

**Task Order: 22**

**March 22, 2020**

**Developing Analysis, Modeling, and Simulation (AMS) Tools for Connected and Automated Vehicle (CAV) Applications**

**Algorithm Description Document: An Improve Cooperative Adaptive Cruise Control Model**

SI Conversion Chart. See https://www.fhwa.dot.gov/publications/convtabl.cfm for html version.

Table of Contents

[Executive Summary 1](#_Toc35777826)

[Chapter 1. PURPOSE OF THIS MODEL 3](#_Toc35777827)

[Purpose of this Document 3](#_Toc35777828)

[Purpose of this Model 3](#_Toc35777829)

[Document Overview 5](#_Toc35777830)

[Chapter 2. MODEL DEVELOPMENT AND LOGIC 7](#_Toc35777831)

[Descriptions of Model Logic 7](#_Toc35777832)

[Model Development 7](#_Toc35777833)

[Speed Adaptation of Connected Drivers 7](#_Toc35777834)

[Traffic Data for Connected Speed Management 9](#_Toc35777835)

[Variable Advisory Speed Computation for Isolated Bottlenecks 11](#_Toc35777836)

[Implementation of the Connected Speed Management in Freeway Corridors 12](#_Toc35777837)

[Chapter 3. MODEL CALIBRATION AND VALIATION 15](#_Toc35777838)

[Assumptions in Calibration and Validation 15](#_Toc35777839)

[Calibration and Validation Datasets 15](#_Toc35777840)

[Calibration and Validation Methodology 17](#_Toc35777841)

[Step 1: Data Preparation 17](#_Toc35777842)

[Step 2: Distribution Calibration 17](#_Toc35777843)

[Step 3: Distribution Validation 18](#_Toc35777844)

[Calibration and Validation Results 19](#_Toc35777845)

[Chapter 4. BASIC GUIDANCE ONE MODEL IMPLEMENTATION 23](#_Toc35777846)

[modeling Process 23](#_Toc35777847)

[Stage 1: Information Gathering 23](#_Toc35777848)

[Stage 2: New System Status Computation 23](#_Toc35777849)

[Stage 3: System Status Update 23](#_Toc35777850)

[Chapter 5. SENSITIVITY STUDY 25](#_Toc35777851)

[Implementation of the developed model into a traffic simulation tool 25](#_Toc35777852)

[Design of simulation experiments 25](#_Toc35777853)

[Simulation results for the different scenarios 27](#_Toc35777854)

[Chapter 6. SUMMARY AND RECOMMENDATIONS 37](#_Toc35777855)

[Chapter 7. Acknowledgments 39](#_Toc35777856)

[References 41](#_Toc35777857)

[APPENDIX A. FUNCTIONS FOR SIMULATION PROCESS CONTROL 43](#_Toc35777858)

[arrivalNewVehicle 45](#_Toc35777859)

[evaluateCarFollowing 45](#_Toc35777860)

[evaluateLaneChanging 47](#_Toc35777861)

[mybehavioralModel 47](#_Toc35777862)

[removedVehicle 48](#_Toc35777863)

[createFreeFlowSpeed 48](#_Toc35777864)

[getAroundLeaderFollowers 48](#_Toc35777865)

[getAroundSpeed 49](#_Toc35777866)

[getLeader 49](#_Toc35777867)

[getSectionInfo 49](#_Toc35777868)

[ReadExternalParameters 50](#_Toc35777869)

[ReadHOVSetting 50](#_Toc35777870)

[SetExternalParameters 51](#_Toc35777871)

[APPENDIX B. FUNCTIONS FOR NEW VEHICLE GENERATION 53](#_Toc35777872)

[AdjustArrivalVehicle\_New 53](#_Toc35777873)

[NGSIM\_Speed 54](#_Toc35777874)

[SetPara4NewVeh 56](#_Toc35777875)

[GetEquPosition 58](#_Toc35777876)

[UpdateLatestArrival 59](#_Toc35777877)

[APPENDIX C: VEHICLE CAR-FOLLOWING AND LANE-CHANGING FUNCTIONS 61](#_Toc35777878)

[AccGapAccepted 67](#_Toc35777879)

[AnticipatedAcc 69](#_Toc35777880)

[BaseCfModel 70](#_Toc35777881)

[BeforeExitorTurningLcSlowDown 72](#_Toc35777882)

[BeforeExitorTurningLcSync 73](#_Toc35777883)

[BeforeLeftLaneChangingMove4HOV 74](#_Toc35777884)

[BeforeOffRampLcSlowDown 76](#_Toc35777885)

[BeforeOnRampLcSlowDown 77](#_Toc35777886)

[BeforeOnRampLcSync 78](#_Toc35777887)

[CalculateDesireForce 79](#_Toc35777888)

[CombineLCDesires 80](#_Toc35777889)

[DesireEquation 82](#_Toc35777890)

[Determine2ndLcAfterLc 83](#_Toc35777891)

[determineCoopOrLc 84](#_Toc35777892)

[determineDrivingMode 85](#_Toc35777893)

[determineGapOrGiveup 86](#_Toc35777894)

[DetermineLcOrMergeOrCoop 87](#_Toc35777895)

[DetermineReceiveOrLcOrCoop 88](#_Toc35777896)

[DisGapAccepted 88](#_Toc35777897)

[DLCCFDecision 90](#_Toc35777898)

[DLCDesire 90](#_Toc35777899)

[ExitCfDecision 92](#_Toc35777900)

[GapAcceptDecision\_Sync\_First 92](#_Toc35777901)

[GippsGap 94](#_Toc35777902)

[GippsDecelerationTerm 96](#_Toc35777903)

[isLaneChangingPossible 97](#_Toc35777904)

[MinCollisionAvoidancePos 98](#_Toc35777905)

[NeedCoop 99](#_Toc35777906)

[NeedDlc 100](#_Toc35777907)

[NeedLC 102](#_Toc35777908)

[NeedLc4Turning 103](#_Toc35777909)

[NeedRampLc 105](#_Toc35777910)

[PosCf 106](#_Toc35777911)

[PosCf2EndofExitTurning 106](#_Toc35777912)

[PosCf2EndofRamp 107](#_Toc35777913)

[PosCfSkipGap 107](#_Toc35777914)

[RampCfDecision 108](#_Toc35777915)

[RunNGSIM 109](#_Toc35777916)

[UpdateAfterLaneChangeCf 109](#_Toc35777917)

[UpdateBeforeLaneChangeCf 110](#_Toc35777918)

[updateCoopCf 112](#_Toc35777919)

[UpdateLc 113](#_Toc35777920)

[UpdateReceiveCf 113](#_Toc35777921)

[updateRegularCf 114](#_Toc35777922)

[Willing2Coop 114](#_Toc35777923)

[ApplyNGSIMModel 115](#_Toc35777924)

[BoostOnrampIncentive 115](#_Toc35777925)

[CrashAvoidancePosition 115](#_Toc35777926)

[distance2EndAccLane 116](#_Toc35777927)

[EliminateDlcDesireOutSideRouteLanes 116](#_Toc35777928)

[GenerateHeadway4Type 117](#_Toc35777929)

[getAccExp 117](#_Toc35777930)

[getComfDecDLC 117](#_Toc35777931)

[getComfDecRampLC 118](#_Toc35777932)

[getDesireHeadway 118](#_Toc35777933)

[getEarlyLaneKeepDis 118](#_Toc35777934)

[getFreeFlowSpeed 119](#_Toc35777935)

[getFrictionCoef 119](#_Toc35777936)

[getGapHeadwayLeader 120](#_Toc35777937)

[getHOVIncluded 120](#_Toc35777938)

[getLastAdaptiveMode 121](#_Toc35777939)

[getLastLCType 121](#_Toc35777940)

[getLengthCACC 121](#_Toc35777941)

[getMAXacc 122](#_Toc35777942)

[getMAXdec 122](#_Toc35777943)

[getMaxDecInSync 122](#_Toc35777944)

[getMinTimeBtwLcs4DLC 123](#_Toc35777945)

[getNextSectionRampType 123](#_Toc35777946)

[GetOnAccLaneFlow 124](#_Toc35777947)

[getOnRampVehCount 124](#_Toc35777948)

[GetPastPos 125](#_Toc35777949)

[getPastPositionReferenceVehs 125](#_Toc35777950)

[GetPositionRelativeFake 126](#_Toc35777951)

[getPoliteness 126](#_Toc35777952)

[getPosition 126](#_Toc35777953)

[getRampDecision 127](#_Toc35777954)

[getRampLCSlowDownDesire 127](#_Toc35777955)

[GetRampType 128](#_Toc35777956)

[GetSectionHOVLane 128](#_Toc35777957)

[GetSectionOfframpLanes 128](#_Toc35777958)

[getSectionSyncCoef 129](#_Toc35777959)

[getSpeed 129](#_Toc35777960)

[InitialCACCACCMode 130](#_Toc35777961)

[isAfterLaneChangeFinish 130](#_Toc35777962)

[IsCoopEffectMuch 130](#_Toc35777963)

[isHOVActive 131](#_Toc35777964)

[IsSectionSource 131](#_Toc35777965)

[getLaneChangeDesire 132](#_Toc35777966)

[MissTurns 132](#_Toc35777967)

[NextSecContainMerge 132](#_Toc35777968)

[OnRampAddCoef 133](#_Toc35777969)

[posEndAccLane 133](#_Toc35777970)

[PreventSimultaneousLC 134](#_Toc35777971)

[RecordACC2Manual 134](#_Toc35777972)

[RecordMissTurnLog 134](#_Toc35777973)

[RecordCrashInformation 135](#_Toc35777974)

[ResetDesires 135](#_Toc35777975)

[ResetRelax 135](#_Toc35777976)

[ResumeAutomatic 136](#_Toc35777977)

[ResumeManual 136](#_Toc35777978)

[Return2Manual 137](#_Toc35777979)

[Safe\_Speed 137](#_Toc35777980)

[setAccExp 138](#_Toc35777981)

[setComfDecDLC 138](#_Toc35777982)

[setComfDecRampLC 138](#_Toc35777983)

[setExtraDesire4FeasibleGapOfframp 139](#_Toc35777984)

[SetExtraHeadways 139](#_Toc35777985)

[setFrictionCoef 139](#_Toc35777986)

[SetInitialVal 140](#_Toc35777987)

[setLastAdaptiveMode 140](#_Toc35777988)

[setMode 141](#_Toc35777989)

[setNewPosition 141](#_Toc35777990)

[setLaneChangeDesire 141](#_Toc35777991)

[setLaneChangeDesireForce 142](#_Toc35777992)

[setLaneChangeDesireThrd 142](#_Toc35777993)

[setLastLCType 143](#_Toc35777994)

[setRampDecision 143](#_Toc35777995)

[setReactionTime 143](#_Toc35777996)

[setRightDLCCoeff 144](#_Toc35777997)

[setRelaxationTime 144](#_Toc35777998)

[SetRiskyRelax 144](#_Toc35777999)

[APPENDIX D: FUNCTIONS FOR CONNECTED VEHICLES, AUTONOMOUS VEHICLES, AND CONNECTED AUTONOMOUS VEHICLES 147](#_Toc35778000)

[ACC\_Manual\_TakeOver\_Check\_CAMP 149](#_Toc35778001)

[CollisionScenario 151](#_Toc35778002)

[determineDrivingModePlusACC 152](#_Toc35778003)

[DetermineLcOrMergeOrCoopOrACC 153](#_Toc35778004)

[NGSIMPlusACC 154](#_Toc35778005)

[Run\_ACC 155](#_Toc35778006)

[CACC\_CC 158](#_Toc35778007)

[CACC\_fixed\_timegap 159](#_Toc35778008)

[CACC\_mode\_switch 160](#_Toc35778009)

[CACC\_veh\_model 162](#_Toc35778010)

[determineDrivingModePlusACC\_CACC\_V2XAHM 163](#_Toc35778011)

[DetermineLcOrMergeOrCoopOrACC0rCACC\_0729 165](#_Toc35778012)

[NGSIMPlusACC\_CACC\_V2VAHM 166](#_Toc35778013)

[Run\_CACC 166](#_Toc35778014)

[CollectSpeedAdvisoryData 168](#_Toc35778015)

[DetermineAdvisorySpeed 169](#_Toc35778016)

[SetAdvisorySpeed 170](#_Toc35778017)

List of Figures

[**Figure 1. Diagram. Components of the PATH model framework. 4**](#_Toc35778018)

[**Figure 2. Equation. Acceleration of a connected vehicle. 8**](#_Toc35778019)

[**Figure 3. Equation. Variables used for calculating the acceleration of a connected vehicle. 8**](#_Toc35778020)

[**Figure 4. Equation. Desired speed of a connected vehicle. 9**](#_Toc35778021)

[**Figure 5. Plot. Segregation of a freeway corridor. 10**](#_Toc35778022)

[**Figure 6. Equation. Computation of the aggregated speed for a section. 10**](#_Toc35778023)

[**Figure 7. Equation. Variable advisory speed for sections upstream from the bottleneck. 11**](#_Toc35778024)

[**Figure 8. Equation. Weighted occupancy computation. 11**](#_Toc35778025)

[**Figure 9. Equation. Spatial, temporal, and maximum/minimum bounds for the advisory speed. 12**](#_Toc35778026)

[**Figure 10. Equation. Identification of bottleneck sections. 13**](#_Toc35778027)

[**Figure 11. Equation. Final advisory speed for a section. 13**](#_Toc35778028)

[**Figure 12. Plot. Overview of system scope, VSA sign and Changeable Message Sign (CMS) locations, and the construction area (Lu et al., 2019). 16**](#_Toc35778029)

[**Figure 13. Equation. Empirical cumulative distribution functions for the low and high VSA levels. 18**](#_Toc35778030)

[**Figure 14. Equation. Kolmogorov-Smirnov test statistic. 18**](#_Toc35778031)

[**Figure 15. Equation. Criterion for rejecting the null hypothesis. 18**](#_Toc35778032)

[**Figure 16. Diagram. Visualization of the Kolmogorov-Smirnov test statistic. 19**](#_Toc35778033)

[**Figure 17. Diagram. Calibrated empirical distributions of ε for the low and high VSA levels (mean and standard deviation in mph). 20**](#_Toc35778034)

[**Figure 18. Diagram. The probability density functions for the calibration and validation datasets. 21**](#_Toc35778035)

[**Figure 19. Equation. Determination of the increased compliance level. 26**](#_Toc35778036)

[**Figure 20. Diagram. Simulated freeway corridor. 27**](#_Toc35778037)

[**Figure 21. Diagram. Average vehicle speed and standard deviation of the speed under various CV market penetrations. 29**](#_Toc35778038)

[**Figure 22. Diagram. Average vehicle fuel efficiency under various CV market penetrations. 30**](#_Toc35778039)

[**Figure 23. Diagram. Fundamental diagram of 10 percent CV market penetration with the baseline compliance level. 33**](#_Toc35778040)

[**Figure 24. Diagram. Fundamental diagram of 10 percent CV market penetration with the full compliance level. 33**](#_Toc35778041)

[**Figure 25. Diagram. Fundamental diagram of 30 percent CACC vehicle market penetration without the VSA. 34**](#_Toc35778042)

[**Figure 26. Diagram. Fundamental diagram of 30 percent CACC vehicle market penetration with the VSA. 34**](#_Toc35778043)

[**Figure 27. Diagram. Logic flow of the PATH simulation algorithm. 44**](#_Toc35778044)

[**Figure 28. Diagram. Vehicles involved in the CF and LC interactions. 62**](#_Toc35778045)

[**Figure 29. Diagram. CF and LC logic for manually-driven vehicles. 63**](#_Toc35778046)

[**Figure 30. Diagram. Driving mode determination for manually-driven vehicles. 64**](#_Toc35778047)

[**Figure 31. Diagram. BCF logic flow for ALC and MLC (off-ramp exiting maneuver). 65**](#_Toc35778048)

[**Figure 32. Diagram. BCF logic flow for MLC (on-ramp merging maneuver). 66**](#_Toc35778049)

[**Figure 33. Diagram. BCF logic flow for DLC. 67**](#_Toc35778050)

[**Figure 34. Diagram. Model logic flow for ACC vehicles. 148**](#_Toc35778051)

[**Figure 35. Diagram. Car-following mode determination for ACC vehicles. 149**](#_Toc35778052)

[**Figure 36. Diagram. Model logic flow for CACC vehicles. 158**](#_Toc35778053)

[**Figure 37. Diagram. Car-following mode determination for CACC vehicles. 158**](#_Toc35778054)

List of Tables

[**Table 1. Sample sizes for the low and high VSA cases. 19**](#_Toc35778055)

[**Table 2. Results of the two-sample Kolmogorov-Smirnov test. 20**](#_Toc35778056)

[**Table 3. Simulation scenarios for CVs. 26**](#_Toc35778057)

[**Table 4. Simulation Scenarios for CACC vehicles. 26**](#_Toc35778058)

[**Table 5. Fuel model parameters. 27**](#_Toc35778059)

[**Table 6. Effects of driver compliance level on speed and vehicle fuel efficiency at 10 percent CV market penetration. 30**](#_Toc35778060)

[**Table 7. Effects of driver compliance level on speed and vehicle fuel efficiency at 20 percent CV market penetration. 30**](#_Toc35778061)

[**Table 8. Effects of driver compliance level on speed and vehicle fuel efficiency at 100 percent CV market penetration. 31**](#_Toc35778062)

[**Table 9. Comparison of the baseline compliance and the full compliance level. 32**](#_Toc35778063)

[**Table 10. Comparison of the baseline and full compliance levels. 35**](#_Toc35778064)

[**Table 11. Sub-Functions executed for creating a new vehicle. 56**](#_Toc35778065)

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 vehicles (CVs), autonomous vehicles (AVs), and connected automated vehicles (CAVs) at the same time. The interaction of various types of vehicle fleet 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 California Partners for Advanced Transportation Technology (PATH) team has developed and integrated a CV model into its existing AMS framework.

A CV 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 CV 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 CV-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 CV on the traffic flow, this study implemented the CV 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 CV is affected by the driver’s compliance with the VSA. The CV 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 CV drivers’ speed adaptations to represent real-world drivers’ speed patterns under the influence of VSA control. CV 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 CV 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 CV speed adaptation under various CV 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 significant effects on the freeway corridor when the CV market penetration is 10 percent. With the advisory speed, the variation of the speed reduced by 6 percent, and the vehicle fuel efficiency increased by 2.2 to 4.9 percent, depending on the results of different energy models. The results suggest that the speed adaptation of a few connected drivers could substantially change the traffic flow pattern, leading to a more energy efficient traffic flow. As CV market penetration further increased, reductions in speed variation and increases in fuel efficiency slowed down. When CV market penetration reached 100 percent, speed variation decreased by 9.1 percent and vehicle fuel economy increased by 2.9 to 10.1 percent, depending on the energy models used. The performance of the VSA algorithm was not sensitive to small changes of the driver compliance level. However, traffic flow patterns changed significantly when CV drivers fully complied with the VSA. Full CV driver compliance with the VSA did not result in significant benefits to speed harmonization or vehicle fuel efficiency, as the VSA algorithm in this study could not generate advisory speed based on predicted traffic conditions. A similar shortcoming was observed with the cooperative adaptive cruise control (CACC) cases. When VSA is implemented with CACC, the CACC controller would perfectly adopt the advisory speed as the set speed. Nonetheless, adding the VSA algorithm to CACC vehicles did not bring notable benefits to the freeway corridor because the VSA controller tended to underuse the bottleneck capacity due to its delayed response to the traffic variations. To address this shortcoming, a predictive VSA algorithm, or the combined application of VSA and ramp metering, is recommended for the future work.

Another important objective of this document is to help future users easily to adapt and customize the proposed CV model, as well as the PATH human driver, adaptive cruise control (ACC), and CACC models in a traffic simulation tool they preferred to meet their simulation needs. To this end, the detailed algorithm description and pseudo codes are given in the Appendices of the document..

# PURPOSE OF THIS MODEL

## Purpose of this Document

Connected and automated vehicles (CAV) technologies offer potentially transformative societal impacts, including significant mobility, safety, and environmental benefits. The United States Department of Transportation (USDOT) has led the development, research, and standards-making of these technologies and is currently developing CAV deployment and implementation approaches and guidelines.

In order for CAV applications to be deployed, State and local transportation agencies must first be able to fully and effectively quantify the impacts of such deployments and identify which application best addresses their unique transportation problem. Traffic analysis, modeling, and simulation (AMS) tools provide an efficient means to evaluate transportation improvement projects prior to deployment. Current AMS tools are not well-suited for evaluating CAV applications due to their inability to incorporate vehicle connectivity/communication and automated driving features. To mitigate this gap, the FHWA has sponsored this project to develop CAV applications/models based on field data to support the CAV simulation community. 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.

This document presents an improved CACC model for light duty CAVs, with the objective of providing detailed information on this model to improve the CAV simulation community. This document is expected to help future users easily adopt and customize the improved CACC model in their traffic simulation tool of choice to meet their simulation needs. To this end, this document provides detailed descriptions of the newly-developed CV 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.

## Purpose of this Model

The PATH team previously developed a simulation framework for modeling mixed traffic. The existing framework contains an HV model developed based on the Next Generation Simulation (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 CACC vehicles (Milanes and Shladover, 2014). The existing models have been calibrated based on field observations (Kan et al., 2019). The detailed algorithms of the HV, ACC, and CACC models can be found in an existing report (Liu et al., 2018). While the framework can simulate typical AV and CAV systems (e.g., ACC and CACC) and their interaction with HVs, it is incapable of capturing the behavior of CVs. The objective of this study was to properly model CVs for implementation in the framework. The components of the PATH model framework are demonstrated in figure 1.



Source: FHWA.

Figure 1. Diagram. Components of the PATH model framework.

As shown in Figure 1, CVs affect traffic via two major functionalities:

* Serving as the leader of CACC strings; and
* Affecting human drivers’ speed behavior via on-board message display of variable speed limit/advisory (VSL/VSA), which are broadcasted via I2V communication by traffic management centers (TMC).

The first functionality was developed in a previous effort, and the function algorithm was reported in a study by Liu et al. (2018). The remainder of this report gives a detailed description of the second functionality. Particularly, the proposed CV model intended to depict the speed adaptation of connected drivers due to I2V-based speed control and quantify the effects of such behavior changes on mixed traffic. While the average driver can overlook speed information from roadside signs or fail to check their current speed against posted speed limits, the CV system can help avoid such human errors by sending warning messages via on-board devices that motivate drivers to follow speed limits. For example, in a study by Farah et al. (2012), drivers equipped with CVs would receive warning messages when they drove faster than the speed limit. Test results show that the average speed of CV drivers was significantly lower than that of non-CV drivers. In fact, the measured free flow speed of CV-equipped drivers was almost equal to the speed limit. The same trend is also found in a study by Spyropoulou et al. (2014) in which the Intelligent Speed Adaptation system was adopted to promote a uniform speed among drivers in the concerned road segment.

As the CV model is incorporated into the PATH model framework, the simulation tool can be used for a rich variety of traffic scenarios reflecting the effects of the latest technological advancements in vehicle automation and connectively, such as isolated ACC operation, CACC string, and CV speed management. Simulating these scenarios in mixed traffic may generate useful insights of the transportation system, thus benefiting the development future traffic management strategies and infrastructure upgrades. Because the model components in the PATH framework have been carefully calibrated and validated with field data, modeling outputs may reasonably model the traffic flow dynamics expected to appear in the real-world system. Application examples and detailed algorithm descriptions are provided in this document to allow readers to easily adopt PATH models in their own 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 CV modeling, CV implementation has been integrated into the existing PATH 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 PATH framework.

# MODEL DEVELOPMENT AND LOGIC

## Descriptions of Model Logic

A CV is a manually-driven vehicle where the human driver’s behavior could be affected by I2V advisory speeds from TMCs. This research adopts the stimulus-response paradigm to model the behavior adaptation of CV drivers in the traffic stream (Liu et al., 2017). This modeling approach quantifies the reaction of drivers to changes in surrounding traffic conditions, referred to as traffic stimuli. In traditional traffic flow models, a stimulus can be a traffic event, such as the speed change of a preceding vehicle or the variation of the space gap between a subject and front vehicle. In the CV environment, additional stimuli arise from the on-board system that relays the advisory speeds generated by VSL/VSA controls. This research incorporated the I2V-affected speed behavior parameter into a state-of-the-art microscopic car-following model as a factor for drivers’ response sensitivity to traffic stimuli. The car-following model was developed by PATH to depict human drivers’ car-following and lane-changing in various traffic conditions (Liu et al., 2018). The human driver model uses a 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 CV drivers’ speed behavior under the influence of the advisory speed. That model was incorporated into the human driver model to replace the desired speed parameter originally used for conventional human drivers.

CV systems can assist the drivers by enabling quicker and more consistent responses, potentially resulting in fewer traffic disturbances and more stable traffic streams. However, the effectiveness of CV systems relies on driver compliance with CV information. CV systems cannot impact traffic flows unless drivers follow system 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 empirical datasets reported by existing studies (Lu et al., 2019).

## Model Development

### Speed Adaptation of Connected Drivers

When a TMC sends a VSA to CVs via I2V communications, CV drivers adjust their speed behaviors accordingly. Because drivers’ reaction to the VSA is naturally inconsistent, their speed adjustments are expected to follow a random distribution rather than staying at a constant level. For this reason, this study developed an empirical stochastic CV model that captured drivers’ speed patterns under VSA influence. The CV car-following model is developed based on the NGSIM oversaturated flow human driver model that was well-calibrated in Kan et al., (2019). Because CV drivers operate vehicles manually, identical car-following and lane-changing logics for human drivers were applied for CV drivers as well within the model framework. This study revised the original human driver acceleration model to capture CV drivers’ speed under the influence of VSA. The model uses the equation in figure 2 to represent a CV driver’s acceleration:



Figure 2. Equation. Acceleration of a connected vehicle.

where: *aF* is the free acceleration, which describes the driver’s acceleration when the speed choice is not constrained by the preceding vehicles; *aN* is the Newell acceleration term (Newell, 2002), which represents the driver’s acceleration when he or she follows the preceding vehicle; and *aG* is the Gipps acceleration term(Ciuffo et al., 2012), which provides a safety constraint for crash avoidance. The equation variables in figure 2 are defined as follows:













Figure 3. Equation. Variables used for calculating the acceleration of a connected vehicle.

where *aMax* is the driver’s maximum acceptable acceleration; γ is the model coefficient; *v(t)* is the current vehicle speed; and *V0* is the driver’s desired speed. For a normal human driver, the desired speed *V0* is a constant parameter that is determined based on the (fixed) speed limit of the road segment and the driver’s level of compliance with the speed limit; τh is the desired headway; *djam* is the jam gap; *τr* is the reaction time; *vsafe(t+τr )* is the speed of the subject vehicle after reaction time; *vl* is the speed of the preceding vehicle; *bf* is the most severe braking that the subject driver wishes to undertake; *b hat* is the subject driver’s estimate of the preceding vehicle’s most severe braking capabilities; and *d(t)* is the clearance gap with regard to the leader at time *t*.

The three variables in figure 2 represent different aspects of the driver’s behavior. Free acceleration, *aF*, is affected by a driver’s maximum acceptable acceleration and desired speed; car-following term, *aN*, is affected by the desired headway and jam gap; and safety, *aG*, is affected 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 reaction time and desired gap, receive little influence from the connected speed control. For this reason, the VSA algorithm can only change a CV driver’s free acceleration, *aF*. The other two acceleration variables, *aN* and *aG*, remain identical to those used in the human driver model. For this reason, *V0* becomes a variable for a connected driver:



Figure 4. Equation. Desired speed of a connected vehicle.

where *ε* is a random parameter that represents the subject driver’s response to (compliance with) the advisory speed control, *σ* is a parameter that represents the random speed fluctuation around a target speed, and  is the VSL/VSA of the current road section. A human driver is not capable of tracking a target speed perfectly. Even if the driver completely complies with the VSL/VSA, he or she cannot always maintain the vehicle speed at precisely the VSL/VSA level. The vehicle speed is expected to fluctuate around the target speed and be subjected to the limit of the front vehicle speed and relative distance. To capture those factors, the equation adopts σ that follows a normal distribution with a zero mean and a standard deviation of 2.8 m/s. This is identical to human drivers’ speed fluctuation around the posted speed limit. The parameter ε is the major parameter that affects the effectiveness of the CV. A value of zero indicates that the driver completely complies with the VSL/VSA from the on-board speed display. In this case, the CV system functions most effectively. As the parameter deviates from zero, the effectiveness of CV will reduce. In this study, the distribution of ε was obtained from the field data. The calibration and validation of ε is discussed in Chapter 3.

### Traffic Data for Connected Speed Management

The CV affects drivers’ speed behavior via variable speed limits or advisories. This study uses a VSA algorithm derived from the study in Lu et al. (2018). The algorithm allows VSA to be displayed as an advisory speed on roadside variable message signs, directly feed back to drivers in CVs, or used as the set-speed in CAVs. In the proposed model, the algorithm generates tailored speed advisories for individual CVs based on their distance from the bottlenecks and the traffic congestion conditions of the bottlenecks. To generate such variable speed advisories, the algorithm continuously monitors the aggregated speed and traffic occupancy patterns of the freeway corridor to locate active bottlenecks. 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 to the most downstream. This allows the advisory speed derived from a downstream bottleneck to override the advisory speed for upstream bottlenecks if the former is lower than the latter. This process gives priority to downstream bottlenecks when their congestion level is higher than their upstream counterparts, thus increasing the corridor throughput.

The VSA algorithm requires aggregated freeway speed and occupancy data as inputs for the variable advisory speed computation. To obtain the speed dataset, the algorithm first segregates the concerned freeway corridor into consecutive data aggregation sections (as shown in figure 5). The speed data from individual CVs are averaged within a section over an update interval (e.g. 30 seconds in this study). The resulting mean speed is a temporal average of CVs’ space mean speed, which represents the traffic condition for the section over a concerned time period. The computation of average speed is shown in figure 6. In low CV market penetration cases, there might not be any CVs 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 dual loops provide the occupancy datasets along a concerned freeway corridor. Since the VSA algorithm adopts the critical occupancy as the control reference, the measured occupancy may be used to calculate 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.

Figure 5. Plot. Segregation of a freeway corridor.



Figure 6. Equation. Computation of the aggregated speed for a section.

where *v bar* is the speed of the traffic flow, *i* is the ID of the data aggregation section, *k* is the ID of the speed limit update interval, *M* is the number of CVs that have traveled in section *i* during the update interval *k*, *Nm* 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 nth speed sample of vehicle *m*.

A data aggregation section should be short (e.g., less than 500 meters) relative to the length of the freeway corridor. This ensures that speed data from each section can 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 such that they can share the same advisory speed. Otherwise, drivers would need to change the speed frequently as they pass each section, which can reduce traffic stability and driving comfort. The combined section is called the speed control section (figure 5). Each speed control section contains multiple data aggregation sections. The VSA algorithm gradually decreases the advisory speed for vehicles upstream from a bottleneck. The advisory speed reduction takes place discretely at the end point of each speed control section.

### Variable Advisory Speed Computation for Isolated Bottlenecks

To reduce or eliminate traffic congestion at an isolated bottleneck, the VSA algorithm decreases the input traffic flow to the bottleneck by recommending a reduced speed for upstream CVs. In the mixed traffic stream, the scattered CVs 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, which helps 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 originating 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., ∀ j ∈ j ≥ i) is the original posted speed limit Vf. The advisory speed for the speed control sections immediately upstream from the bottleneck (i.e., ∀ j ∈ ⌊(i – j) / Nc⌋ = 1) is:



Figure 7. Equation. Variable advisory speed for sections upstream from the bottleneck.

where Nc is the number of data aggregation sections in each speed control segment, ςo1, ςo2 > 0 are the unitless control gains (ςo1 = 0.6, and ςo2 = 0.6 respectively), Oc is the critical occupancy (Oc = 50 percent in this study), and Oj is the weighted bottleneck occupancy. Oj is calculated by a semi-globally looking ahead algorithm:



Figure 8. Equation. Weighted occupancy computation.

where oi is the measured occupancy, and ρ1, ρ2, and ρ3 are weights (ρ1 = 0.65, ρ2 = 0.2, and ρ3 = 0.15).

The advisory speed of speed control sections at further upstream sections (i.e., ∀ j ∈ ⌊(i – j) / Nc⌋ > 1) gradually increases from the advisory speed of the section immediately upstream from the bottleneck with a step speed of Vs. In addition, the advisory speed of all sections is bounded by the spatial and temporal limits and the maximum and minimum limits:







Figure 9. Equation. Spatial, temporal, and maximum/minimum bounds for the advisory speed.

The VSA algorithm has the following effects:

* Vehicles at the bottleneck section and its downstream sections (i.e., j ≥ i) can travel at the original posted speed limit. This allows the 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 advisory speed change between two consecutive update intervals and two consecutive speed control segments is less than the step speed Vs. With this constraint, the advisory speed changes gradually over time and space. Especially when the VSA algorithm first becomes active, this prevents the controller 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 Vl,min and maximum speed limit Vl,max. For the traffic safety consideration, the maximum advisory speed should be no larger than the posted speed limit. The lower bound is associated with the smallest traffic flow that the VSA algorithm can generate via the speed control. As the lower bound decreases, the algorithm is given more room to regulate the traffic flow.

### 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 may consider all sections in a freeway corridor. Since the (recurrent) congestion usually takes place near the freeway merge, diverge, or weaving areas, considering sections outside of those regions could increase the computation burden of the controllers. Thus, the proposed bottleneck-searching procedure only examines a subset of the data aggregation sections that covers the merge, diverge, and weaving segments. To this end, the algorithm requires 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 (or diverge or weaving) segment are mapped to a unique link ID. If a link contains Nl data aggregation sections, a data aggregation section p is identified as a bottleneck if the average speed:



Figure 10. Equation. Identification of bottleneck sections.

where p is the section number (p ∈ [1, Nl]). Section p is upstream from section q, if p < q, and VT is the threshold speed. VT can be set as a function of the free flow speed Vf, (i.e., VT = β∙Vf, where the coefficient 0 < β < 1). The bottleneck search moves from section 1 to section Nl (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 to increase the computation efficiency. Afterwards, the VSA algorithm uses the equations in figures 7 through 9 to determine the speed limits for **all** data aggregation sections of the corridor.

The VSA algorithm applies the above bottleneck search 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:



Figure 11. Equation. Final advisory speed for a section.

where Vl,ju is the advisory speed computed for the upstream link, and Vl,jd is the advisory speed computed for the downstream link. With this updated 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 lower 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.

# MODEL CALIBRATION AND VALIATION

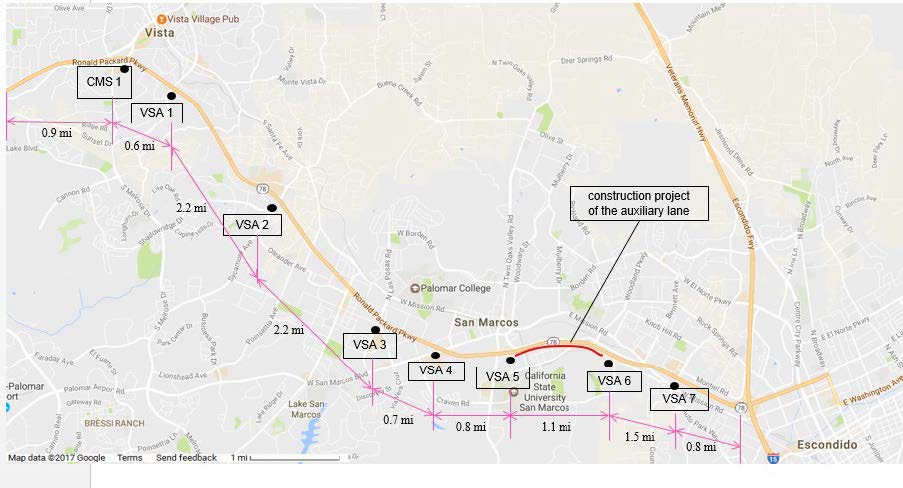
## Assumptions in Calibration and Validation

The parameter ε in figure 4 denotes the deviation of a driver’s desired speed from a given advisory speed. It can describe the driver’s compliance level to the speed control. For this reason, the parameter should be calibrated and validated based on speed datasets that depict the behavior of real-world drivers under the influence of an active VSL/VSA control. To determine ε, the research adopts the following assumptions:

* The calibration and validation require datasets 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 could be reflected by drivers’ speed choices when they have the freedom to select their speed levels. This usually happens in uncongested roads. In congested traffic, on the other hand, the vehicle movement is restricted by the preceding vehicles. Thus, the speed data in congested traffic cannot depict the desired speed distributions.
* Since there is no massively-implemented VSA/VSL system that connects with drivers via I2V systems, it is impossible to obtain enough speed data for CVs for the model calibration and validation. To address the data limitation, CV driver speed patterns are assumed to be similar or close to the behavior of normal human drivers affected by the VSA/VSL displayed on the roadside changeable message signs. This assumption justifies the use of VSA field test data on San Diego, CA for model calibration and validation.
* Once the CV model captures the desired speed preference of individual CV drivers, it could be further used for microscopic simulation of mixed traffic, which would enable sensitivity analyses that can reveal the impact of CV market penetration levels on traffic flow and vehicle energy efficiency.

## Calibration and Validation Datasets

With the above assumptions, the PATH team calibrated and validated ε with field data that describes the free speed choices of normal human drivers under the influence of VSA. The calibration and validation data were collected from a California Department of Transportation (Caltrans) Project, “Field Experiment of Variable Speed Advisory (VSA),” which included an extensive traffic data collection (Lu et al., 2018). The VSA data samples were used because most of the U.S. transportation agencies adopt VSA instead of VSL for traffic management due to institutional issues. This field test was conducted on the State Route 78 Eastbound (SR-78E) from Vista Village Drive in the City of Vista to the freeway interchange of SR-78E, and on Interstate 15 (I-15) in the city of Escondido, California, as shown in the figure 12. The roadway segment is a three-lane freeway with a posted speed limit of 29.1 m/s (65 mph), with 10 on-ramps and 10 off-ramps. This freeway corridor contained high traffic volume in both AM and PM peak hours. Data was collected for five weeks, between March 30, 2018 and May 4, 2018, with VSA activation.



Original map: © 2017 Google (see Acknowledgements section).

Figure 12. Plot. Overview of system scope, VSA sign and Changeable Message Sign (CMS) locations, and the construction area (Lu et al., 2019).

Recurrent bottlenecks were observed daily on weekdays in the study road segment. The morning bottleneck occurred between 6 AM and 9 AM at two locations, one near San Marcos Blvd (postmile of 12.27) and another near interchange of SR-78E to US-15 (postmile16.6). The evening bottleneck occurred between 2 PM and 7 PM, near the interchange of SR-78E to US-15NB and US-15SB15 southbound. Vehicle speeds dropped from 29.1 m/s (65 mph) to as low as 6.7 m/s (15 mph) after the onset of the congestion. A construction project was located on the new auxiliary lane in both directions of SR-78, between Twin Oaks Valley Rd and Woodland Parkway, and a speed limit of 24.6 m/s (55 mph) was posted for that stretch of the roadway.

Seven VSA signs were located at the critical locations determined from the analysis on the recurrent bottlenecks. Calculated VSA values were then rounded to multiples of 2.2 m/s (5 mph) and displayed on the VSA signs. The VSA signs were located on the roadside to display the Advisory Speed, which was updated every 30 seconds in real time. 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 speed posted on the VSA during morning and evening peak hours was recommended to drivers, but not enforced.

Traffic data was collected from two sources. Traffic flow, speed, and occupancy data was collected from loop detectors in the study corridor at 30-second intervals. The data was subsequently fused with real-time speed and speed advisory data–captured every 30 seconds by radar sensors mounted on the VSA display equipment–at seven different sites along the 17.4-km (10.8-mile) section of SR-78E. These two sources of data were then processed for the estimation of the overall traffic state along the corridor, which, in turn, was used to estimate the effectiveness of VSA on drivers’ speed behaviors.

The entire dataset, including traffic volume, speed, occupancy, and VSA level, provides five weeks’ worth of data samples. Data samples of four weeks will be used to determine the distribution of ε. The remaining data samples will be adopted for model validation.

## Calibration and Validation Methodology

The methodology used to calibrate and validate the random distribution of the drivers’ response to the speed limit control is provided below.

### 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 2.2 to 29.1 m/s (5 to 65 mph) in increments of 2.2 m/s (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 11.2 m/s (25 mph). On the other hand, the field data showed a similar bimodal pattern for lower VSA levels from 2.2 to 15.6 m/s (5 to 35 mph), with a peak at 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 17.9 to 29.1 m/s (40 to 65 mph), with a peak near 0 m/s, indicating good compliance. For this reason, the data samples were grouped into two VSA levels: low VSA level, from 0 to 15.6 m/s (0 to 35 mph), and high VSA level, from 17.9 to 29.1 m/s (40 to 65 mph). Note that the VSA feedback to the driver is a multiple of five. Therefore, such division does not lose the generality.

For each VSA level, the deviation data was divided randomly into a calibration set and a validation set. For each set of speed observations, ulow,i ∈ Ulow for low VSA level and uhigh,i ∈ Uhigh for high VSA level, 70 percent of the data was randomly selected for the calibration dataset, εc,low,i ∈ Υlow and εc,high,i ∈ Υhigh, where εc,low,i = ulow,i - Vl and εc,high,i = uhigh,i - Vl. The remaining 30 percent of the data was selected for the validation dataset, εv,low,i ∈ Plow and εv,high,i ∈ Phigh, where εv,low,i = ulow,i -Vl and εv,high,i = uhigh,i - Vl.

### 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 dataset. More formally, given the data points of sample size, N, Y1, Y2, … YN are sequenced in increasing order. The ECDF is defined as ECDF(i) = n(i) / N , where n(i) is the number of data points less than Yi. Using the calibration set Υ, the ECDFs of ε were found for the low VSA level from 0 to 15.6 m/s (0 to 35 mph), and for the high VSA level from 17.9 to 29.1 m/s (40 to 65 mph). The difference between the observed speed and VSA for low and high VSA levels are noted as εlow and εhigh, respectively. A Python package, statsmodels, provides statistical model estimation functions. This package was used to determine the ECDFs of εlow and εhigh as Fc,n,low(ε) and Fc,n,high(ε), respectively.





Figure 13. Equation. Empirical cumulative distribution functions for the low and high VSA levels.

### Step 3: Distribution Validation

The two-sample Kolmogorov-Smirnov test (KS test) is a statistical test used to validate if two samples come from an empirical distribution of a continuous random variable. In this work, the KS test is performed to validate the calibrated distribution, Fc,n,low(ε) and Fc,n,high(ε). 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, Dn,m, is given as the following:



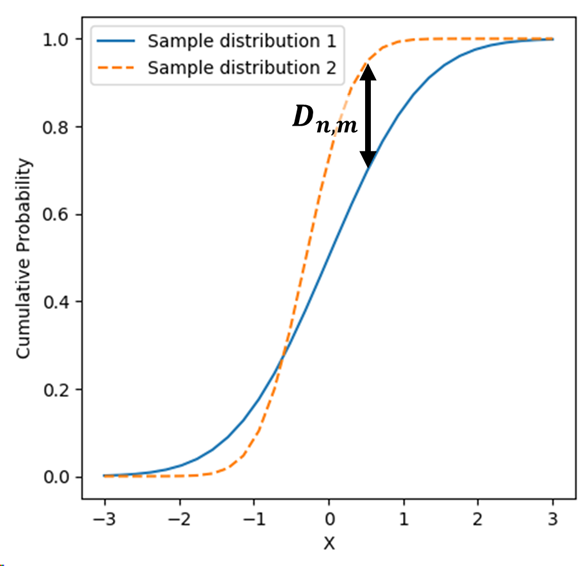
Figure 14. Equation. Kolmogorov-Smirnov test statistic.

where Fc,n(ε) is the ECDF of the calibration data set of size n, and Fv,m(ε) is the ECDF of the validation data set of size m. The test statistic, Dn,m, is the largest difference between the two curves, Fc,n,low(ε) and Fv,n,high(ε), as shown in figure 15 below. The null hypothesis is rejected at a significance level α for large samples, if:



Figure 15. Equation. Criterion for rejecting the null hypothesis.

where the critical value is given as c(α=0.05) = 1.36. The null hypothesis is accepted otherwise.



Source: FHWA.

Figure 16. Diagram. Visualization of the Kolmogorov-Smirnov test statistic.

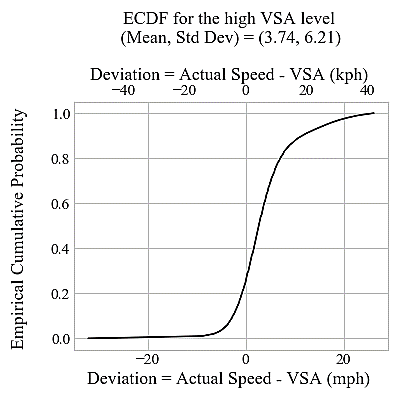
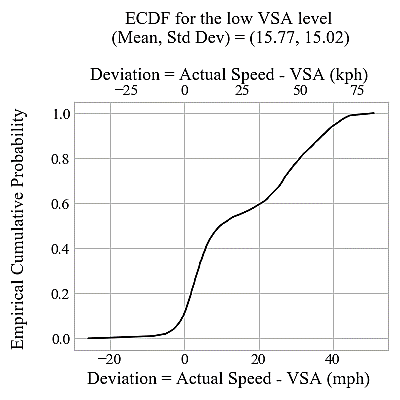
## Calibration and Validation Results

The sample size for the calibration and validation datasets for low and high VSA levels are given below. Since the data was gathered from a field observation of drivers’ speed on the public road, the sample collection was not performed in a controlled experiment. Though 30 is generally a sufficient sample size for a statistical analysis for well-designed random experiments, this rule may not be suitable for this uncontrolled observation. In addition, ε for each 2.2 m/s (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, however, the deviation for the low and 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 VSA cases.

|  |  |  |
| --- | --- | --- |
|  | Low VSA  (0-15.6 m/s, 0-35 mph) | High VSA  (17.9-29.1 m/s, 40-65 mph) |
| Sample size of calibration dataset | 5,380 | 73,706 |
| Sample size of validation dataset | 2,306 | 31,589 |
| Total sample size | 7,686 | 105,295 |

The calibration results of the empirical cumulative distribution functions for the low and high VSA levels are shown in figure 17. The x-axis represents the deviation in miles per hour, while the y-axis represents 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.

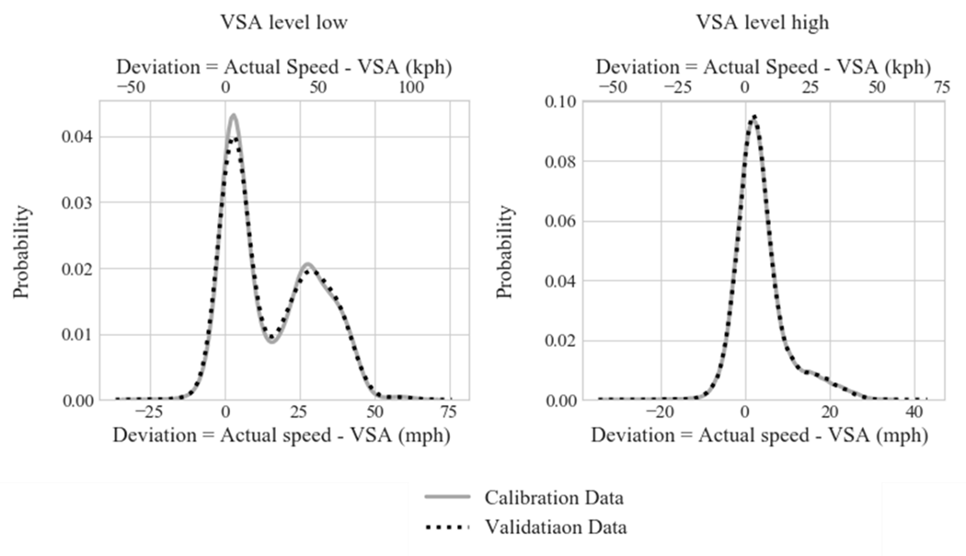
Figure 17. Diagram. Calibrated empirical distributions of ε for the low and high VSA levels (mean and standard deviation in mph).

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

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

|  |  |  |
| --- | --- | --- |
|  | Low VSA (0-35 mph) | High VSA (40-65 mph) |
| Null hypothesis | Data samples 1 and 2 follow a common distribution | |
| Data sample 1 | Calibration datasets | |
| Data sample 2 | Validation datasets | |
| Level of significance | 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,  **accept the null hypothesis**. | Since the D statistic is smaller than the critical D-value,  **accept the null hypothesis**. |

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 datasets in figure 18.



Source: FHWA.

Figure 18. Diagram. The probability density functions for the calibration and validation datasets.

# BASIC GUIDANCE ONE MODEL IMPLEMENTATION

## modeling Process

Understanding the update process of the simulation tool is essential for implementing the proposed CV 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 needs to model thousands of vehicles in a network. In general, the computation process within an update interval can be divided into three stages: information gathering, new system status computation, and system status update.

### Stage 1: Information Gathering

The information gathering 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 inputs from the previous stage. In this stage, the tool calculates the new position, speed, and acceleration for individual modeled vehicles; new state for the traffic signals; and 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, and the signal lights change color. The new scheme of the traffic management strategies are executed, which concludes the update interval.

The proposed CV algorithm may be easily implemented in the three-stage update framework:

* The algorithm requires CV speed and loop detector occupancy as inputs. Those inputs may be 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) should provide functions to access attributes of simulated entities. Those functions may be called to collect the inputs for the CV algorithm.
* The CV algorithm computes the advisory speed based on the CV speed and loop occupancy inputs. The computation process may be 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 may be integrated into the customized update functions.
* The computed advisory speed must be sent to individual CVs in the traffic stream, which may be achieved in the final stage. In this case, the user must develop a function to update the desired speed for only the CV type, while keeping the original desired speed for other vehicle types.

# SENSITIVITY STUDY

## Implementation of the developed model into a traffic simulation tool

The CV model was implemented in the PATH modeling framework by creating a new CV vehicle class, in addition to the existing HV, ACC, and CACC classes. The simulation framework implements the car-following (CF) and lane-changing (LC) mechanisms for the four vehicle classes in the Aimsun simulation package via its MicroSDK and API programming tools. The MicroSDK contains two essential classes—a behavior model (BM) class and a simulated vehicle (SV) class. The BM class configures the global parameters (e.g., simulation time, time step, and Origin-Demand matrix) of the simulation environment, and controls the process of each simulation run (e.g., start/end of 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 PATH model framework can reproduce the patterns of a traffic flow mixed with regular human-driven, connected human-driven, ACC, 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 CV speed behaviors.

## Design of simulation experiments

The developed CV model was tested under various simulation scenarios. Those scenarios considered the CV market penetration rate (MPR) and the human drivers’ compliance level 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 to full compliance level. The baseline compliance level was obtained based on the empirical data described in figure 17. A modeled driver’s compliance (i.e., ε in figure 4) was determined via the following two steps:

* **Step 1.** Generate a random real number µ that follows a uniform distribution between 0 and 1.
* **Step 2.** Take µ as a cumulative probability, and find its corresponding ε in figure 17.

With the above steps, the driver’s speed deviation from the VSA would follow the empirical distribution identified in the model calibration. In scenarios where drivers’ compliance level is increased, the random compliance ε changes based on the increased rate of the compliance. If the compliance level increases by θ (0 < θ <= 1), the new compliance is given:



Figure 19. Equation. Determination of the increased compliance level.

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

Table 3. Simulation scenarios for CVs.

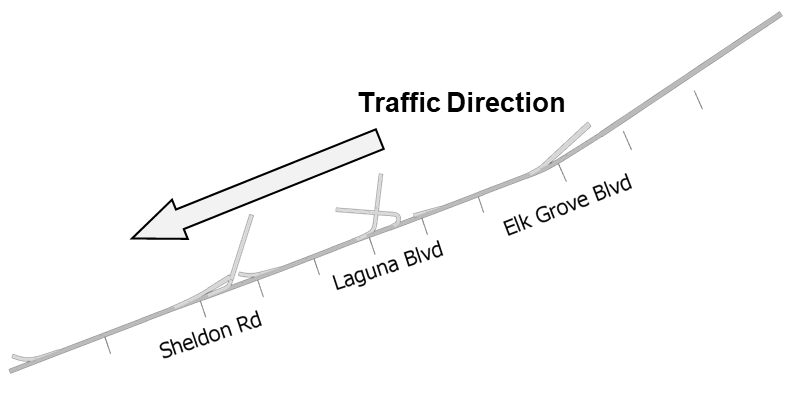
|  |  |  |
| --- | --- | --- |
| ID | CV MPR (%) | Compliance Increase θ |
| 1 | 0 | N/A |
| 2 | 10 | 0 (baseline compliance) |
| 3 | 20 | 0 |
| … | … | 0 |
| 11 | 100 | 0 |
| 12 | 10 | 0.05 |
| 13 | 10 | 0.10 |
| 14 | 10 | 0.15 |
| … | … | … |
| 16 | 10 | 0.30 |
| 17 | 20 | 0.05 |
| 18 | 20 | 0.10 |
| 19 | 20 | 0.15 |
| … | … | … |
| 22 | 20 | 0.30 |
| 23 | 100 | 0.05 |
| 24 | 100 | 0.10 |
| 25 | 100 | 0.15 |
| … | … | … |
| 28 | 100 | 0.30 |

In addition to CVs, 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 as the following:

Table 4. Simulation Scenarios for CACC vehicles.

|  |  |  |
| --- | --- | --- |
| ID | CACC MPR (%) | VSA On or Off |
| 33 | 10 | Off |
| 34 | 20 | Off |
| 35 | 30 | Off |
| 36 | 10 | On |
| 37 | 20 | On |
| 38 | 30 | On |

The simulation scenarios were evaluated on a 7-kilometer freeway corridor (see figure 20). The freeway corridor was coded into the Aimsun simulation tool based on the real-world Statue Route 99 (SR-99) corridor south from downtown Sacramento, CA. The corridor contained three on-ramp bottlenecks at the Elk Grove Boulevard, Laguna Boulevard, and Sheldon Road interchanges, respectively. The Sheldon Road interchange is the most downstream bottleneck causing the most severe congestion during the morning peak hours. The simulation runs cover an 8-hour period from 4 AM to 12 PM. The formation and recovery of the bottlenecks may be captured entirely during the simulation period. The simulation inputs were real-world traffic counts obtained from the Performance Measurement System (PeMS) database.



Source: FHWA.

Figure 20. Diagram. Simulated freeway corridor.

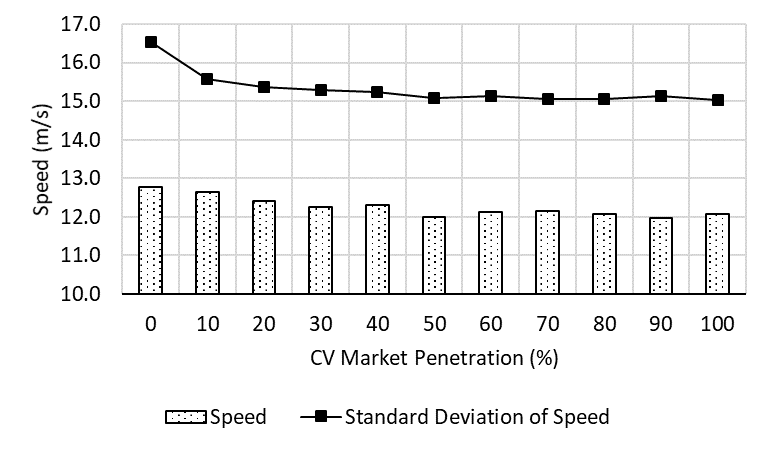
## Simulation results for the different scenarios

The impacts of CV market penetration are depicted by the average traffic speed and vehicle fuel economy. The fuel results were computed using two models: the Virginia Tech 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 could provide a comprehensive picture on the effectiveness of the CV speed adaptation. The parameters used in the two fuel models are listed in table 5.

Table 5. Fuel model parameters.

|  |  |  |  |
| --- | --- | --- | --- |
| MOVES Model Parameters | | Virginia Tech Model Parameters | |
| Rolling resistance (kw-s/m) | 1.56E-1 | Alpha 0 | 5.92E-4 |
| Rotational resistance (kw-s2/m2) | 2.00E-3 | Alpha 1 | 4.24E-5 |
| Aerodynamic drag coefficient (kw-s3/m3) | 4.93E-4 | Alpha 2 | 1.00E-6 |
| Vehicle age distribution | California average vehicle age distribution in 2019 | Vehicle mass (kg) | 1453 |
| Fuel supply | MOVES default | Driveline efficiency | 0.92 |
| Fuel formulation | MOVES default | Density of air at sea level at a temperature of 15 °C (kg/m3) | 1.23 |
| Alternative vehicle and fuels technology | MOVES default | Vehicle drag coefficient | 0.30 |
| Temperature | Average July temperature measured in Sacramento, CA | Correction factor for altitude | 1.00 |
| Humidity | Average July humidity measured in Sacramento, CA | Vehicle frontal area (m2) | 2.32 |
|  |  | Rolling resistance parameters Cr, C1 and C2 | 1.75, 3.28E-1, 4.58 |

As figure 21 shows, the average speed of the traffic flow slightly decreased with CV market penetration increase. The decrease is between 2 and 6 percent. The speed decreased because the CVs upstream from the bottleneck adopted the reduced speeds recommended by the VSA algorithm. As CVs began to slow down, they caused following vehicles to reduce their speeds as well, eventually leading to the slowdown of traffic flow. The speed reduction helped the bottleneck recover from congestion, as queued vehicles were released faster and spent less time in congested status. For this reason, the average speed of modeled vehicles did not drop significantly despite CVs suggesting that drivers travel at speeds much lower than the posted speed limit.

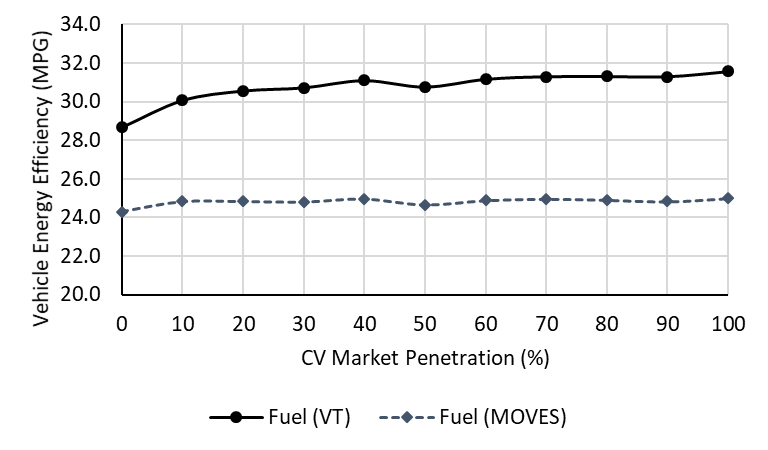


Source: FHWA.

Figure 21. Diagram. Average vehicle speed and standard deviation of the speed under various CV market penetrations.

The standard deviation of the speed decreased with CV market penetration increase, which indicates that speed behavior became more consistent as more vehicles received the VSA. Notably, speed variation showed the most significant decrease when CV market penetration increased from 0 to 10 percent. Further increases in CV market penetration rate resulted in only minor changes in speed deviation. This finding suggests that traffic flow can be substantially improved even with a small portion of vehicles following the VSA. The speed adaptation of the small percentage of CVs following the VSA can affect following non-CVs, thereby resulting in changes to overall traffic flow pattern.

The reduction of speed variation led to the harmonization of vehicle speed, which resulted in improved vehicle fuel efficiency. As figure 22 shows, vehicle fuel efficiency increased with CV market penetration. The largest increase was observed when CV market penetration increased from 0 to 10 percent, which is consistent with the trend in speed variation reduction. The Virginia Tech model gave a higher estimation of vehicle fuel efficiency than did the MOVES model, as the Virginia Tech model used parameters of more recent vehicles than did the MOVE model (newer vehicles generally have higher energy performance than older vehicles). Despite differences in result magnitudes, the two models produce curves with similar trends.



Source: FHWA.

Figure 22. Diagram. Average vehicle fuel efficiency under various CV market penetrations.

The effects of driver compliance are depicted in tables 6, 7, and 8. The results indicate that a small change in driver compliance level has little influence on both average speed and fuel efficiency. The speed and fuel performance of the studied freeway corridor changed less than 2 percent as driver compliance level increased from 5 to 30 percent. This trend is consistent when CV market penetration is 10 percent, 20 percent, or 100 percent, implying that VSA performance is not sensitive to small increases in driver compliance, regardless of CV market penetration levels.

Table 6. Effects of driver compliance level on speed and vehicle fuel efficiency at 10 percent CV market penetration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compliance Increase (%) | Fuel Efficiency from MOVES Model (mpg) | Fuel Efficiency from Virginia Tech Model (mpg) | Vehicle Speed (m/s) | Std of Vehicle Speed (m/s) |
| 0 | 24.8 | 30.1 | 12.7 | 15.6 |
| 5 | 24.8 | 30.0 | 12.6 | 15.5 |
| 10 | 25.0 | 30.3 | 12.8 | 15.4 |
| 15 | 24.7 | 30.1 | 12.4 | 15.4 |
| 20 | 24.9 | 30.2 | 12.6 | 15.4 |
| 25 | 24.7 | 30.0 | 12.4 | 15.3 |
| 30 | 25.0 | 30.3 | 12.7 | 15.3 |

Table 7. Effects of driver compliance level on speed and vehicle fuel efficiency at 20 percent CV market penetration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compliance Increase (%) | Fuel Efficiency from MOVES Model (mpg) | Fuel Efficiency from Virginia Tech Model (mpg) | Vehicle Speed (m/s) | Std of Vehicle Speed (m/s) |
| 0 | 24.8 | 30.6 | 12.4 | 15.4 |
| 5 | 24.7 | 30.4 | 12.2 | 15.3 |
| 10 | 25.1 | 30.8 | 12.5 | 15.1 |
| 15 | 24.9 | 30.7 | 12.4 | 15.2 |
| 20 | 25.0 | 30.8 | 12.5 | 15.1 |
| 25 | 25.0 | 30.8 | 12.4 | 15.2 |
| 30 | 25.1 | 30.9 | 12.5 | 15.1 |

Table 8. Effects of driver compliance level on speed and vehicle fuel efficiency at 100 percent CV market penetration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compliance Increase (%) | Fuel Efficiency from MOVES Model (mpg) | Fuel Efficiency from Virginia Tech Model (mpg) | Vehicle Speed (m/s) | Std of Vehicle Speed (m/s) |
| 0 | 25.0 | 31.6 | 12.1 | 15.0 |
| 5 | 25.1 | 31.6 | 12.1 | 15.1 |
| 10 | 25.0 | 31.5 | 11.9 | 14.9 |
| 15 | 25.0 | 31.5 | 11.9 | 15.0 |
| 20 | 24.9 | 31.5 | 11.8 | 14.8 |
| 25 | 25.1 | 31.8 | 11.9 | 14.8 |
| 30 | 25.2 | 31.9 | 11.9 | 14.8 |

Speed and vehicle fuel efficiency did not significantly change as the compliance level further increased to 100 percent (see table 9). However, increasing compliance levels led to substantial changes in traffic flow patterns. Figures 23 and 24 show the fundamental diagrams of the bottleneck link at a CV market penetration rate of 10 percent. Figure 23 depicts the results when the baseline compliance level is applied, whereas figure 24 illustrates the results with 100 percent improvement of the compliance level. When compared to figure 23, the points in figure 24 shift left and up toward higher flow regions, which indicates improvement in bottleneck traffic flow as a result of CV speed adaptation. However, the traffic flow increase at the bottleneck did not benefit the overall performance of the freeway corridor, for reasons explained below.

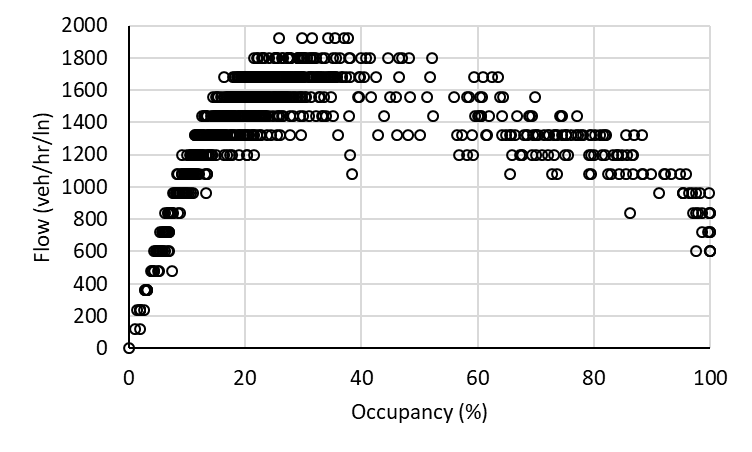
The increase in compliance level resulted in the increased capability of CVs to regulate traffic flow. Consequently, the CV speed adaptation could more effectively reduce the input flow to the bottleneck, leading to a decrease in the congestion level and an increase in the bottleneck traffic flow rate. These events explain the left and upward shift of the points in the fundamental diagram in figure 24. On the other hand, the VSA algorithm implemented in the analysis is a reactive algorithm. It reduces the input traffic flow to the bottleneck during congested periods, increasing the input flow only after congestion dissipates. The delayed response of the VSA algorithm does not always benefit traffic flow.

In some cases, the bottleneck is congested, while the on-ramp traffic will be light in the next a few minutes. The congestion level should reduce due to the decrease of the on-ramp traffic disturbance. This naturally leads to an increase of the bottleneck flow rate. However, the VSA algorithm will continue reducing the input flow to the bottleneck at this case because it detects the congestion status of the bottleneck. This can overly reduce the input flow, making some of the bottleneck capacity underused. This condition is represented by points in the dashed box in Figure 24, where the occupancy is in the middle range but the flow rate is overly reduced by the VSA.

In other cases, the bottleneck congestion has been relived, but the heavy on-ramp traffic will break down the bottleneck traffic again in the next few minutes. The VSA algorithm would increase the input flow because it only detects the light traffic condition of the bottleneck. This could create a much heavier congestion than the no control case. Figure 24 represents this condition with the points in the dotted box–the capacity reduction caused by the excessive traffic input is much higher than the rest points in the same occupancy range. As the increase in driver compliance level leads to enhanced VSA control capability, the mentioned undesirable control behaviors also occur more frequently. The negative effect of those behaviors can offset the benefits from bottleneck flow increase. As a result, the overall effect of the compliance increase on freeway corridor performance becomes a minor one.

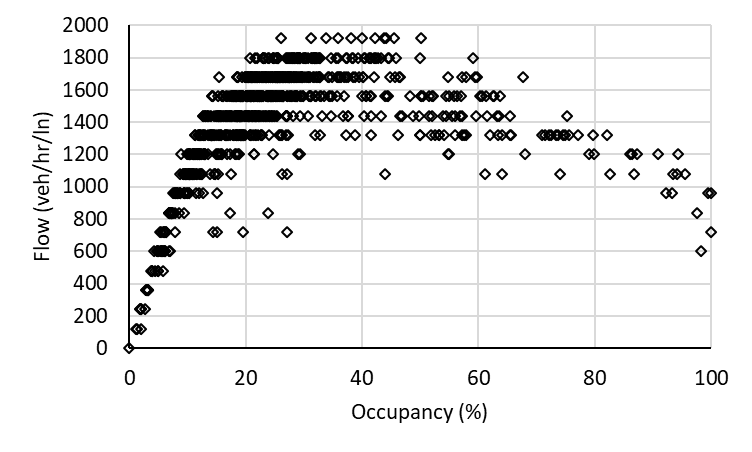
Table 9. Comparison of the baseline compliance and the full compliance level.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CV MPR (%) | Compliance Increase (%) | Fuel Efficiency from MOVES Model (mpg) | Fuel Efficiency from Virginia Tech Model (mpg) | Vehicle Speed (m/s) | Std of Vehicle Speed (m/s) |
| 10 | 0 | 24.8 | 30.1 | 12.7 | 15.6 |
| 10 | 100 | 25.1 | 30.2 | 12.2 | 15.1 |
| 20 | 0 | 24.8 | 30.6 | 12.4 | 15.4 |
| 20 | 100 | 25.1 | 30.7 | 11.9 | 14.9 |
| 100 | 0 | 25.0 | 31.6 | 12.1 | 15.0 |
| 100 | 100 | 24.8 | 31.3 | 11.1 | 14.6 |



Source: FHWA.

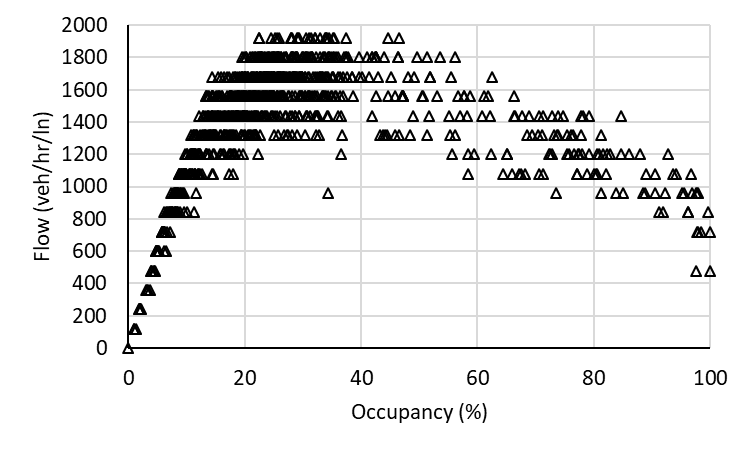
Figure 23. Diagram. Fundamental diagram of 10 percent CV market penetration with the baseline compliance level.



Source: FHWA.

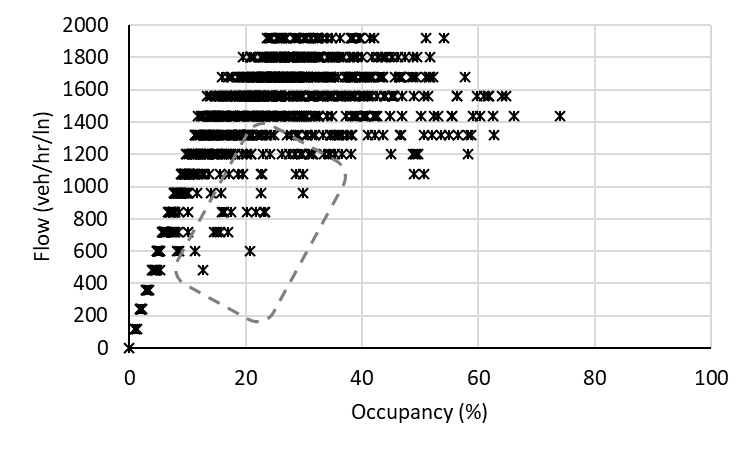
Figure 24. Diagram. Fundamental diagram of 10 percent CV market penetration with the full compliance level.

When the VSA algorithm is applied with CACC vehicles, the vehicles perfectly adopt the advisory speed. As a result, the previously mentioned undesirable control behaviors become more prominent in the CACC case than in the CV case. Figures 25 and 26 display the fundamental diagrams with and without I2V advisory speed, respectively, for the 30 percent CACC market penetration case. Comparison of the two plots reveal that congestion at the bottleneck has been largely removed by the VSA control. Nonetheless, the plots also demonstrate an increase in cases of bottleneck capacity underuse, as shown by the relatively large number of points in the dashed box in figure 26. In this case, queued vehicles upstream from the bottleneck spent more time in the congested condition, reducing the speed and vehicle energy performance of the corridor. Table 10 provides a comparison of traffic speed and vehicle fuel economy in the presence and absence of VSA. Unlike previous CV cases, the two energy models generate inconsistent results with regards to vehicle fuel efficiency when examined with and without the VSA. In addition, average vehicle speed is significantly reduced, although speed variation is also reduced with the VSA control. The analysis results do not indicate that the VSA algorithm is beneficial to traffic flow under the CACC environment.



Source: FHWA.

Figure 25. Diagram. Fundamental diagram of 30 percent CACC vehicle market penetration without the VSA.



Source: FHWA.

Figure 26. Diagram. Fundamental diagram of 30 percent CACC vehicle market penetration with the VSA.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Fuel Efficiency from MOVES Model (mpg) | Fuel Efficiency from Virginia Tech Model (mpg) | Vehicle Speed (m/s) | Std of Vehicle Speed (m/s) |
| No VSA | 25.6 | 28.9 | 14.6 | 15.3 |
| VSA | 24.3 | 29.6 | 11.3 | 13.7 |
| Δ | -4.8% | 2.4% | -22.8% | -10.2% |

The above results indicate that the reactive nature of the VSA controller impedes improvements in speed harmonization and vehicle fuel efficiency, regardless of CV driver or CACC vehicle compliance with 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 short-term disturbances from on-ramp traffic and subsequently compute an input flow to the bottleneck that matches the bottleneck traffic condition, leading to more efficient bottleneck operation. However, because traffic flow changes rapidly within the bottleneck area, a fast and localized speed control is needed to create input flows that maximize bottleneck throughput. Such speed controls may be difficult for human drivers to implement. In this sense, the predictive VSA algorithm is expected to perform best with an automated speed controller, such as CACC.

Another potential avenue for improvement involves combining on-ramp metering with VSA control. With ramp metering, the disturbance caused by on-ramp traffic becomes controllable. The VSA algorithm can generate speed commands based on the metering rate, and the combined control algorithm is easy to implement for human drivers.

# SUMMARY AND RECOMMENDATIONS

This document presents the modeling approach used by the PATH team to depict CV driver compliance behaviors in response to the I2V-based VSA in mixed traffic. CV driver speed adaptation due to VSA was captured by a stimulus-response model, which allows the desired speed of drivers to change based on the VSA level. A random variable was used to model the fluctuation of the desired speed in the traffic flow. Field VSA test data was used for model calibration and validation to allow the speed adaptations of CV drivers to represent real-world driver speed patterns under the influence of the VSA. CV driver compliance was found to be best modeled with empirical distributions of two speed groups: the low speed group when the VSA is under 35 mph, and the 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 empirical distributions parameters. The two-sample Kolmogorov-Smirnov test was conducted to show the goodness of fit of those empirical cumulative distribution functions with the real-world data. The validated model has been applied to evaluate the effectiveness of CV on freeway corridor operations.

It should be noted that the model calibration and validation datasets represent the response of drivers to the VSA displayed on the roadside message sign, not to direct feedback in the vehicle. Admittedly, such feedback will have less influence on the driver than would in-vehicle displays. For this reason, this research designed a sensitivity analysis to explore the effectiveness of CV speed adaptation under various CV 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 I2V-based VSA control could have significant effects on the freeway corridor when CV market penetration is 10 percent. With VSA, speed variation decreased by 6 percent, while vehicle fuel efficiency increased by 2.2 to 4.9 percent depending on the results of different energy models. Results suggest that the speed adaptation of a few connected drivers could substantially change the traffic flow pattern, leading to a more energy-efficient traffic flow. As CV market penetration further increased, reductions in speed variation and improvements in fuel efficiency became less rapid. When CV market penetration reached 100 percent, speed variation decreased by 9.1 percent, while vehicle fuel economy increased by 2.9 to 10.1 percent. The performance of the VSA algorithm was not sensitive to small changes in driver compliance level. However, traffic flow patterns changed significantly when CV drivers fully complied with the VSA. Full compliance did not produce benefits for speed harmonization or vehicle fuel efficiency, as the VSA algorithm in this study could not generate advisory speeds based on predicted traffic conditions. A similar shortcoming was also observed in the CACC cases. When the VSA is implemented with CACC, the CACC controller would perfectly adopt the advisory speed as the reference speed. Nonetheless, adding the VSA algorithm to CACC vehicles did not bring notable benefits to the freeway corridor, as the VSA controller tends to underuse bottleneck capacity due to its delayed response to traffic variations. To address this shortcoming, a predictive VSA algorithm, or the combined application of VSA and ramp metering, is recommended.

The methods to implement the proposed CV model, as well as the PATH human driver, ACC, and CACC models, in the simulation tool have also been reported in the research. The detailed algorithm description and pseudo codes are given in the Appendices of the document. These materials are helpful for researchers implementing the PATH modeling framework in their customized tools.

# Acknowledgments

The original map in figure 12 is the copyright property of Google® Earth™ and can be accessed from https://www.google.com/earth.(Google®. 2017). The map shows the locations of VSA signs, CMS signs, and a construction area, in the study area (the State Route 78 Eastbound (SR-78E) from Vista Village Drive in the City of Vista to the freeway interchange of SR-78E, and on Interstate 15 (I-15) in the city of Escondido, California).

# 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.

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 PATH 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 PATH 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 PATH model is named mybehavioralModel. The logic flow of this class is demonstrated in figure 27. In the figure, the bold notations indicate the PATH 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 27. 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 PATH simulation algorithm, the new vehicle creation and LC/CF update rely on functions in the simulated vehicle (SV) class. The SV class in the PATH framework is named myVehicleDef. When a new vehicle is created, the PATH 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 11. 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::setEstimateLeaderDecCoeff | A coefficient representing the uncertainty when estimating the max deceleration of the leading vehicle. |
| 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::setRandomPolitenessOptional | Politeness threshold. |
| 16 | myVehicleDef::setFrictionCoef | Friction coefficient used for determining the free flow speed in different lanes. |
| 17 | myVehicleDef::setGapAcceptanceModel | 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::setLaneChangeDesireThrd | 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. |
| 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/s2 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. |
| 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.

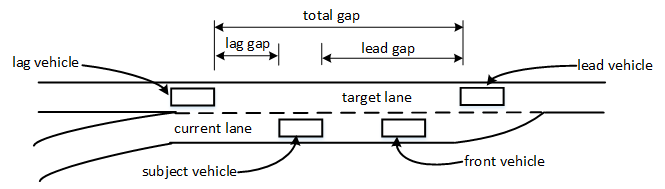
-1, otherwise.

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 28. The overall logic flow of the CF and LC algorithms is shown in figure 29. 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 29, 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 30 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 figures 31 through 33. Figure 31, in particular, displays the CF functions when the modeled driver intends to make an ALC or MLC towards the freeway exit. Figure 32 shows the CF logic for drivers that make the MLC on an on-ramp. Figure 33 demonstrates the CF logic for drivers that make DLCs. Sections C.1 and C.2 present detailed descriptions for functions in figures 29 through 33.





Source: FHWA

**Figure 28. Diagram. Vehicles involved in the CF and LC interactions.**



Source: FHWA

**Figure 29. Diagram. CF and LC logic for manually-driven vehicles.**



Source: FHWA

**Figure 30. Diagram. Driving mode determination for manually-driven vehicles.**



Source: FHWA

**Figure 31. Diagram. BCF logic flow for ALC and MLC (off-ramp exiting maneuver).**



Source: FHWA

**Figure 32. Diagram. BCF logic flow for MLC (on-ramp merging maneuver).**



Source: FHWA

**Figure 33. 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 35 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 36). 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 PATH 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 34. Diagram. Model logic flow for ACC vehicles.**



Source: FHWA

**Figure 35. 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 36 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,* whichidentifies the driving mode based on the vehicle status in the current step (figure 37). 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 PATH 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 36. Diagram. Model logic flow for CACC vehicles.**



Source: FHWA

**Figure 37. 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 Vehicle Functions

The car-following and lane-changing logics for CV are the same as the HV algorithms described in Appendix C. The difference between a CV and an HV is that the CV can receive real-time VSL/VSA from the TMC. Once receiving the VSL/VSA, the CV driver changes the desired speed based on the VSL/VSA level and the compliance level. The PATH 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 PATH 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 PATH framework only needs to run the VSL/VSA algorithm once in every simulation interval to generate updated desired speed levels for all CVs. This creates an efficient model implementation that is important for evaluating large road networks. The CV 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 CVs 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.