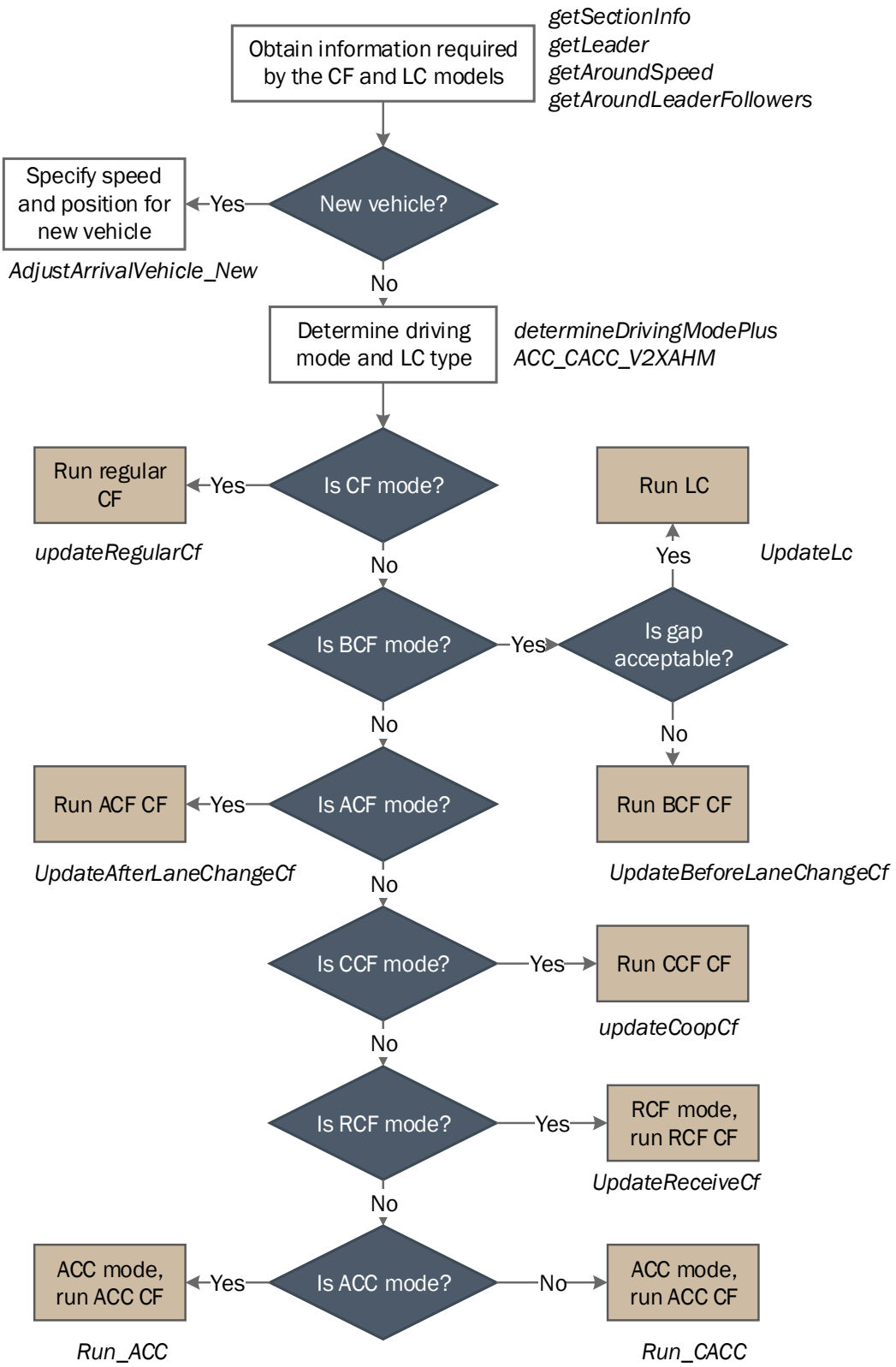
- The gap regulation model gives a too large acceleration, apply speed regulation model:
- a_new = min(a_new, 0.4*(v_ref – current speed)).
- Compute updated speed v_new = current speed + des_a*timestep.

Compute new position based on v_new and a_new.

Return the updated speed and position.

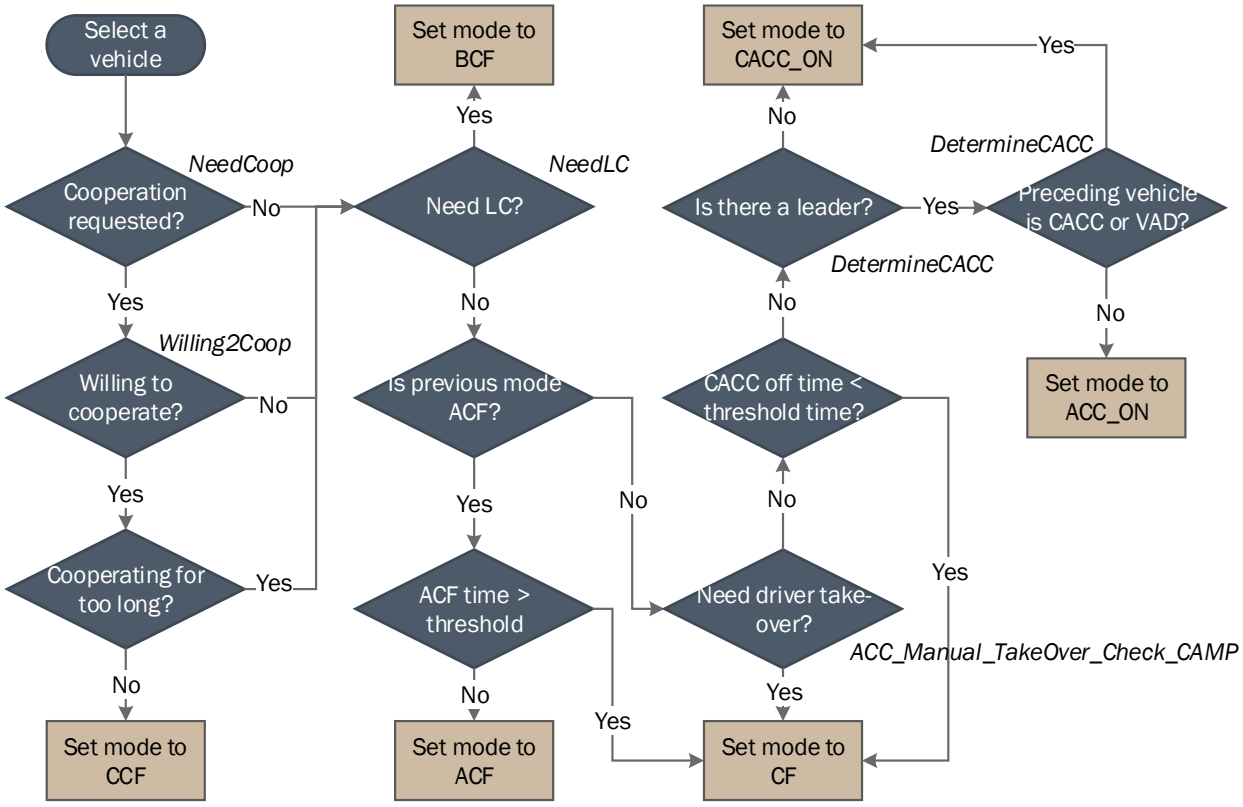
D.2 COOPERATIVE ADAPTIVE CRUISE CONTROL FUNCTIONS

Figure 41 shows the logic flow of the CACC algorithm. All processes in the figure are embedded in an umbrella function named *NGSIMPlusACC_CACC_V2VAHM*. For a CACC vehicle, the PATH framework first determines the driving mode for the subject vehicle with the function *determineDrivingModePlusACC_CACC_V2XAHM*, which identifies the driving mode based on the vehicle status in the current step (Figure 42). The driving mode is associated with the CF models to be implemented to update the vehicle status. Similar to the ACC counterpart, the function first checks if it needs to switch to the manually-driven mode. The mode switch takes place if the subject vehicle is willing to yield to a lane changer, if the vehicle attempts to make a lane change, or there is a collision risk. When there is no need for a switch to the manual mode, the CACC vehicle will adopt either the ACC controller or the CACC controller, depending on whether the preceding vehicle is a normal manually-driven vehicle or a vehicle with connectivity. In the modeling framework, both CACC vehicles and manually driven vehicles with vehicle awareness devices (VADs) are counted as vehicles with connectivity. Those vehicles can serve as CACC string leaders, allowing a following CACC vehicle to adopt the CACC mode. The longitudinal movement update uses the *Run_ACC* function if the CACC vehicle uses ACC mode, and the *Run_CACC* function if the CACC mode is applied.



Source: FHWA

Figure 41. Diagram. Model logic flow for CACC vehicles.



Source: FHWA

Figure 42. Diagram. Car-following mode determination for CACC vehicles.

CACC_CC

Syntax

double myVehicleDef::CACC_CC()

Description

This function runs CF model for a CACC vehicle when it is in the speed regulation mode. The subject vehicle could be either the string leader or the string follower.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Pseudo code

Compute reference speed:

- Set reference speed v_ref = free flow speed.
- If (preceding vehicle speed spd_pre <= v_ref):

- Set reference speed based on preceding vehicle's speed and car-following distance (a linear interpolation method is used to get the reference speed):
  - Compute a lower distance bound = max(3, intra-string gap).
  - Compute a higher distance bound = max(12, 4 * intra-string gap).
  - If (car-following distance <= lower bound):
    - v_ref = leader_spd.
  - Else if (car-following distance > higher bound):
    - v_ref = free flow speed.
  - Else:
    - v_ref = leader_spd + (car following distance - lower bound)*(free flow speed - leader_spd) / (higher bound - lower bound).

Compute acceleration using the speed regulation model:

- Compute a target acceleration a_new = min(a_new, 0.4*(v_ref – current speed)).
- Apply the acceleration bounds to the target acceleration a_new = max(min_acc, min(a_new, max_acc)).
- Compute updated speed v_new = current speed + a_new*timestep.

Return the updated speed and position.

CACC_fixed_timegap

Syntax

double myVehicleDef::CACC_fixed_timegap()

Description

This function runs CF model for a CACC vehicle when it is a string leader in the gap regulation mode.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Pseudo code

Compute CACC leader gap = inter-string gap + (intra-string gap - inter-string gap)*(relaxation time - time elapsed since last follower-leader switch) / relaxation time.

Compute reference speed:

- Set reference speed v_ref = free flow speed.
- If (preceding vehicle speed spd_pre <= v_ref):

- Set reference speed based on preceding vehicle's speed and car-following distance (a linear interpolation method is used to get the reference speed):
  - Compute a lower distance bound = max(3, intra-string gap).
  - Compute a higher distance bound = max(12, 4 * intra-string gap).
  - If (car-following distance <= lower bound):
    - v_ref = leader_spd.
  - Else if (car-following distance > higher bound):
    - v_ref = free flow speed.
  - Else:
    - v_ref = leader_spd + (car following distance - lower bound)*(free flow speed - leader_spd) / (higher bound - lower bound).

Compute acceleration using the CACC gap regulation model:

- Time gap error ek = distance to leader - CACC leader gap*current speed.
- Speed error ek_dot = preceding vehicle speed - current speed - CACC leader gap *current acceleration.
- Target speed Vk = current speed + 0.45*ek + 0.0125*ek_dot.
- If (Vk > v_ref):
  - Set Vk = v_ref.
- Target acceleration des_a = (Vk -current speed) / timestamp.
- Apply the acceleration bounds des_a = max(min_acc, min(des_a, max_acc)).
- Compute updated speed v_new = current speed + des_a*timestep.

Return the updated speed and position.

CACC_mode_switch

Syntax

double myVehicleDef::CACC_mode_switch()

Description

This function determines the mode of the CACC controller. The function considers the following conditions to maintain a stable string operation. First, it keeps the current mode when a subject vehicle is in the last section of the study network. This helps create a stable traffic flow when vehicles in a string are about to leave the network. Without this consideration, the CACC strings would switch string leaders and followers frequently, leading to large traffic disturbances. Second, it introduces a hysteresis when the subject vehicle approaches the preceding vehicle. The hysteresis control uses a lower distance bound and a higher distance bound. When the car following distance is larger than the higher bound, the controller adopts the speed regulation mode to perform a free flow travel. When the car following distance is lower than the lower bound, the controller uses the gap regulation mode to achieve a safe car-following. In addition,

the CACC controller will apply the previous driving mode when the car following distance fluctuates between the lower and higher bound. This treatment can avoid frequent mode switches between the speed and gap regulation mode when the vehicle gradually approaches the preceding vehicle. Third, this function considers the case where a VAD vehicle is the string leader. It then assigns the string follower mode to the subject vehicle, instead of ACC mode. Finally, the function also considers the string length limit. It will not assign the string follower mode to the subject vehicle if the string length reaches to the limit. The subject will become a string leader when the preceding string does not have available room.

Inputs Arguments

None.

Output Arguments

An integer indicating the driving mode of the subject vehicle. There are three possible modes: string leader speed regulation, string leader gap regulation, and string follower mode. The string follower also adopts the speed regulation and gap regulation mode. It only applies the speed regulation when there is no preceding vehicle. Since it is easy to check the existence of the preceding vehicle when updating the CF status, this function does not distinct follower speed regulation and follower gap regulation mode for simplifying the output.

Pseudo code

If (there is no leader):

- If (current section is the last section):
    - If (subject vehicle is currently a CACC string leader):
        - Return string leader speed regulation mode.
    - Else:
        - Return current mode, keeping the current mode until the subject vehicle leaves the network.
- Else:
    - Return string leader speed regulation mode.

Else:

- If (distance to preceding vehicle > an upper bound of the detection range):
    - Return string leader speed regulation mode.
- Else if (distance to preceding vehicle > an lower bound of the detection range):
    - If (subject vehicle is currently a CACC string leader in speed regulation mode):
        - Return string leader speed regulation mode.
    - Else if (subject vehicle is currently a CACC string leader in gap regulation mode):
        - Return string leader gap regulation mode.
    - Else:

        - Return string follower mode.
- Else:
    - If (subject vehicle currently not in CACC string):
        - If (preceding vehicle is a VAD vehicle):
            - Return string follower mode.
        - Else:
            - Return string leader speed regulation mode.
- Else:
    - If (the preceding vehicle in CACC string AND string length > length limit):
        - Return string leader gap regulation mode.
    - Else:
        - If (preceding vehicle is a VAD vehicle):
            - Return string follower mode.
        - Else:
            - Return string leader speed regulation mode.

CACC_veh_model

Syntax

double myVehicleDef::CACC_veh_model()

Description

This function runs CF model for a CACC vehicle when it is a string follower in the gap regulation mode.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Pseudo code

Compute CACC follower gap = intra-string gap + (inter-string gap - intra-string gap) * (relaxation time - time elapsed since last leader-follower switch) / relaxation time.

Compute reference speed:

- Set reference speed v_ref = free flow speed.
- If (subject vehicle switches from leader to follower and time elapsed since last switch < relaxation time):

    - Set v_ref = preceding vehicle speed + 2, preventing the subject vehicle from braking aggressively.
- If (preceding vehicle speed spd_pre <= v_ref):
    - Set reference speed based on preceding vehicle's speed and car-following distance (a linear interpolation method is used to get the reference speed):
        - Compute a lower distance bound = max(3, intra-string gap) .
        - Compute a higher distance bound = max(12, 4 * intra-string gap).
        - If (car-following distance <= lower bound):
            - v_ref = leader_spd.
        - Else if (car-following distance > higher bound):
            - v_ref = free flow speed.
        - Else:
            - v_ref = leader_spd + (car following distance - lower bound)*(free flow speed - leader_spd) / (higher bound - lower bound).

Compute acceleration using the CACC gap regulation model:

- Time gap error ek = distance to leader - CACC follower gap*current speed.
- Speed error ek_dot = preceding vehicle speed - current speed - CACC leader gap *current acceleration.
- Target speed Vk = current speed + 0.45*ek + 0.0125*ek_dot.
- If (Vk > v_ref):
    - Set Vk = v_ref.
- Target acceleration des_a = (Vk - current speed) / timestamp.
- Apply the acceleration bounds des_a = max(min_acc, min(des_a, max_acc)).
- Compute updated speed v_new = current speed + des_a*timestep.

Return the updated speed and position.

determineDrivingModePlusACC_CACC_V2XAHM

Syntax

int myVehicleDef::determineDrivingModePlusACC()

Description

This function determines the driving mode of a CACC vehicle at the current update interval (see figure D4 for the logic flow).

Inputs Arguments

None.

Output Arguments

None.

Sub-functions

DetermineLcOrMergeOrCoopOrACC0rCACC_0729 ()

DetermineLcOrMergeOrCoop()

Determine2ndLcAfterLc()

determineCoopOrLc()

DetermineReceiveOrLcOrCoop()

determineGapOrGiveup()

Pseudo code

If (the subject vehicle is within a node, e.g., intersection):

- If previous mode is BCF:
    - Set mode to CF.
- Else:
    - Set mode to previous mode.

Else:

- If (current mode is CF):
    - Return DetermineLcOrMergeOrCoopOrACC0rCACC_0729 ().
- Else if (current mode is ACF):
    - Return Determine2ndLcAfterLc().
- Else if (current mode is CCF):
    - Return determineCoopOrLc().
- Else if (current mode is RCF):
    - Return DetermineReceiveOrLcOrCoop().
- Else if (current mode is BCF):
    - Return determineGapOrGiveup().
- Else if (current mode is ACC_ON):
    - Return DetermineLcOrMergeOrCoopOrACC0rCACC_0729 ().
- Else if (current mode is CACC_ON):
    - Return DetermineLcOrMergeOrCoopOrACC0rCACC_0729 ().
- Else:
    - Return current mode.

DetermineLcOrMergeOrCoopOrACC0rCACC_0729

Syntax

int myVehicleDef:: DetermineLcOrMergeOrCoopOrACC0rCACC_0729 ()

Description

This function determines the driving mode of a CACC vehicle if the current mode is CF, ACC_ON or CACC_ON.

Inputs Arguments

None.

Output Arguments

An integer representing the driving mode.

Sub-functions

NeedCoop ()

NeedLC ()

ACC_Manual_TakeOver_Check_CAMP ()

Pseudo code

If (NeedCoop()):

- Set mode as CCF and return.

Else if (NeedLC()):

- Set mode as BCF and return.

Else if (there is a collision risk determined with ACC_Manual_TakeOver_Check_CAMP):

- Set mode as CF and return.

Else:

- If (current time – last CACC off time < a threshold recovery time):
    - Set mode as CF and return.
- Else:
    - If (there is no leader OR the preceding vehicle is a CACC vehicle or VAD vehicle):
        - Set mode ad CACC_ON and return.
    - Else:
        - Set mode as ACC_ON and return.

NGSIMPlusACC_CACC_V2VAHM

Syntax

myVehicleDef::PositionSpeed myVehicleDef:: NGSIMPlusACC_CACC_V2VAHM (bool mode_predetermined)

Description

This is an umbrella function that updates the driving mode and determines the new lane, speed, and location for a subject CACC vehicle. It first determines the driving mode with *determineDrivingModePlusACC_CACC_V2XAHM*. Based on the driving mode, the function calls various sub-functions to calculate the updated speed and location.

Inputs Arguments

mode_predetermined: a bool indicating if the driving mode is predetermined or not.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

determineDrivingModePlusACC_CACC_V2XAHM ()

updateRegularCf ()

UpdateBeforeLaneChangeCf ()

UpdateAfterLaneChangeCf ()

updateCoopCf ()

UpdateReceiveCf ()

Run_ACC ()

Run_CACC ()

Run_CACC

Syntax

double myVehicleDef::Run_CACC()

Description

This function runs CF model for ACC vehicles.

Inputs Arguments

None.

Output Arguments

A PositionSpeed struct that stores the new position and speed of the subject vehicle.

Sub-functions

CACC_mode_switch()

CACC_fixed_timegap()

CACC_CC()

CACC_veh_model ()

Pseudo code

Compute CACC controller mode with CACC_mode_switch(). The CACC controller could adopt three modes: string leader speed regulation, string leader gap regulation, and string follower mode.

If (subject vehicle switches from string leader to string follower, or vice versa):

- Record mode switch time and switch type (leader to follower or follower to leader).

If (string leader gap regulation):

- Update new speed and location with CACC_fixed_timegap().

Else if (string leader speed regulation):

- Update new speed and location with CACC_CC().

Else if (there is no preceding vehicle):

- Update new speed and location with CACC_CC(), which represents CACC string follower in the speed regulation mode.

Else:

- Update new speed and location with CACC_veh_model (), which is CACC string follower gap regulation mode.

D.3 CONNECTED MANUALLY DRIVEN VEHICLE FUNCTIONS

The car-following and lane-changing logics for CMDV are the same as the HV algorithms described in Appendix C. The difference between a CMDV and an HV is that the CMDV can receive real-time VSL/VSA from the TMC. Once receiving the VSL/VSA, the CMDV driver changes the desired speed based on the VSL/VSA level and the compliance level. The modeling framework implements the modeling procedure in Aimsun API instead of that in Aimsun MicroSDK to reduce the computation load. If the procedure were executed in MicroSDK, the modeling framework would call the VSL/VSA functions repeatedly for all vehicles in the network. Since many of the calculations in these functions are identical for all vehicles, such an implementation scheme could induce much unnecessary computation demand. On the other hand, the Aimsun API offers an *AAPIPostMagage* function that is automatically executed at the end of each simulation interval. As the VSL/VSA functions are embedded in *AAPIPostMagage*, the modeling framework only needs to run the VSL/VSA algorithm once in every simulation interval to generate updated desired speed levels for all CMDVs. This creates an efficient model implementation that is important for evaluating large road networks. The CMDV algorithm integrates the following functions into *AAPIPostMagage*:

- CollectSpeedAdvisoryData(), which collects section-based traffic data for calculating the VSL/VSA.

- DetermineAdvisorySpeed(), which computes the VSL/VSA for vehicles in different road sections.
- SetAdvisorySpeed(), which sets the desired speed of each CV based on its compliance and the real-time VSL/VSA level.

CollectSpeedAdvisoryData

Syntax

void CollectSpeedAdvisoryData ()

Description

This function collects speed data from individual CMDVs in the study network. The collected speed data points are mapped to each data aggregation segment. Those speed data points are the basis for computing the average speed of each data aggregation segment in each speed limit update interval.

Inputs Arguments

None.

Output Arguments

None.

The average speed of each data aggregation segment is stored internally in the Aimsun API.

Pseudo code

For (each link in the network):

- For (each vehicle in a link):
    - Get vehicle speed and acceleration.
    - Get ID of the data aggregation segment that is associated with the vehicle.
    - If (vehicle is connected):
        - Compute vehicle miles traveled (VMT) and vehicle hours traveled (VHT) of the vehicle in the current simulation step based on the speed and acceleration.
        - Store VMT and VHT to the associated data aggregation segment.

Compute average speed of each data aggregation segment: average speed = total VMT / total VHT.

DetermineAdvisorySpeed

Syntax

void DetermineAdvisorySpeed ()

Description

This function determines the VSL/VSA for each data aggregation segment based on the average speed data collected via *CollectSpeedAdvisoryData()*. The user will need to import a list of link

IDs before calling the function. The link ID list records the road links that might become bottlenecks during peak hours. Those links usually locate near the on-ramp and off-ramp areas.

Inputs Arguments

None.

Output Arguments

None.

The VSL/VSA of each data aggregation segment is store internally in the Aimsun API.

Pseudo code

For (each link in the potential bottleneck links):

- Get link average speed.
- If (link average speed < speed threshold):
    - The link is an active bottleneck.
    - Set VSL/VSA of the link to max VSL/VSA.
    - For (each link upstream from the current link):
        - If (VSA/VSL of the target link <= 0):
            - Set VSA/VSL = min(max VSL/VSA, min(min VSL/VSA, bottleneck speed * step speed + coefficient alpha step speed)).
        - Else:
            - The VSL/VSA of the target link has been set by a previous iteration, break the loop.
- For (each link downstream from the current link):
    - If (VSA/VSL of the target link <= 0):
        - Set VSA/VSL = max VSL/VSA, links downstream from the bottleneck get high advisory speed.
    - Else:
        - The VSL/VSA of the target link has been set by a previous iteration, break the loop.

SetAdvisorySpeed

Syntax

void SetAdvisorySpeed ()

Description

This function sets the desired speed of a CV based on the VSL/VSA of the current link and the driver's compliance level.

Inputs Arguments

None.

Output Arguments

None.

The VSL/VSA of each data aggregation segment is store internally in the Aimsun API.

Pseudo code

For (each link in the network)

- For (each vehicle in a link):
    - Get VSL/VSA of the current link.
    - If (vehicle is connected):
        - Get compliance level of the vehicle.
        - Set desired speed = max(min VSL/VSA, VSL/VSA + random compliance term + random speed deviation term).
        - If (desired speed - original desired speed > original desired speed * acceleration coefficient):
            - Set desired speed = original desired speed * (1 + acceleration coefficient).
        - Else if (desired speed - original desired speed < -original desired speed * deceleration coefficient):
            - Set desired speed = original desired speed - original desired speed * deceleration coefficient.