CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

SmartPath Regulation Layer Implementation: A User's Guide

Jason Carbaugh, Luis Alvarez
Pin-Yen Chen, Roberto Horowitz
University of California, Berkeley

California PATH Research Report
UCB-ITS-PRR-97-48

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California. This report does not constitute a standard, specification, or regulation.

Report for MOU 287

November 1997

ISSN 1055-1425

SmartPath Regulation Layer Implementation

A User's Guide

SmartPath Regulation Layer Implementation
A User's Guide[*]

Jason Carbaugh, Luis Alvarez, Pin-Yen Chen and Roberto Horowitz

Department of Mechanical Engineering
University of California
Berkeley, CA 94720
{carbaugh, alvar, pychen, horowitz@me.berkeley.edu}

November, 1997

[*] Work supported by the PATH program, Institute of Transportation Studies, University of California, Berkeley under Cal Trans grants MOU-287.

ABSTRACT

The implementation on SmartPath[1] of the regulation layer maneuvers for the hierarchical architecture in [3] is described. The first part of this report explains with some detail the structure of SmartPath in order to facilitate modifications or additions to this maneuvers. The second part of the report is devoted to understand the implementation of the existing set of maneuvers; the explanations are tightly connected with theoretical work developed in PATH under MOU-135. The third part focus on modifications or additions to existing maneuvers. One example of such a modification is presented in the fourth part, where the design of the splinning process for the current join maneuver is changed. Finally, sample code for the join maneuver and some programming tools is included.

KEYWORDS

Automated highway systems (AHS), hierarchical AHS architectures, regulation layer maneuvers, SmartPath, feedback-based safe maneuvers.

ACKNOWLEDGEMENTS

This work was executed under grant MOU-238 of the Institute of Transportation Studies of the University of California at Berkeley and is part of a larger effort to develop control systems for AHS in normal and degraded mode of operation. Besides the authors, this group includes Prof. Shankar Sastry, Prof. Pravin Varaiya, Dr. Datta Godbole, Dr. John Lygeros, Dr. Raja Sengupta, Tony Lindsey, John Haddon and other graduate students. Their input to this work is gratefully acknowledged. The authors wish to thank specially Farokh Eskafi and Delnaz Khorramabadi for their help in all the issues related with SmartPath.

EXECUTIVE SUMMARY

This report describes the implementation on SmartPath[1] of regulation layer maneuvers for the hierarchical architecture in [3]. The aim is two fold. On the one hand this report contains information that is helpful to SmartPath users interested in the simulation of automated vehicle control laws in the SmartPath environment. On the other, this report provides an overview on the set of maneuvers currently implemented.

The report is divided in four parts. The first part explains the structure of SmartPath with focus on what is necessary to know in order to facilitate modifications or additions to these regulation layer maneuvers. The second part of the report is devoted to describe the set of regulation layer maneuvers currently implemented in SmartPath: join, split, gentle stop, crash stop and leader. SmartPath simulation results are included. The description of these maneuvers is tightly connected with theoretical work developed in PATH under MOU-135 [2,6]. The third part focus on the modification of the existing maneuvers or additions of new ones. Skeletons for function definitions and a function call tree are included. Some programming tools helpful to test the code prior to its implementation on SmartPath are also described. One example of such a modification is presented in the fourth part, where a new design of the splinning process for the current join maneuver is presented. Finally, the code for the programming tools and some sample code for the join maneuver implementation in SmartPath are included in the appendices. Some of the code presented in this report is currently being ported to SHIFT [8].

Table of Contents

Abstract 1
Keywords 1
Acknowledgements 1
Executive Summary 2

PART I Getting Started

1. Introduction 6
1.1. Scope 6
1.2. AHS Control Hierarchy 6
1.2.1. Physical Layer 7
1.2.2. Regulation Layer 7
1.2.3. Coordination Layer 7
1.2.4. Link Layer 8
1.2.5. Network Layer 8
2. SmartPath Directory Structure: *Where it lives* 9
3. SmartPath Function Calls: *How it Works* 10
3.1. Regulation Layer Framework 10
3.2. Maneuver Module Structure 11
4. SmartPath Execution: *What it Takes* 13
4.1. Input Files 13
4.2. Making and Running 13
4.3. Output Files 13

PART II Understanding Existing Maneuvers

5. The Regulation Layer: *Maneuver Development* 14
5.1. Theoretical Basis 14
5.2. The Safety Boundary 14
5.3. Maneuver Descriptions 15
6. The Regulation Layer: *Maneuver Implementation* 15
6.1. Boundary Curves 15
6.2. Controller Regions 16
6.3. Trajectory Zones 18
6.3.1. All Maneuvers 18
6.3.2. Join 18
6.3.3. Split 18
6.3.4. Gentle Stop/Crash Stop 19
6.3.5. Leader 19
6.4. Maneuver Performance Results and Open Issues 21

PART III Modifying and Creating Manuevers

7. The Regulation Layer: *Maneuver Code Structure* 34
7.1. Function Definitions 34
7.2. Function Call Tree 35
8. Tools for Maneuver Testing.. 35
8.1. Tool Overview.. 35
8.2. twocar.exe 36
8.3. looklstate.m, looktstate.m 36
8.4. sortcar.m... 36

PART IV Join Maneuver Spline Improvements

9. Motivation 37
10. Derivation of Join Maneuver Desired Trajectory 38
10.1. Definition of Terms: 38
10.2. Desired Trajectory in SPLINE3 Zone... 39
10.3. Desired Trajectory in DECEL Zone.. 40
10.4. Desired Trajectory in SPLINE2 Zone... 40
11. Results 43

Appendices

A: Derivation of Desired Trajectory for Leader law 46
B: Parameter Values Used for this Report... 48
C: Source Code: twocar.c 49
D: Sample Maneuver Module Code: regmerge.c 54

List of Figures

1: AHS Hierarchy and Information Flow
2: Function Call Tree for Execution of a Regulation Layer Maneuver
3: General Maneuver Controller Regions
4a: Desired Trajectory Zones: Join Maneuver
4b: Desired Trajectory Zones: Split Maneuver
5: Desired Trajectory Zones: Leader Maneuver
6a: Join Maneuver Boundary Curves
6b: Join Trajectory Tracking
7: Join State History
8a: Split Maneuver Boundary Curves
8b: Split Trajectory Tracking
9: Split State History
10a: Gentle Stop Maneuver Boundary Curves
10b: Gentle Stop Trajectory Tracking
11: Gentle Stop State History
12a: Crash Stop Maneuver Boundary Curves
12b: Crash Stop Trajectory Tracking
13: Crash Stop State History
14a: Leader Maneuver Boundary Curves: Starting Near
14b: Leader Trajectory Tracking: Starting Near
15: Leader State History: Starting Near
16a: Leader Maneuver Boundary Curves: Starting Far
16b: Leader Trajectory Tracking: Starting Far
17: Leader State History: Starting Far
18: Join Maneuver Splining Method Comparison
19a: Improved Join Maneuver Boundary Curves
19b: Improved Join Trajectory Tracking
20: Improved Join State History

PART I

Getting Started

1. Introduction

1.1. Scope

The user's guide in Parts I through III of this report is meant for the uninitiated user of SmartPath[1] who wishes to simulate automated vehicle control laws for testing and validation. The reader will of course need to spend some time getting familiar with the history and development of the hierarchical AHS[2] controller construct that is simulated in the SmartPath architecture [1]. Once that has been accomplished, however, there is a learning curve associated with running and interpreting the results of the simulator. This guide serves to provide some details about this process so that the user can spend more time focused on the topic of study, and less in trying to understand the simulator architecture. With this goal in mind, the scope of this document will be limited to what is known in the AHS architecture as the Regulation Layer. No treatment aside from interfacing details will be given to the large portion of SmartPath which is outside the realm of the Regulation Layer.

Another function of the user's guide is to explain the methods used in implementing the Regulation Layer controllers developed in [2]. It serves as documentation for the resulting source code, and will assist the programmer/designer by providing a reasonable roadmap to facilitate modifications. Lastly, some software development tools are described which assist in the transition from maneuver design to the SmartPath platform implementation.

Part IV of this report details a new method for generating smooth splines between the different portions of the desired join maneuver trajectory. This method results in join maneuvers that are not only safe but comfortable when the front vehicle's behavior is favorable. The derivation of the splines and simulation results are shown.

1.2. AHS Control Hierarchy

The AHS architecture mentioned above divides the task of controlling all vehicles on an automated highway into a hierarchical structure. Each controller operates at a more macroscopic scale than the one below it, and thus abstracts the highway to a higher level. Although much documentation about the development of this hierarchy exists [3,4] a brief overview of each layer is given here to provide the reader with a framework within which to think about the modules which will be discussed in detail as we read on.

[1] SmartPath is an automated highway simulation developed by Farokh Eskafi. A copy can be downloaded from: http://www.path.berkeley.edu/~delnaz/SmartPath. PATH stands for Partners for Advanced Transit and Highways

[2] AHS: Automated Highway System

Five *layers* make up the AHS information flow structure: Physical, Regulation, Coordination, Link, and Network. The first three: Physical, Regulation, and Coordination; are considered to be onboard each individual vehicle. Thus, in an AHS, there are as many of each of these layers as there are vehicles on the highway. The Link and Network Layers consider aggregate traffic information, and thus are associated with roadside intelligence and control. Figure 1 is a diagram of the information flow between layers [5].

1.2.1. Physical Layer

The most microscopic layer of simulation is the Physical Layer. It represents the plant of the AHS control system. Thus, the simulation of this layer incorporates dynamic models of automated vehicles. In an actual AHS, the control input could be something like voltages to a servo motors that position a vehicle's throttle, steering, and other actuators. The output would be the vehicle trajectory and the signals from each sensor on the vehicle. In the SmartPath simulator, the longitudinal control input is the desired jerk (rate of change of acceleration) of the vehicle. During this development, no dynamic model was used for lateral control. The vehicle is modeled as a linear third order system for longitudinal control. The vehicle trajectory is stored, and the vehicles' sensor signals are simulated based on the time history of the behavior of all the vehicles in the simulation.

1.2.2. Regulation Layer

The Regulation Layer simulates the vehicle borne longitudinal and lateral control systems. The input to the Regulation Layer is a maneuver initiation command from the Coordination Layer. Based upon the maneuver command and the interpretation of the environment by the sensors, the Regulation Layer controller calculates the desired control signal to be sent to the Physical Layer. The Regulation Layer also monitors the status of the maneuver. It flags the Coordination Layer when the maneuver is complete, or when the maneuver must be aborted. Since the Regulation Layer is responsible for controlling the details of vehicle trajectories, it must also be responsible for the vehicle's safety. Certainly, the other layers will and should assist in assuring vehicle safety since, intuitively, the presence of more information about the environment facilitates better planning of a safe vehicle trajectory in that environment. However, under *degraded modes* [6] of operation, this extra information may not be available, and the Regulation Layer must still be able to guarantee safety, though perhaps at the cost of reduced speed and throughput.

1.2.3. Coordination Layer

The Coordination Layer handles communication between nearby vehicles. It is responsible for message passing and managing conflicts between desired behaviors of vehicles maneuvering close to one another. These decisions and negotiations are arbitrated based on the current maneuver in which each vehicle is involved, and certain desired local flow characteristics as communicated by the roadside Link Layer.

1.2.4. Link Layer

The Link Layer manages the macroscopic flow for a section (link) of highway. A link is generally considered to be on the order of 1 mile in length [3]. The Link Layer controller is responsible for avoiding congestion, and changing traffic patterns to recover from a system failure. It sets local targets for platoon size, highway speed and number of vehicles changing lanes based on its assessment of the success of the Coordination Layer in maintaining flow, getting vehicles to their desired exit ramps, etc., and on the regional traffic advisories and suggested routes communicated by the Network Layer.

1.2.5. Network Layer

The highest abstraction of the AHS system is the Network Layer. A Network Layer controller may oversee a number of AHS highways. In function, its purpose is similar to traffic advisory radio messages and signs which announce congestion due to traffic incidents and other delays. It helps to proportion vehicles among the AHS highways, both getting vehicles to their destinations and keeping any one AHS highway from being disproportionately congested. In order to combat congestion, the Network Layer also controls the number of vehicles entering and exiting at particular points on the automated highway.

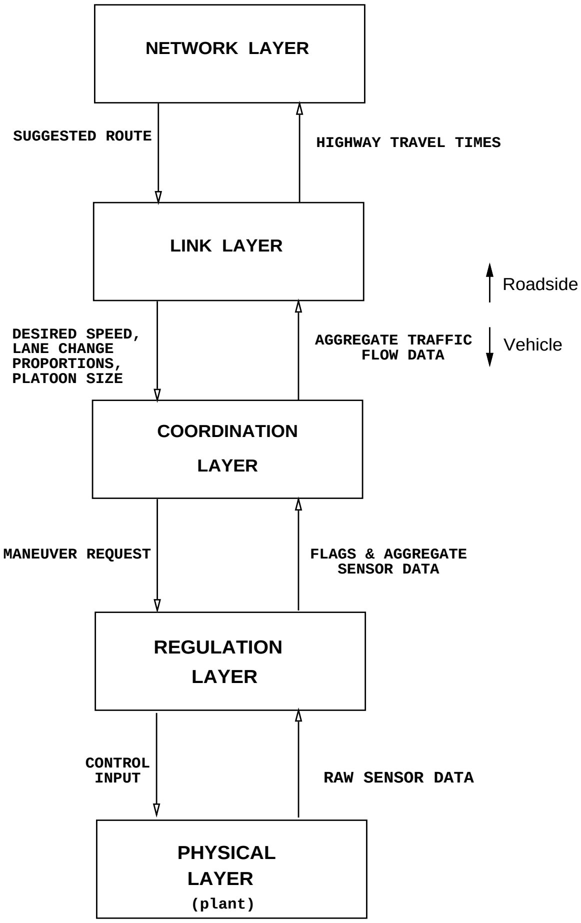


Figure 1

AHS Hierarchy and Information Flow

2. SmartPath Directory Structure: *Where it lives*

With this hierarchical framework in mind, we now proceed to understand how this scheme is reflected in the SmartPath simulator, and more specifically, how to use this understanding to implement Regulation Layer designs. Because there are many modules full of many lines of code in SmartPath (about 30K lines devoted to AHS architecture, plus underlying event manager software), we begin by pointing the reader to the important directories and files that will be involved in the design process. Then once the Regulation Layer has been located, we continue to zoom in by looking at how the different modules and functions work together. Finally, a brief description of the process of running SmartPath is given. With this introduction, one should begin to feel comfortable developing new Regulation Layer material.

The directory structure of SmartPath is as follows. The parent directory of this whole tree may be something like Sm_Release or sm_devel, depending on the name chosen by the user who downloaded it. The entries in bold face are the directories and files that may be used or modified when one inserts and tests a new Regulation Layer control law.

```
/Bin
/Data
      [casename].cars   [casename].config [casename].error
      [casename].state
/Libraries
/Includes
/Src
      /Communication
      /Highway
      /Link
      /Sensor
      /Coordination
      /Hwgadgets
      /Network
      /Simulation
      /Geom
      /Io
      /Regulation
            Makefile          housekeep.c       kmer.c
            regAutoAL.c       regAutoML.c       regAutoTL.c
            regMan.c          regapply.c        regchange.c
            regextern.c       regfollow.c       reginit.c
            reglateral.c      reglead.c         regmerge.c
            regsim.c          regsplit.c        rgSave.c
```

`/Data` is where all of the simulation input and output files reside. The input files are the specifications of the vehicles, highway, and vehicle/highway geometry and configurations. See [7] for details on the format of these files. In general the files may be described as follows:

Inputs

[casename].cars: Specifies the vehicle type for each vehicle in the simulation.

[casename]`.config:` Specifies the size of each vehicle type, the highway topology, details of vehicle generation during the simulation, and simulation time parameters.

Outputs

[casename]`.error:` Reports any errors detected during the simulation.

[casename]`.state:` Records the state of all vehicles throughout the entire simulation. This file is used for analysis of controller performance and can even be fed into post processors to render and animate the simulation scenario.

`/Src` is the directory which contains all the source code. One can see that there is a subdirectory for each of the hierarchical layers described in Part 1 - except for the Physical Layer. In SmartPath, the Physical Layer is actually inside the Regulation Layer. The file `kmer.c` is a numerical integrator that calculates the simulated vehicle response to the control signal from the Regulation Layer.

`/Src/Regulation` contains all of the Regulation Layer framework and maneuver modules. This is where most of the work will be done in implementing control laws.

`/Src/Simulation` contains the code which oversees the execution of the simulation. This directory also contains the executable that launches SmartPath.

3. SmartPath Function Calls: *How it Works*

3.1. Regulation Layer Framework

Figure 2 shows a scheme of how the Regulation Layer works using a Function Call Tree. Each large box represents one module (file) within the `/Regulation` directory. The small boxes with rounded corners contain the file name of each module, and the square-cornered boxes below them are the names of the functions which are defined in that module. An arrow pointing from one function A to another function B signifies that A calls B.

Reading the diagram from left to right, the Coordination Layer sends a maneuver request to the Regulation Layer by calling the function `Rg_AutoRegulationAL` with the appropriate arguments. This sets the maneuver into motion as `Rg_AutoRegulationALloop` runs every simulation time step (SmartPath currently records all the vehicle states at 0.1 second intervals) until the maneuver is completed. Notice that the Coordination Layer can send its commands to any one of the four modules in the first column. This depends on the type and location of the vehicle. In descending order, these functions refer to:

An automated vehicle in an Automated Lane
An automated vehicle in a Manual Lane
An automated vehicle in a Transition Lane
A manual vehicle in a Manual Lane

The `Rg_AutoRegulationALloop` has one further responsibility. It checks if the vehicle state is safe, and decides to allow or disallow each requested maneuver to begin.

It also continuously monitors the state of the maneuver to see if it is completed, or if it must be aborted. This information is then relayed up to the Coordination Layer.

In the second column, we see the `regapply.c` module. The function `ApplyControl` calls the appropriate lateral and longitudinal controller functions from within the respective maneuver modules. As to timing, the controller functions are called once every simulation time step (currently 0.1 sec), which is assumed to be the same as the interval between sensor readings.

Not shown in the diagram is a construct that builds many bridges between each module without explicit function calls. It is a database called the State Table. It contains many variables which keep track of the state of the vehicle, its sensors, its communication, and more. Each of the state variables are accessed from the code as:
`rgStateTable[cid]->variable_name`

3.2. Maneuver Module Structure

The Regulation Layer controller for each maneuver is contained in a file named: `reg[Maneuver_Name].c`. Some examples are: `regmerge.c`, `regsplit.c`, `reglead.c`. The functions which are called during the execution of a maneuver are of the form shown below (with examples):

```
Init[Maneuver_Name]              InitSplit         InitLeader
SafeTo[Maneuver_Name]            SafeToSplit       SafeToLeader
SafeToA[Manuever_Name]           SafeToASplit      SafeToALeader
[Action][Maneuver_Name]          DecelToSplit      BeLeader
Complete[Maneuver_Name]          CompleteSplit     CompleteLeader
```

`Init[Maneuver_Name]` is an initialization function which must be called before executing the maneuver. It initializes all of the necessary variables in the vehicle state table.

`SafeTo[Maneuver_Name]` checks if the current state of the maneuvering vehicle is safe. Currently, this function only returns a verdict of unsafe if an impact is imminent at a relative velocity that is higher than the allowable impact velocity. The purpose of this function is to act as a permission gateway to the execution of the maneuver. If the function returns a ’0’, then the maneuver will be disallowed.

`SafeToA[Maneuver_Name]` checks if it is safe to immediately abort the current maneuver. Following the logic of the maneuver designs described in Part II, if the maneuver has begun, then the state is guaranteed to be safe. Therefore this function simply returns a ’1’ signifying ’Safe to Abort’.

`[Action][Maneuver_Name]` is the heart of the module. It calculates the control signal that will feed into the vehicle’s longitudinal actuator. For now, this value is the jerk of the maneuvering vehicle. The jerk is assumed to have no dynamics, and thus can be set arbitrarily within the jerk limits of the vehicle.

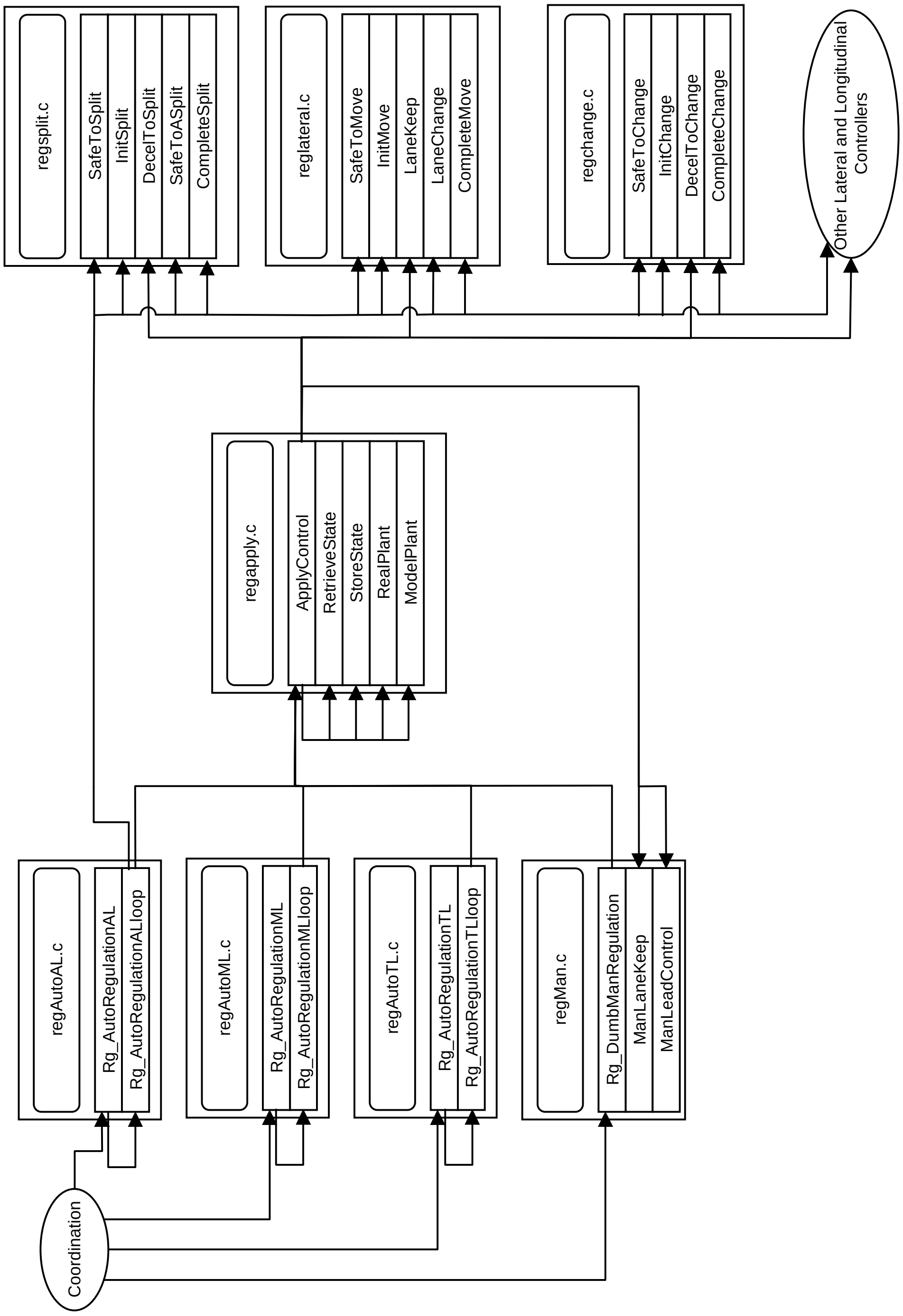


Figure 2

Function Call Tree for Execution of a Regulation Layer Maneuver

`Complete`*`[Maneuver_Name]`* monitors the maneuvering vehicle's state in relation to a target final state region. This function determines if the maneuver is Complete, Not Complete, Unable to Complete (e.g. out of sensor range of front vehicle).

4. SmartPath Execution: ***What it Takes***

4.1. Input Files

In order to run SmartPath, one must first verify that the input configuration is as desired. As stated above, this mainly involves two files: *`[casename]`*`.cars` and *`[casename]`*`.config`. In fact, since most information is in the latter, it is usually the only one that needs verification. See [7] for more details about the format of SmartPath's input and output files.

4.2. Making and Running

Once all of the input files are in order, the source code is ready to be compiled. This is accomplished by executing the Makefile in the appropriate directories. Look at the extensions of the Makefiles in each directory, and use the appropriate Makefile for the platform upon which you are running SmartPath. When doing Regulation Layer development, compilation is usually necessary only in the /Regulation and /Simulation directories. If changes are made in other directories, it will be necessary to remake them. Changes in some global header files will require a remake in all directories. To execute SmartPath with Regulation Layer changes, execute the following commands:

```
cd /Src/Regulation
make
cd /Src/Simulation
make

Dosim [casename]
```

where *`[casename]`* matches the name of the input file: *`[casename]`*`.config` in `/Data`.

4.3. Output Files

If any errors are detected by SmartPath during run-time, they are recorded in *`[casename]`*`.error`. Otherwise, look for the simulation results in *`[casename]`*`.state`. This is an ascii file with columns of numbers of the format:

{Time [s]}{Car ID}{Car ID of Platoon Leader}{Velocity}{Acceleration}...
{Highway Segment #}{X Position [m]}{Y Position [m]}{Z Position [m]}...
{Yaw Angle}{Pitch Angle}{Roll Angle}{Tire Angle}

PART II

Understanding Existing Maneuvers

5. The Regulation Layer: *Maneuver Development*

5.1. Theoretical Basis

The maneuvers whose implementation is described below were developed directly from [2]. The general philosophy of this design is to track a minimum time trajectory for a desired maneuver. A *safety boundary* is defined in the state space that describes the relative motion of pair of vehicles involved in that maneuver (the relative distance, relative velocity and the velocity of the lead car are chosen in [2]). This boundary is such that if the front vehicle of the pair suddenly brakes at its maximum capability, then the back vehicle is able to brake to avoid a collision at an unsafe relative velocity. If the rear vehicle strays from its desired trajectory (this depends on the controller performance) such that its state is close to the safety boundary, then the vehicle brakes at its maximum capability in order to drive itself into a safer region of the state space. It is proven that, under this philosophy, maneuvers so designed will always be safe under normal operating conditions. The reader is advised to acquire this paper before reading further, for the terminology used here will be adopted from that paper without a full explanation.

Regulation controllers for some maneuvers have been implemented in SmartPath using the above design methodology. Discussion of implementation issues and documentation of source code are the two major purposes of the sections that follow. The modular structure adopted to represent maneuvers in SmartPath enables new maneuvers to be programmed quickly, as it standardizes concepts common to all maneuvers. For example, all maneuvers must calculate the safety boundary and all maneuvers must have a desired trajectory for the back vehicle to track.

5.2. The Safety Boundary

The concept of a safety boundary is developed in [2], where it is shown that for any initial state outside X_{bound}, an unsafe impact (an impact with a relative velocity above a parameter: v_{Allow}) may be imminent, where X_{bound} is a boundary curve derived in the relative distance/relative velocity state space. This boundary is the absolute safety limit; even if the back vehicle is braking at its maximum capability when it crosses this boundary, an unsafe impact can occur. In order to prevent this, another boundary curve X_{safe} is defined. As soon as a vehicle detects that it is crossing over the boundary X_{safe}, it applies maximum braking. X_{safe} is defined such that the vehicle will be guaranteed safety if it does. The distance between the X_{bound} and X_{safe} boundaries depends on the reaction delay between the front vehicle braking hard and the back vehicle detecting the emergency condition and braking hard.

5.3. Maneuver Descriptions

The maneuvers that have been implemented thus far are:

Join (previously known as **Merge**). A vehicle or platoon joins the tail of another platoon. The formation of platoons increases throughput on the highway because the vehicles follow at very close (intra-platoon) spacings.

Split A vehicle or partial platoon slows to break away from the rest of the platoon. Two types of split exist: Fast Split and Slow Split. In a Slow Split, the splitting platoon decelerates and accelerates such that is ends up one inter-platoon spacing away from the platoon ahead. The Slow Split is essentially the opposite of a **Join**. The Fast Split is simply a logical delineation, to allow the back platoon to immediately begin another maneuver, such as an emergency lane change. All details below will refer to the development of the Slow Split.

Leader A vehicle is a free-agent or the leader of a platoon. In this development, no distinction is made between these two cases. In actuality, the leader of a platoon may have additional constraints placed upon it. For example, when a platoon leader wishes to brake hard, it is conceivable that it would coordinate with the followers within its platoon. They are following at such close spacings that they cannot react in time if their leader brakes hard without warning. An free-agent has an empty set of followers and therefore no need of coordinating.

Gentle Stop A vehicle brakes to a stop with comfortable deceleration. A vehicle may want to slow gently if it experiences a moderate failure, such a loss of cooperative vehicle communications, or perhaps even a blown tire.

Crash Stop A vehicle brakes to a stop with maximum deceleration. This maneuver could be initiated if the vehicle experienced a debilitating failure, such as a failure of its vehicle range/range rate detection sensors. The vehicle would stop as fast as it could to minimize the risk to itself and other vehicles on the highway.

Many other maneuvers exist which give an automated vehicle a wide range of activities it may perform on the highway. Some are normal mode maneuvers and some are degraded mode maneuvers which are initiated under certain failure conditions. Of these additional maneuvers, some are more difficult to cast in the safety framework because they involve multiple vehicles. The problems arise when one tries to choose a strategy which trades off safety between different vehicles. In any case, some examples of these additional maneuvers are: Lane Change, Platoon Follower, Entry/Exit, Front Dock, Aided Stop, Take Immediate Exit, Back Up. Many of these additional maneuvers have been implemented in SmartPath, though they do not necessarily use the same structure as described in this manual.

6. The Regulation Layer: *Maneuver Implementation*

6.1. Boundary Curves

The equations which describe the safety boundary were developed in [2]. They are solved for the velocity safety limit of the back car. The boundary defined by v_{bound1} covers the case when an impact occurs after the front vehicle has come to a stop. The boundary defined by v_{bound2} covers the case when both vehicles are moving when the collision occurs. For clarity, these equations are rewritten here in terms that match the implemented source code.

$$v_{bound1} = \sqrt{-2a_{min}\Delta x + v_{lead}^2 + \Delta v_{allow}^2} \quad (1)$$

$$v_{bound2} = \Delta v_{allow} + v_{lead} \quad (2)$$

$$v_{bound} = \max(v_{bound1}, v_{bound2}) \quad (3)$$

The boundary defined by v_{safe} is likewise restated below. Upon crossing this boundary, a vehicle will brake hard to assure safety. Again, v_{safe1} covers the case when the front vehicle would be stopped when the collision occurs; v_{safe2} covers the case when the front vehicle would still be moving. An additional term, Δv_{buff} is a factor of safety. The main reason for its inclusion is to account for a possible braking jerk constraint on the back vehicle and to account for controller transients. The boundary equation derivations do not account for jerk constraints on either vehicle.

$$v_{safe1} = -(a_{max} - a_{min})d - \Delta v_{buff} + \ldots$$

$$\sqrt{-2a_{min}\Delta x + v_{lead}^2 + \Delta v_{allow}^2 - a_{min}(a_{max} - a_{min})d^2} \quad (4)$$

$$v_{safe2} = -(a_{max} - a_{min})d - \Delta v_{buff} + \Delta v_{allow} + v_{lead} \quad (5)$$

$$v_{safe} = \max(v_{safe1}, v_{safe2}) \quad (6)$$

A final boundary is calculated which is essentially v_{safe1} evaluated at zero allowable impact velocity. Its use for maintenance of passenger comfort is explained in the next section. It is the boundary which a vehicle must not cross if it wishes to avoid *any* impact with the vehicle in front.

$$v_{NoColl} = -(a_{max} - a_{min})d - \Delta v_{buff} + \sqrt{-2a_{min}\Delta x + v_{lead}^2 - a_{min}(a_{max} - a_{min})d^2} \quad (7)$$

6.2. Controller Regions

The term *region* is used to describe the current state (distance and velocity relative to the vehicle in front) of a vehicle in relation to the boundary curves defined above. These regions are depicted graphically in Figure 3, and can be seen to cover the entire state space (assuming shaded regions on the edges continue to infinity). Four of the seven regions result from the previously discussed boundaries: X_{bound} and X_{safe}:

NORMAL: All states within X_{safe}. A vehicle within this region simply follows whatever control law it happens to be executing at the time.

BRAKE: All states between X_{safe} and X_{bound}. While in this region, a vehicle brakes hard until its state returns to X_{safe}. This is uncomfortable, and should occur as infrequently as possible. Potential causes for entering this region are poor controller performance and antagonistic behavior of the vehicle in front.

UNSAFE: All states outside X_{bound} before the collision. Within this region, the vehicle continues to brake hard, but a collision is imminent if the lead car mantains its maxim braking. This region is unreachable under a properly implemented safe maneuver.

CRASH: All states in which vehicles occupy the same space. This region is unreachable under a properly implemented safe maneuver.

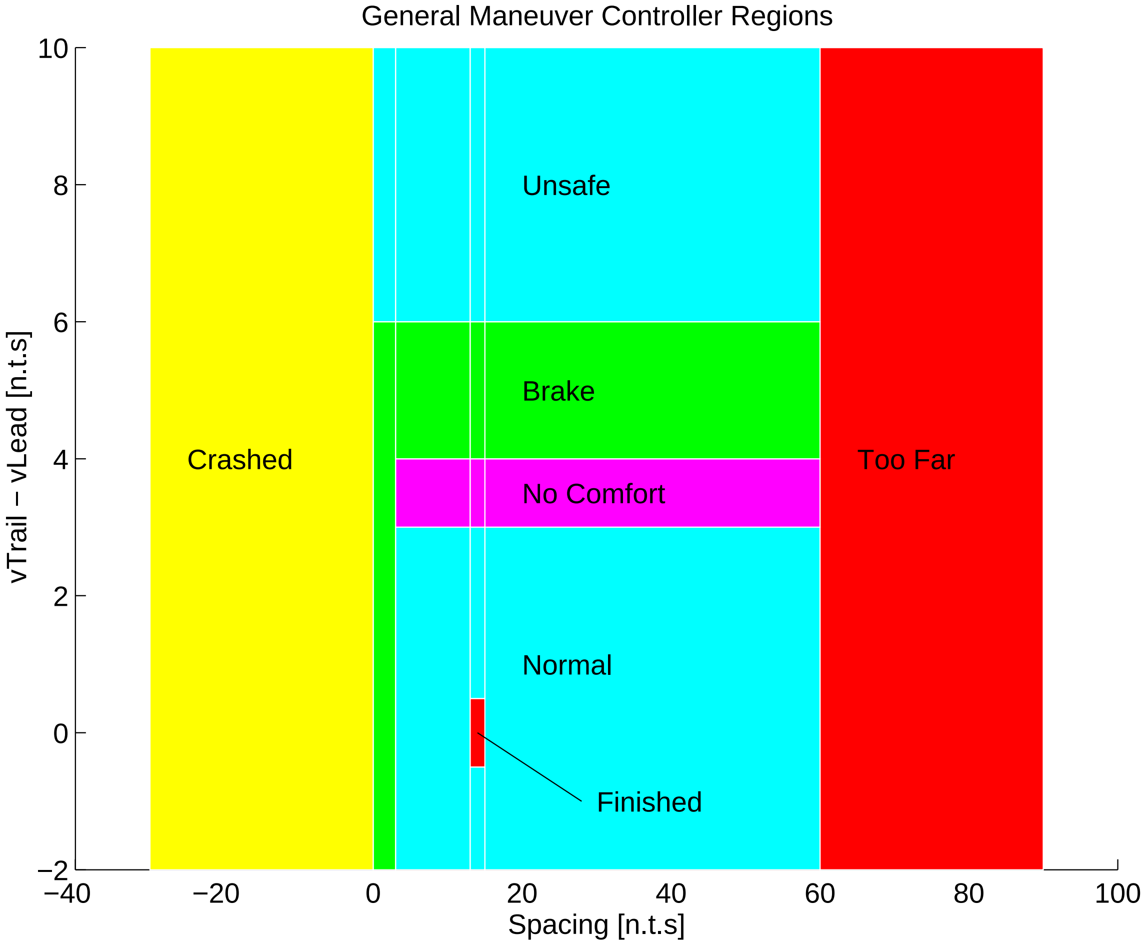


Figure 3

General Maneuver Controller Regions

In order to maintain comfort of the passengers in the vehicle, the acceleration and jerk of the vehicle is normally restricted well within the vehicle capabilities. Thus, while the state of the vehicle is in the NORMAL region, maneuvers should not only be safe, but also comfortable. However, while under comfort constraints, a vehicle may not be able to keep itself away from the X_{safe} boundary, and thus have a tendency to 'bounce' off the boundary by alternately braking hard and drifting back to the boundary. In order to reduce this effect, an additional region 'NOCOMFORT' was added. Another boundary curve was defined by evaluating the boundary of X_{safe} with a zero allowable impact velocity (v_{Allow}). This new boundary is denoted as X_{nocoll} .

NOCOMFORT: All states between X_{nocoll} and X_{safe}. Within this region, the saturation limits on acceleration and jerk are lifted so that the vehicle can remain within X_{safe} although reducing the comfort of the passengers.

The final two regions are defined to determine if and when the maneuver will be complete. Some maneuvers such as Join cannot be completed if a the vehicle in front goes out of sensor range, so this region must be distinguished. Also, some maneuvers accomplish a certain goal which must be determined to be 'in progress' or 'finished'. The completion of a maneuver is defined in terms of a target region in the state space. For example, a Join is finished when the vehicles are one intra-platoon spacing apart, and at zero relative velocity.

TOO_FAR: All states in which the vehicle in front is out of sensor range.

FINISHED: All states within the 'completion target' portion of the state space. This completely depends on the goal of the maneuver, but will be somewhere in the NORMAL or perhaps the TOO_FAR region.

6.3. Trajectory Zones

6.3.1. All Maneuvers

All maneuvers exist for the purpose of moving the vehicle state to a target state space region: either a FINISHED region or a desired operating point. The path by which the state travels to get to this desired region should follow a desired trajectory. This desired trajectory is chosen to reduce or minimize maneuver completion time and to provide a level of passenger comfort while remaining inside the NORMAL region. It does this by defining a velocity setpoint for the vehicle controller at any initial state in the NORMAL region (at this development stage, the velocity setpoint depends only on the current spacing and lead vehicle velocity and not on the current relative velocity). The desired trajectory is, in general, a piecewise continuous function in the spacing/relative speed space. The *zone* in which a vehicle's state lies is determined by the equation which defines the desired trajectory and thus setpoint of the controller.

6.3.2. Join

Figure 4a diagrams the zones of a Join maneuver. In the first and third zones, SAFE1 and SAFE2, the desired trajectory is the boundary of X_{safe}. This is to reduce maneuver completion time. The fifth zone, DECEL, departs from the safety boundary in order to move toward the target completion region. For a Join, this requires a zero relative velocity at one intra-platoon spacing (within some tolerance). The second and fourth zones, SPLINE1 and SPLINE2 are cubic splines between the adjacent trajectory curves. They provide some comfort by avoiding sharp vehicle accelerations in response to a non-smooth desired velocity trajectory.

6.3.3. Split

The desired trajectory for the Split maneuver is simpler than the Join. It requires only two zones: ADVANCE and RETREAT. These two zones simply follow the minimum time trajectory at comfortable accelerations to a desired Split completion distance.

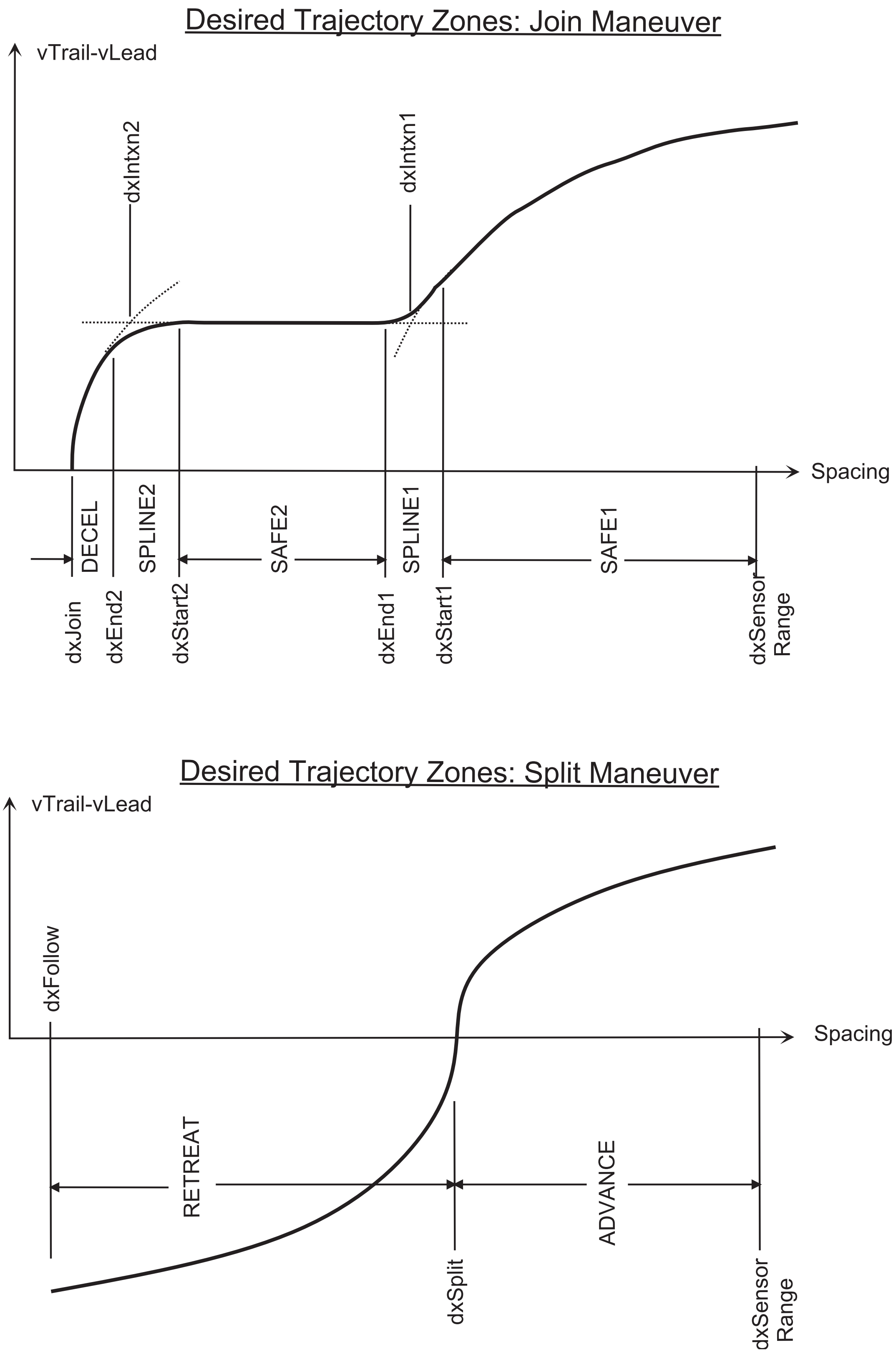
6.3.4. Gentle Stop/Crash Stop

The Stop maneuvers do not use the concept of zones since they are open loop controllers. Instead of tracking a desired velocity, the Stop maneuvers simply decelerate at a constant rate until the vehicle comes to rest. The safety boundary is still monitored, however, and thus the vehicle will still brake hard in response to a breach of that boundary.

6.3.5. Leader

In many ways, the Leader law is similar to the Split law. Both seek to maintain a large spacing between platoons at zero relative velocity. Thus, both include the zones ADVANCE and RETREAT. However, an additional feature was added to the Leader trajectory - sacrificing a small increase in completion time for the sake of passenger comfort. It is a modification which could potentially be made to the other maneuvers as well. It can be seen from looking at the desired trajectory diagrams (Figures 4a-5) that most desired trajectories bring the vehicle state to the target region at a non-zero acceleration. The result is oscillation about the desired equilibrium point as the next maneuver tries to regulate the state around that point. In order to avoid this in the Leader law, the quadratic minimum time curves which cross the target state at non-zero acceleration are replaced by cubic splines which go to zero acceleration at the target state.

The other two boundary conditions that are used to define the splines are the desired velocities at the edges of the NORMAL region. At very close spacings, the desired velocity approaches the minimum speed allowed on the highway under normal conditions: v_{Slow} (or v_{front} if $v_{front} < v_{Slow}$). At spacings which approach the sensor range, the desired velocity approaches the value set by the Link Layer for large scale flow optimization. Thus, if a vehicle is traveling on open road (in the CRUISE zone) traveling at vLink, it will not suffer a sudden change in setpoint if a vehicle comes into sensor range. See Appendix A for derivation of the Leader law desired velocity curves.



Desired Trajectory Zones: Join Maneuver

Desired Trajectory Zones: Split Maneuver

Figure 4

Join and Split Maneuver Desired Trajectory Zones

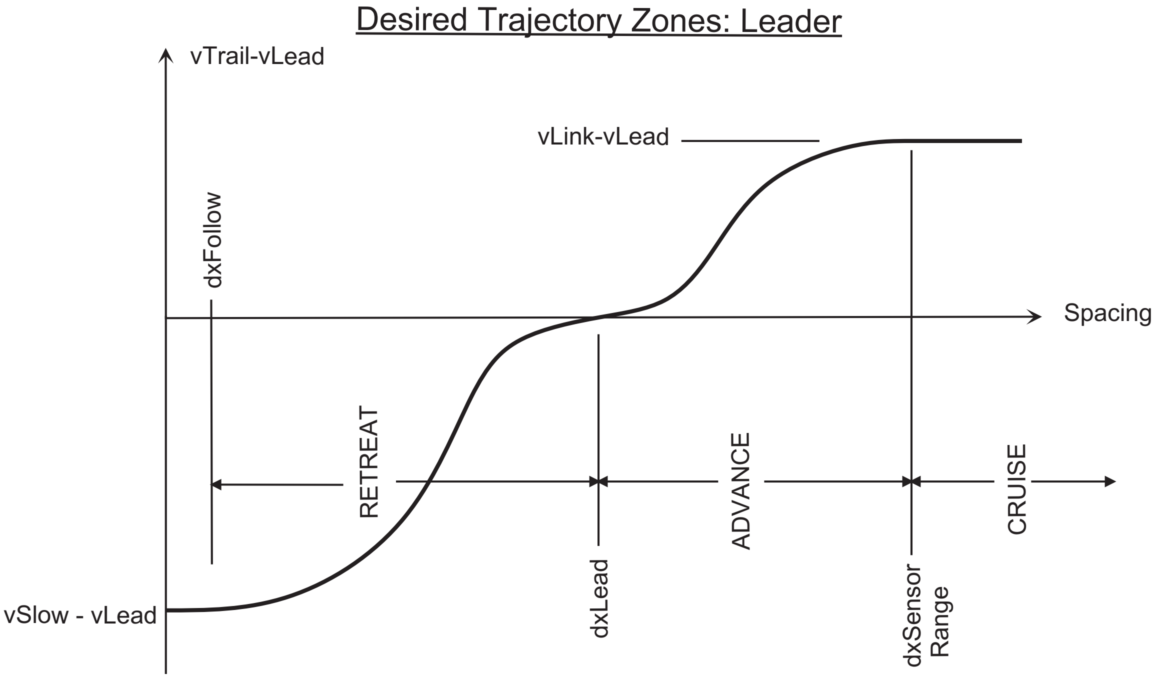


Figure 5

Leader Law Desired Trajectory Zones

6.4. Maneuver Performance Results and Open Issues

Figures 6b,8b,10b,12b,14b,16b show examples of performance for each maneuver for a certain set of gains. Figures 7,9,11,13,15,17 show this performance in terms of the full state of the back vehicle (position, velocity, acceleration, jerk). Note that two sets of results are shown for the Leader law using two different initial spacings. In studying all of the maneuver performance plots, some design issues are worth noting:

- Not observable in the figures is the high sensitivity of performance to gain with the existence of a hard safety boundary. In the Join maneuver, high gains cause the vehicle state to get thrown rapidly away when it hits the safety boundary. Over-correction results in the vehicle state continuously bouncing off the safety boundary. One possible solution would be to track a desired trajectory which is farther within the safety boundary, thus sacrificing a little completion time for lower sensitivity to overshoot.
- In the Join maneuver, the desired trajectory in the DECEL region is overshot even when the controller is tuned aggressively. One may wish to design a more comfortable trajectory as a Join finishes. Currently, the vehicle is slamming on its brakes at the conclusion of any Join, even in normal conditions. This behavior is due to the choice of Δv_{buff} , that is too small for the sampling time used in the SmartPath simulations, and also to the width of the SPLINE2 region . One correction to this problem, based on a reformulation of the SPLINE2 is presented in chapter 9.

- All desired velocity curves are subject to saturation by v_{Fast}, the speed limit of the highway under normal conditions. Currently, there is no splining region to smooth the transition into this 'saturation zone'.

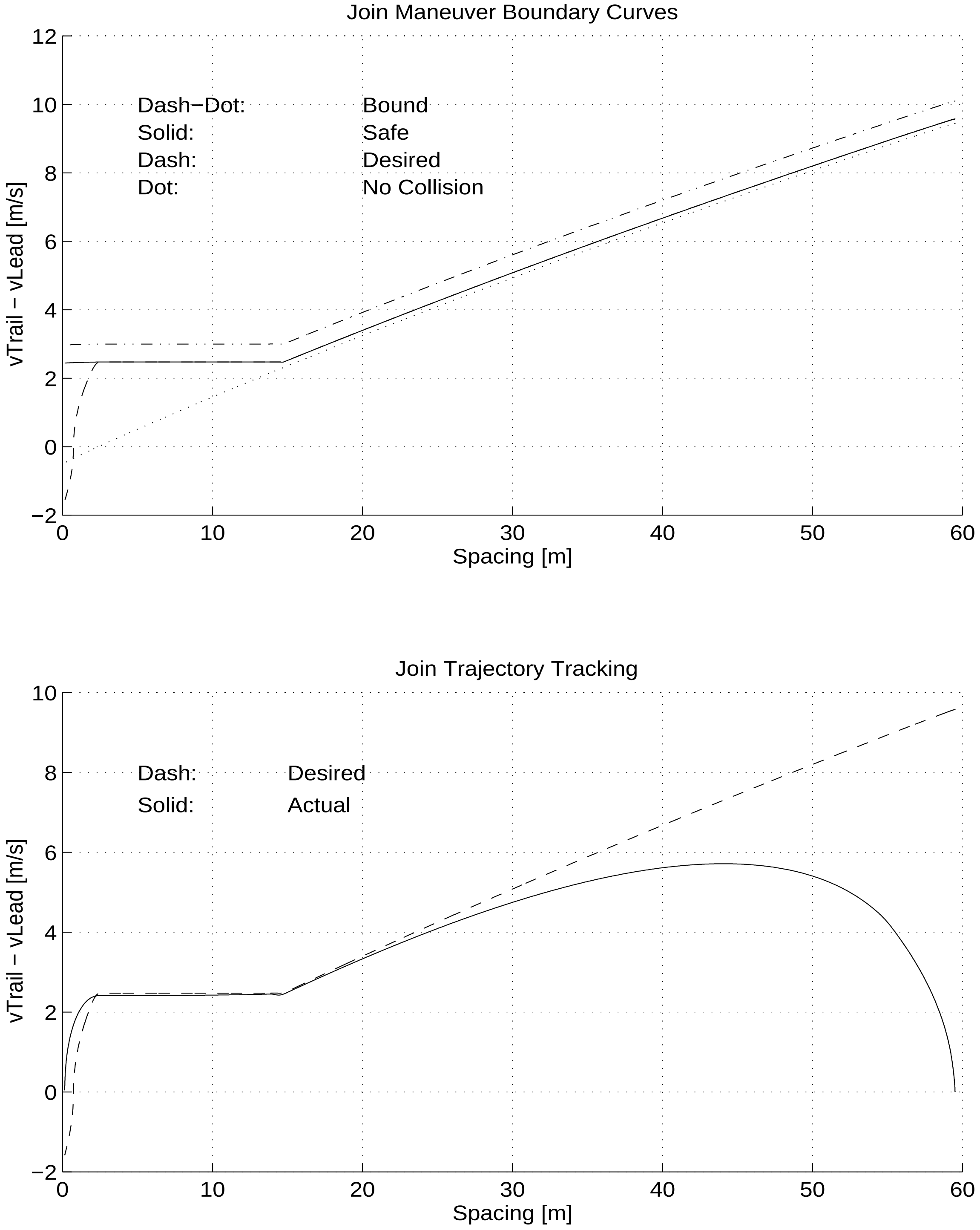


Figure 6

Join Maneuver Safety Boundaries and Performance

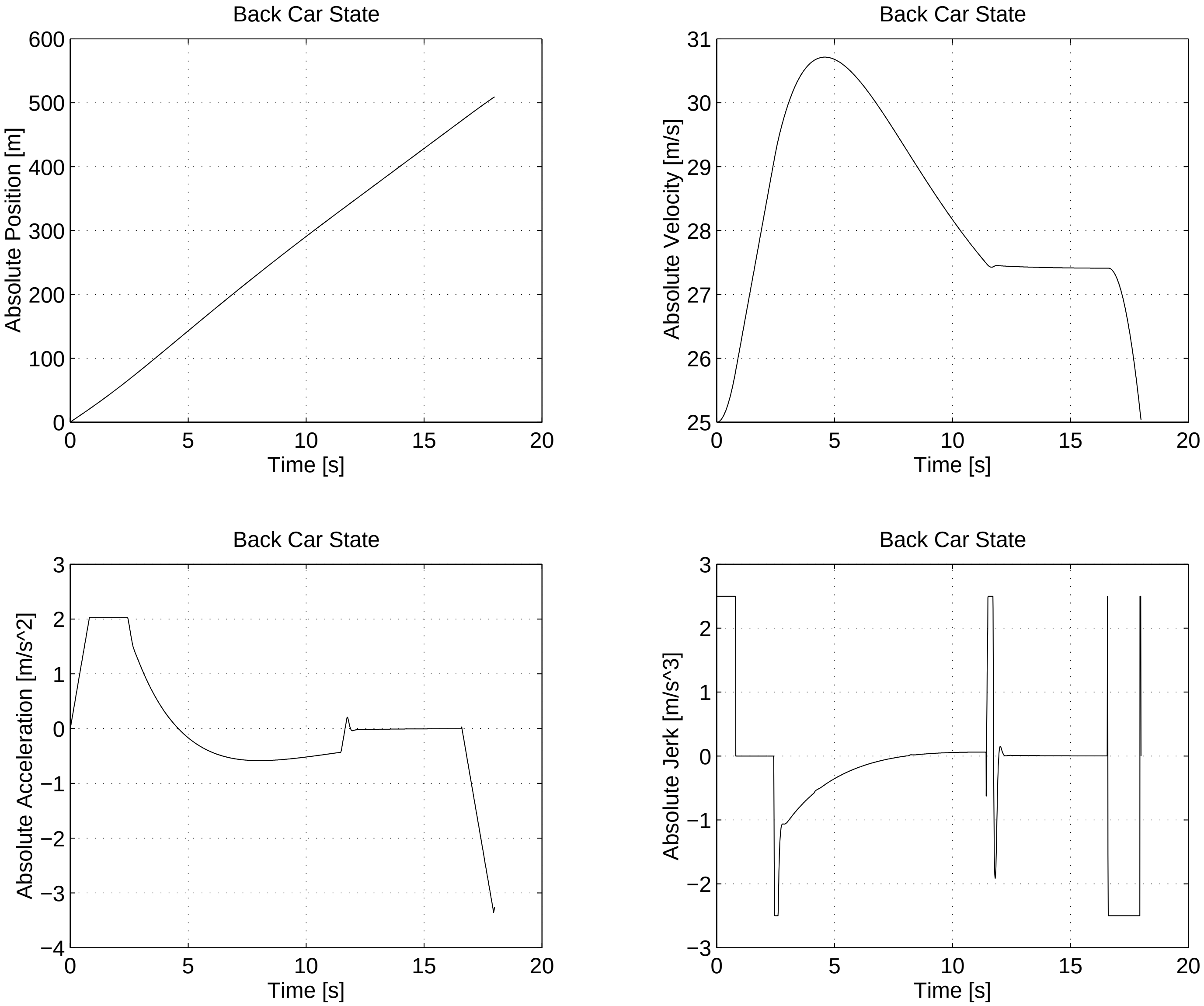


Figure 7

Join Maneuver State History

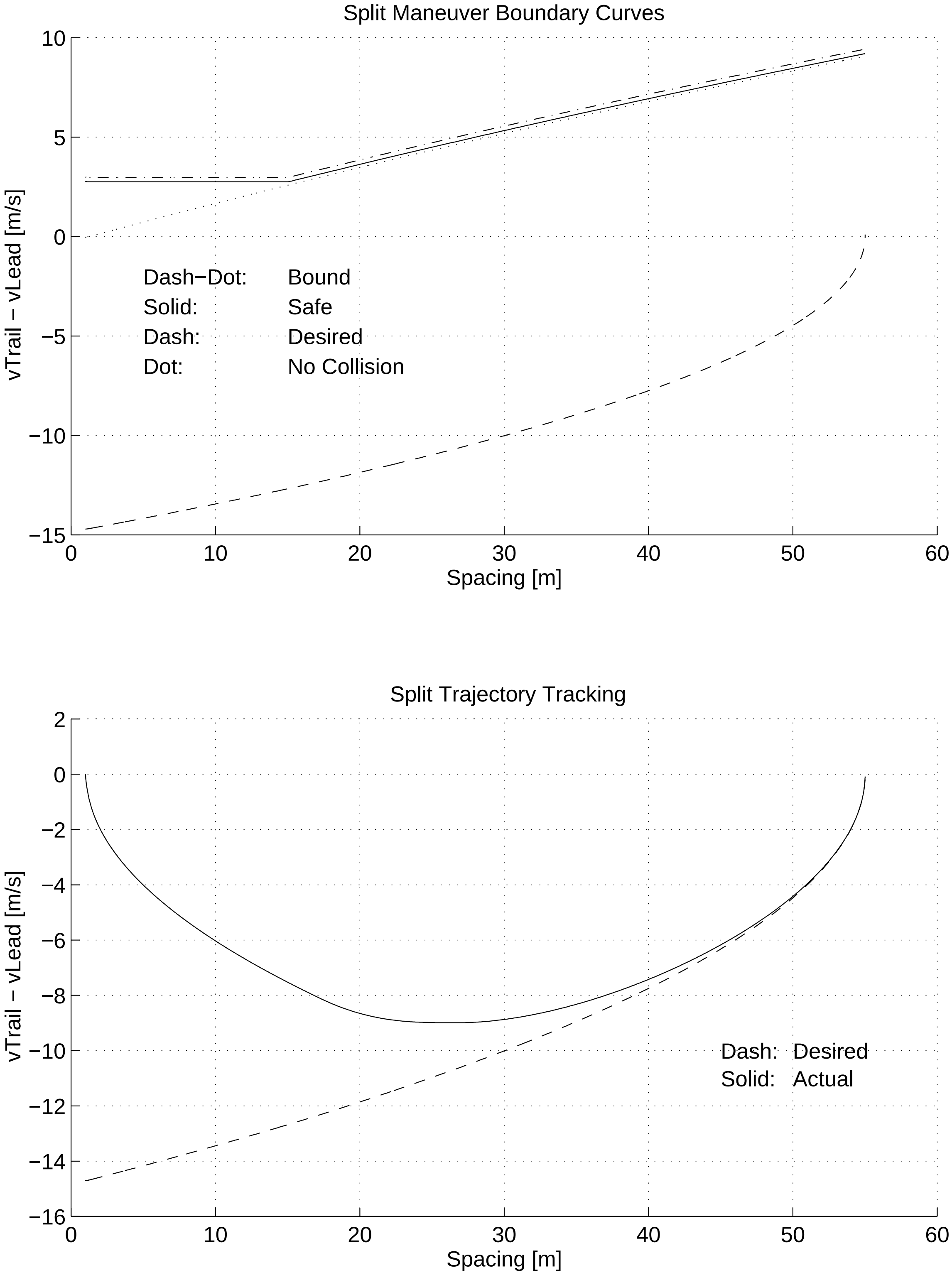


Figure 8

Split Maneuver Safety Boundaries and Performance

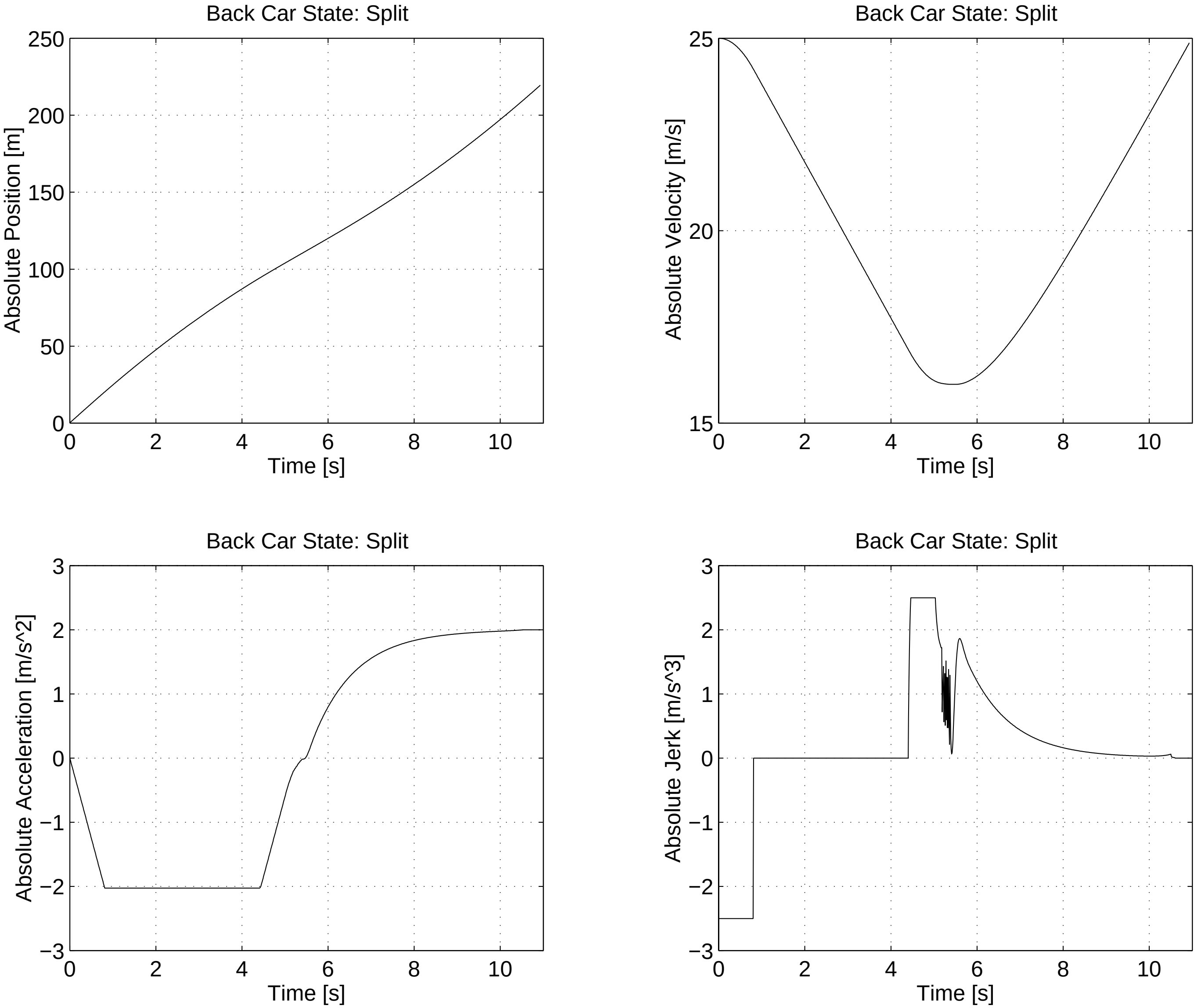
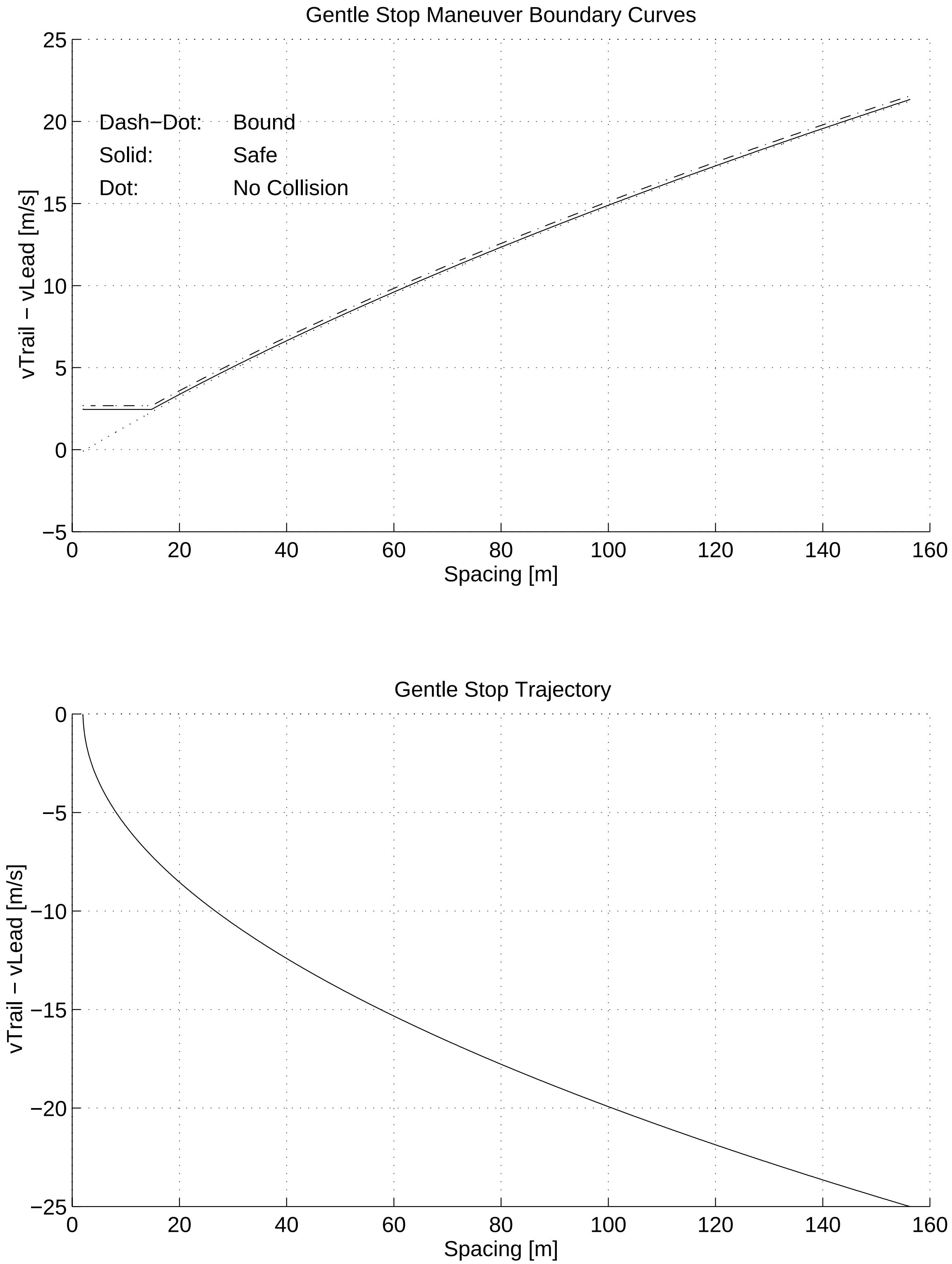


Figure 9

Split Maneuver State History


Gentle Stop Maneuver Boundary Curves

Gentle Stop Trajectory

Figure 10

Gentle Stop Maneuver Safety Boundaries and Performance

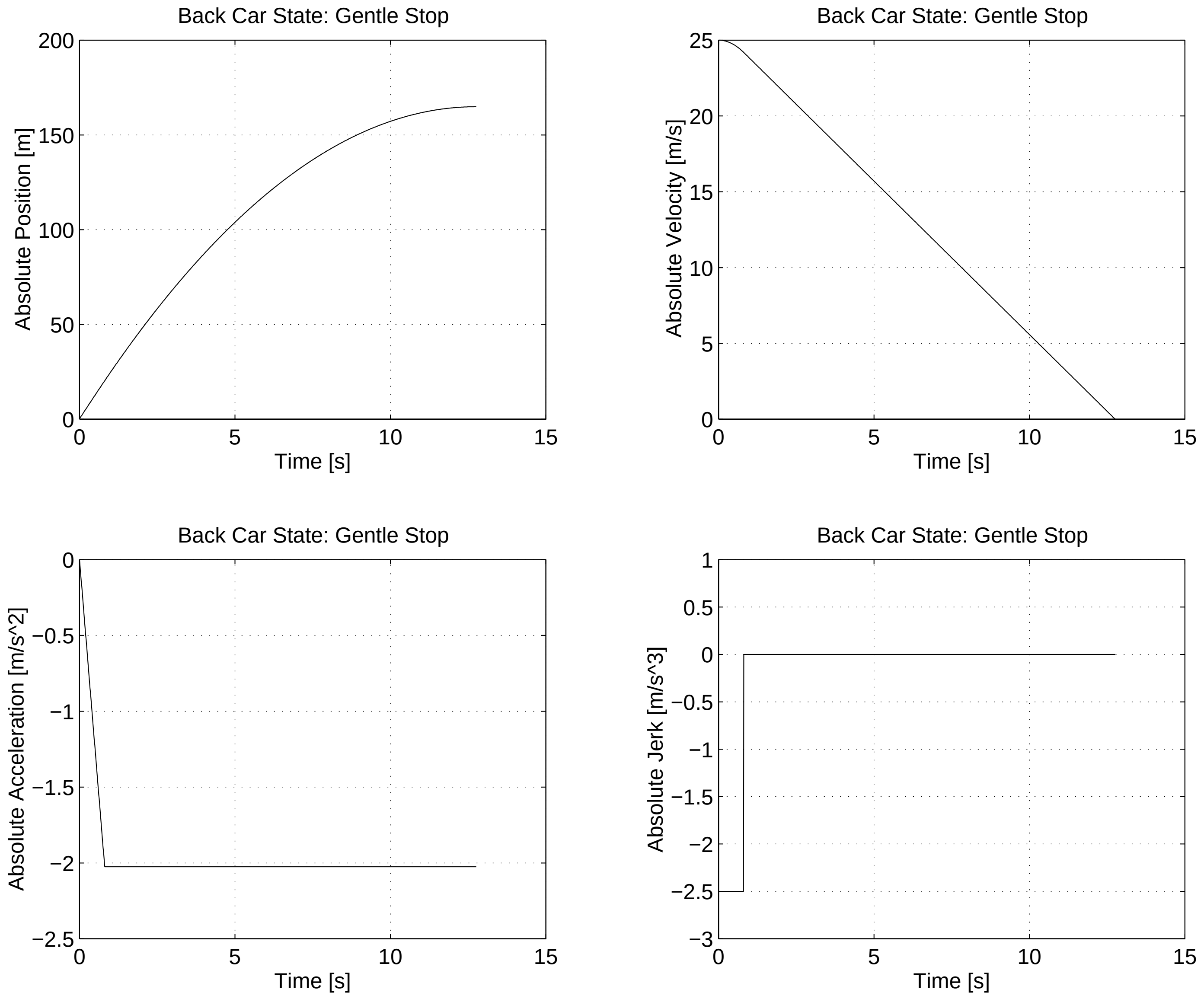


Figure 11

Gentle Stop Maneuver State History

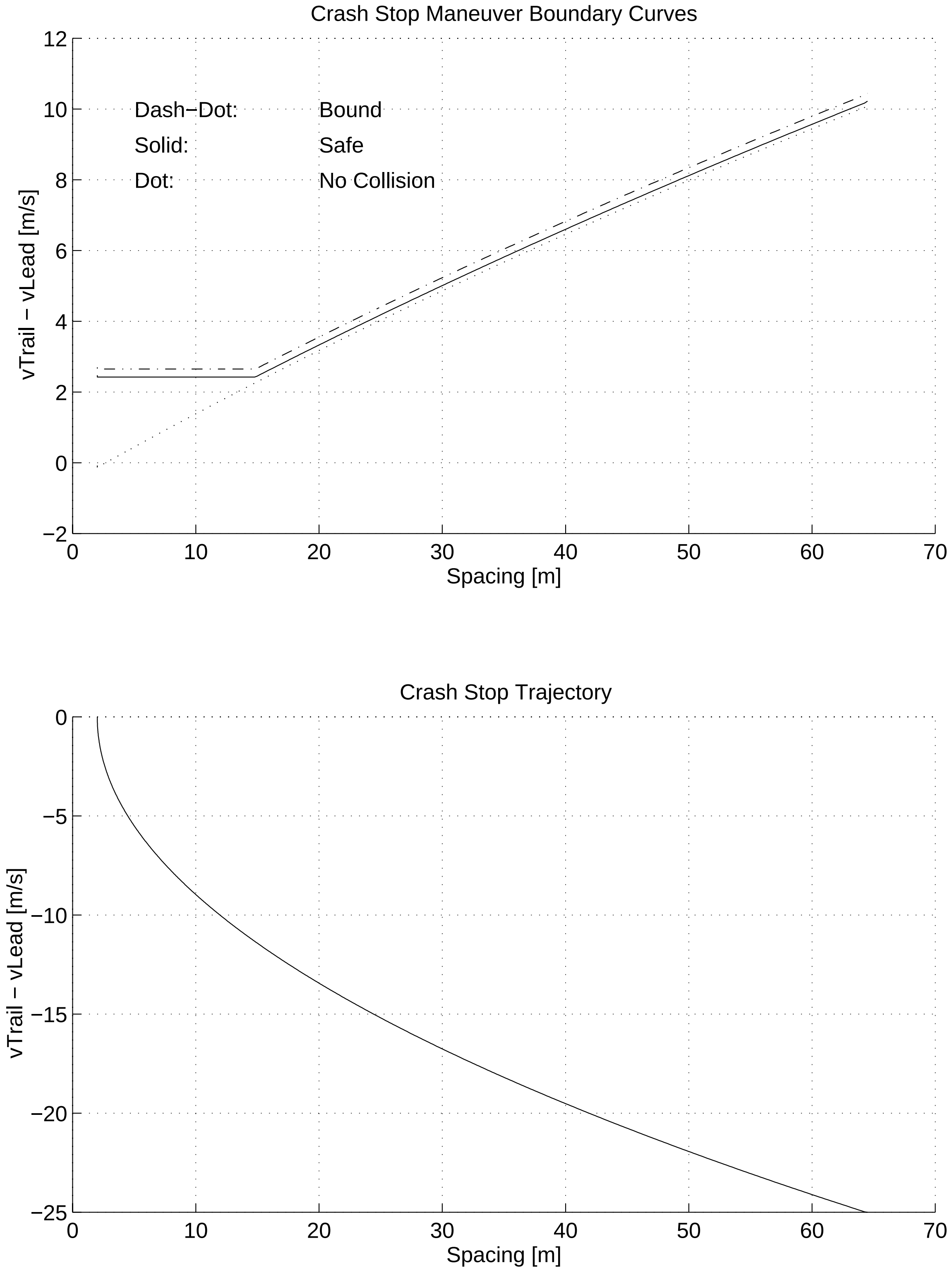


Figure 12

Crash Stop Maneuver Safety Boundaries and Performance

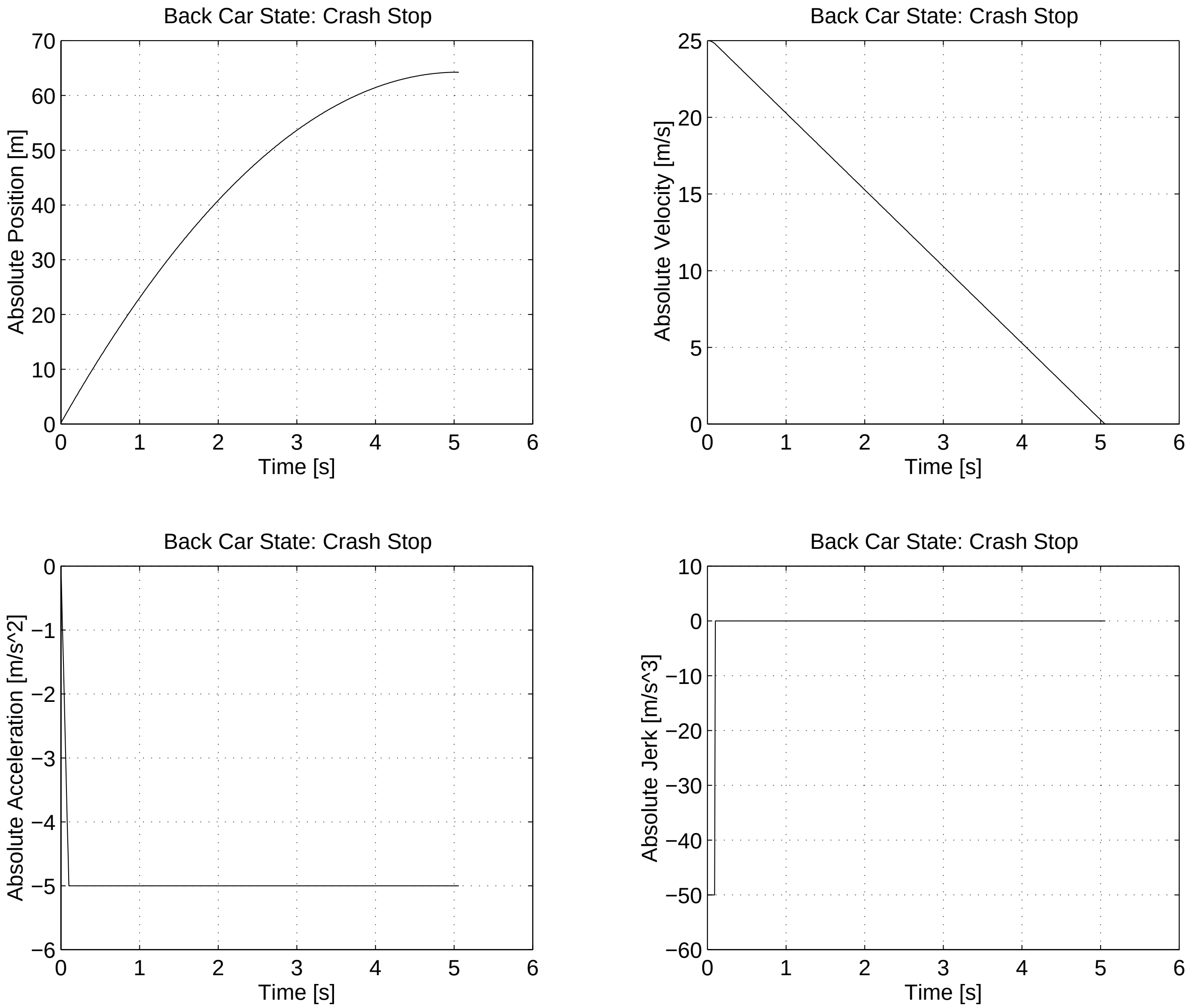


Figure 13

Crash Stop Maneuver State History

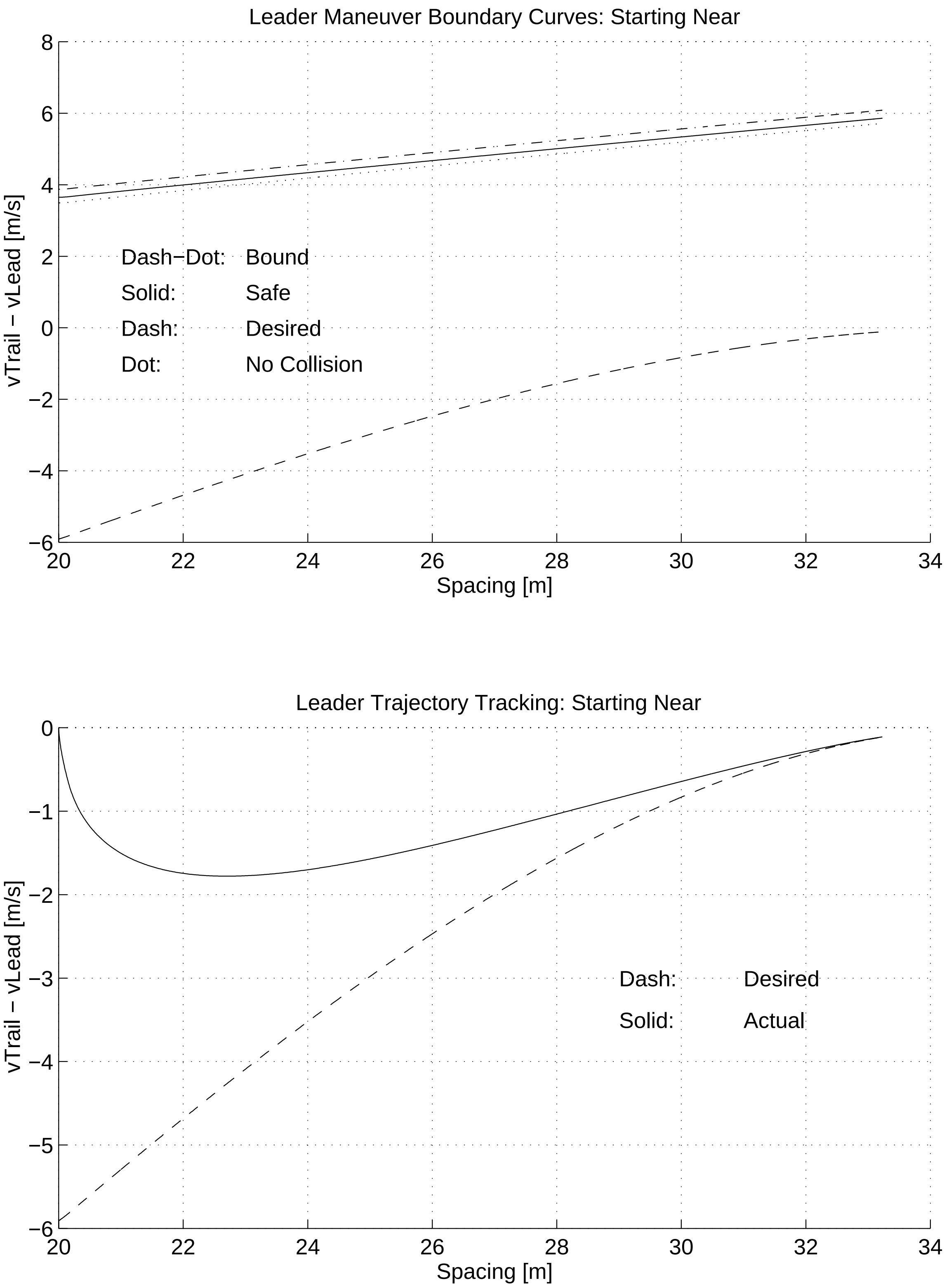


Figure 14

Leader Maneuver Safety Boundaries and Performance: Initially Near Front

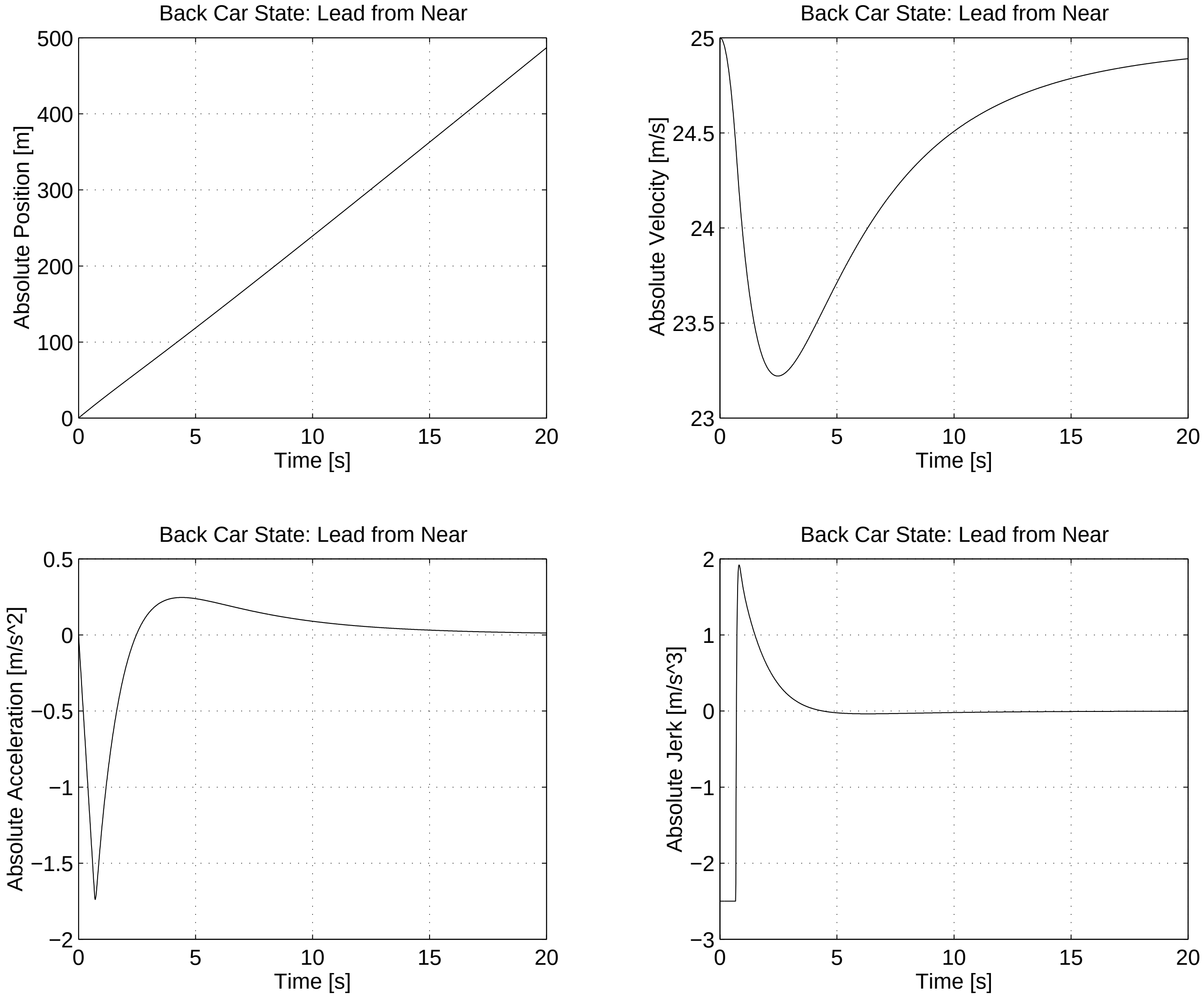


Figure 15

Leader Maneuver State History: Initially Near Car in Front

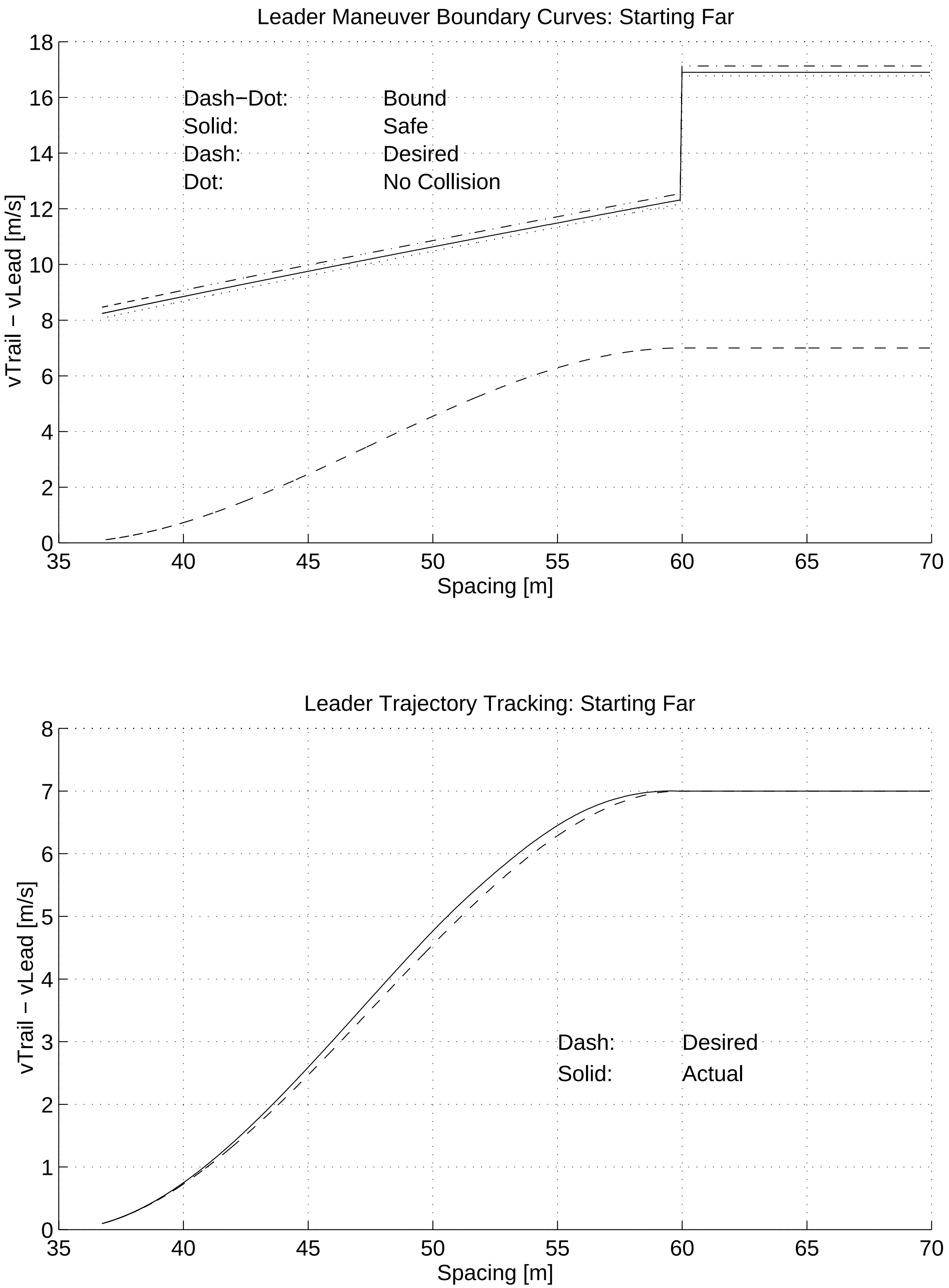


Figure 16

Leader Maneuver Safety Boundaries and Performance: Initially Far from Front

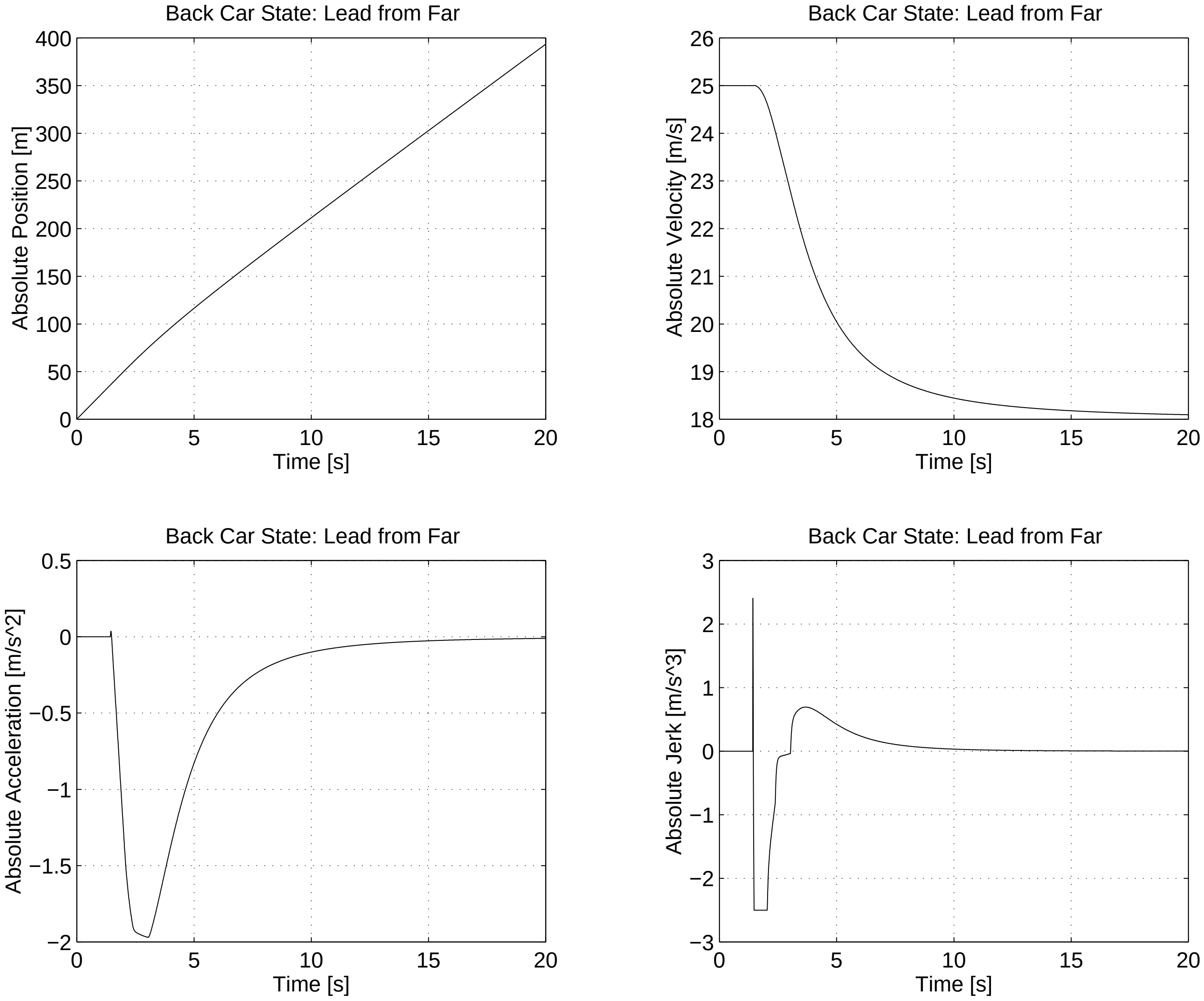


Figure 17

Leader Maneuver State History: Initially Far from Car in Front

PART III

Modifying and Creating Maneuvers

7. The Regulation Layer: *Maneuver Code Structure*

7.1. Function Definitions

In order to standardize and simplify the development of regulation maneuvers, each uses the same set of functions and structure (Appendix C is an example maneuver's source code). Therefore, only the specific equations that uniquely define a maneuver need to be inserted into the framework to begin testing. Each of these functions are shown below. They calculate the variables needed to determine the controller output, such as the desired trajectory and the safety boundary. The functions are common to all maneuvers in name and purpose, though the calculation contained within may be different:

`inv2by2` inverts a 2x2 matrix. This is used in the calculation of spline coefficients.
`traj` returns the desired back vehicle trajectory and its first and second partial derivatives with respect to spacing and front vehicle velocity.
`intxn` calculates the spacing at the point where two desired trajectories intersect. This value is used when that intersection will be smoothed with a spline.
`zone` determines the zone which contains the back vehicle's state.
`spline` calculates the spline coefficients and regressors of the cubic splines at the intersections of desired trajectories. Also calculates the first and second partial derivatives of the coefficients and regressors with respect to spacing and front vehicle velocity.
`vdes` calculates the desired velocity of the back platoon, given the back vehicle's current state and zone.
`dvdes` calculates the first partial derivatives of `vdes` with respect to spacing and front vehicle velocity.
`ddvdes` calculates the second partial derivatives of `vdes` with respect to spacing and front vehicle velocity.
`vsafe` calculates the boundary of X_{safe}: the maximum safe velocity of the back vehicle such that no collision will occur if the back vehicle slams on its brakes as soon as it crosses this velocity limit.
`vbound` calculates the boundary of X_{bound}: the maximum safe velocity of the back vehicle with zero braking delay.
`vnocoll` calculates the boundary of X_{NoColl}: the maximum velocity of the back vehicle that ensures that NO collision occurs.
`region` determines the state space region of the back vehicle desiring to execute or continue a maneuver.

7.2. Function Call Tree

In order to visualize the role of each function within a maneuver module, a function call tree is presented below. Functions contain calls to functions immediately beneath them on the tree. For example, the function ’region’ contains a call to the function ’vsafe’. The functions at the top of the tree are those mentioned above that are called by the master task of the Regulation Layer (regAutoAL.c) as it interprets commands from the Coordination Layer.

```
SafeToSplit
      CarAheadDist        (gets sensor read distance to vehicle ahead)
      CarAheadVel         (gets sensor read relative velocity of vehicle ahead)
      region
InitSplit
DecelToSplit
      Sm_Clock            (gets current simulation time)
      CarAheadDist
      CarAheadVel
      traj
            intxn
            zone
            spline
                  inv2by2
                  vdes
                  dvdes
            vdes
            dvdes
            ddvdes
      region
            vsafe
            vbound
            vnocoll
SafeToASplit
CompleteSplit
```

Both spline and traj call the functions vdes and dvdes. They are called in spline for the purpose of calculating the spline coefficients. They are called in traj because the controller uses these derivatives to calculate the control output. The function region is called in SafeToSplit to check if it is safe to begin splitting, and repeatedly in DecelToSplit to monitor proximity to the safety boundary.

8. Tools for Maneuver Testing

8.1. Tool Overview

The tools described below were developed in order to facilitate easy maneuver development outside of the SmartPath environment, and subsequent testing after the integration into SmartPath. The tools are written in C or MATLAB.

8.2. twocar.exe

This executable is derived from the source code `twocar.c`. It calls the functions within a maneuver module to execute it just as SmartPath would. Some external SmartPath functions, such as `Sm_Clock` and `CarAheadDist`, are also emulated by this program. All of the interfaces needed between SmartPath and the maneuver modules have thereby been reproduced. Once a maneuver module has been successfully run and debugged using this tool, it should work in SmartPath. Even if some problems do arise, many potential problem sources can be eliminated by debugging in this friendly environment.

The name of the tool implies its scope of usefulness. Like all of the maneuvers detailed in this report, it handles only two-vehicle interactions. Complex maneuvers such as Lane Change and Front Dock cannot be simulated without this tool being extended. For the two-vehicle maneuvers, however, it is easy to test maneuvers or even series of maneuvers. In SmartPath, maneuvers necessarily depend on communication protocols and the behavior of other vehicles on the highway. In the two-car environment, these external forces can be eliminated, and maneuvers can be forced based on any user-defined criteria. In addition, the front vehicle of the pair can be forced to perform any sort of behavior to verify the reaction of the back vehicle.

The inputs to the program are the initial state of the two vehicles, the desired maneuvers to be carried out, and some simulation parameters. They are all at the beginning of `twocar.c`. The outputs of the program are two files: `lstate.dat` and `tstate.dat`. They are a history of the states of the front and back vehicles respectively, and are ascii columns with the format:
{Time [s]}{Position [m]}{Velocity [m/s]}{Acceleration [m/s^2]}{Jerk [m/s^3]}

8.3. looklstate.m, looktstate.m

Once `twocar.exe` has been executed, the state of both the front and back vehicles can be viewed with the simple MATLAB scripts: `looklstate.m` and `looktstate.m` respectively. They will display four plots on one page: one plot per state variable. This program was used to generate Figures 7,9,11,13,15,17.

8.4. sortcar.m

When a maneuver has been executed in SmartPath, one would like to view the state history of the vehicles involved. However, the output of SmartPath in *`[casename]`*`.state` records all the cars at once. This simple MATLAB script sorts out the state data of a specified vehicle for subsequent analysis.

PART IV

Join Maneuver Spline Improvements

9. Motivation

In simulating the join maneuver described in the report, significant velocity tracking error can be seen in the SPLINE2 and DECEL zones (Figure 6). This is not the result of improper gain setting in the controller. Rather, it is the result of the saturation of the control jerk in the NORMAL region. This saturation is in place to assure the comfort of passengers under normal operating conditions. In order to eliminate this tracking error without relaxing the comfort constraints, it is necessary to derive desired state space trajectories that can be tracked while maintaining comfortable levels of acceleration and jerk.

One way to accomplish this task would be to simply increase the length of the SPLINE2 zone. The jerk required to transition from zero acceleration in SAFE2 zone to comfortable deceleration in the DECEL zone would then remain within the comfortable jerk limits. However, for longer splines as shown in Figure 18, the resulting cubic spline can have a local maximum between the spline endpoints that exceeds the safe velocity. A vehicle tracking this desired trajectory is thus drawn to the safety boundary and forced to apply emergency braking. To avoid this scenario, it is desirable to redefine SPLINE2 while still striving to minimize the maneuver completion time. In this addendum, SPLINE2 is redefined by setting the jerk of the desired trajectory to be at the comfort limit. It should be noted that by this method, the cubic spline start and end points are calculated from the trajectory boundary conditions. In [2], the spline endpoints are centered on the relative spacing axis around the zone intersection points (SAFE1/SAFE2 and SAFE2/DECEL).

Using these same comfort considerations, a new zone has been added at the end of the DECEL zone: SPLINE3. The purpose of this portion of the join trajectory is to guide the back vehicle to its desired join spacing while also bringing its acceleration and jerk to zero. This allows the follower law to take over vehicle control at its desired target state. In [2], the final state of the back vehicle is not an equilibrium state, and the follower law must bring the back vehicle's acceleration and jerk to zero.

The desired trajectory for SPLINE1 could also be rederived using this same methodology, but it is not done here. The existing cubic spline is left in place as originally designed in [2]. The length of the spline is chosen for a given set of controller gains such that the actual state of the vehicle remains comfortable while tracking SPLINE1. In the SPLINE1 zone, the existing cubic spline does not exhibit the same extremum between the endpoints as in the SPLINE2 zone.

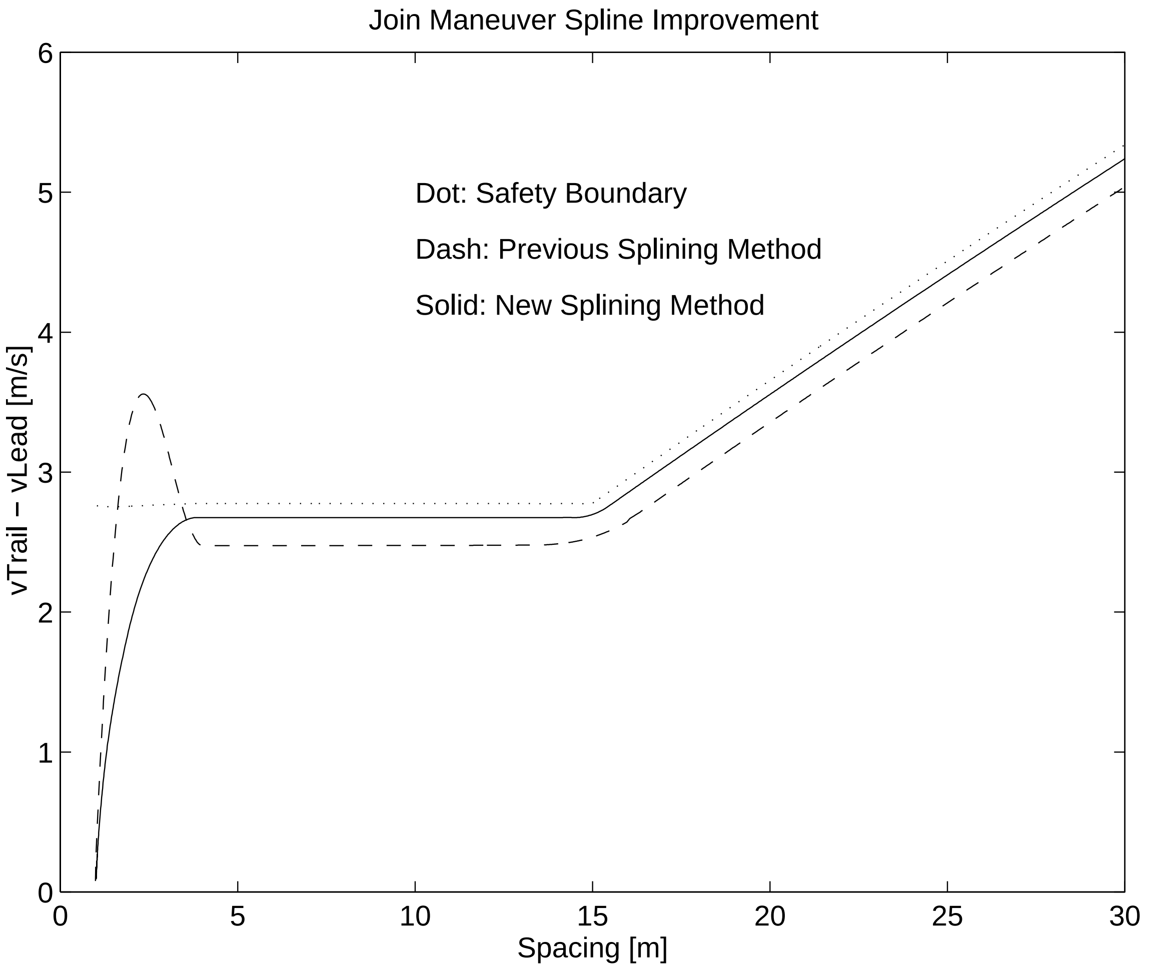


Figure 18

Join Maneuver Splining Method Comparison

10. Derivation of Join Maneuver Desired Trajectory

10.1. Definition of Terms:

j_{ComMax} Maximum comfortable jerk of the back vehicle.

j_{ComMin} Minimum comfortable jerk of the back vehicle (a negative number).

a_{ComMin} Minimum comfortable acceleration of the back vehicle (a negative number).

a_{max} Maximum acceleration of front and back vehicles.

a_{min} Minimum acceleration of front and back vehicles (a negative number).

v_{lead} Lead vehicle absolute velocity.

v_{des} Desired back vehicle absolute velocity.

Δv_{allow} Allowable relative velocity at impact.

Δv_{buff} Buffer relative velocity offset between desired and safe trajectories.

Δx Spacing between the front vehicle’s rear bumper and the back vehicle’s front bumper.

Δx_{join} Desired spacing at the end of a join maneuver.

t Time since the vehicle entered the current zone.

d Delay after the lead vehicle begins hard braking that the back vehicle achieves its maximum deceleration.

end (subscript) Denotes the ending point of a spline (at the smallest
spacing
in the spline zone).

start (subscript) Denotes the starting point of a spline (at the largest spacing in the spline zone).

10.2. Desired Trajectory in SPLINE3 Zone

The desired velocity trajectory for the SPLINE3 portion of the join maneuver is now derived, beginning with the general equation for a vehicle at the maximum comfortable jerk relative to a vehicle in front at a constant velocity:

$$\Delta\dddot{x} = -j_{ComMax} \tag{8}$$

In order for the SPLINE3 zone to be a smooth transition from the DECEL zone to an ideal follower, the following boundary conditions apply:

$$\Delta x_{end,spline3} = \Delta x_{join} \tag{9}$$

$$\Delta\dot{x}_{end,spline3} = 0 \tag{10}$$

$$\Delta\ddot{x}_{end,spline3} = 0 \tag{11}$$

$$\Delta\ddot{x}_{start,spline3} = -a_{ComMin} \tag{12}$$

Integrating (8) and using boundary conditions (12), (10), and (9), the desired relative acceleration, relative velocity, and spacing are:

$$\Delta\ddot{x} = -j_{ComMax} \cdot t - a_{ComMin} \tag{13}$$

$$\Delta\dot{x} = -\frac{1}{2} j_{ComMax}(t^2 - t^2_{end,spline3}) - a_{ComMin}(t - t_{end,spline3}) \tag{14}$$

$$\begin{aligned}\Delta x = {} & j_{ComMax}(-\frac{1}{6}t^3 + \frac{1}{2}t \cdot t^2_{end,spline3} - \frac{1}{3}t^3_{end,spline3}) \\ & + a_{ComMin}(-\frac{1}{2}t^2 + t \cdot t_{end,spline3} - \frac{1}{2}t^2_{end,spline3}) + \Delta x_{join}\end{aligned} \tag{15}$$

Finally, using boundary condition (11) in equation (13):

$$t_{end,spline3} = -\frac{a_{ComMin}}{j_{ComMax}} \tag{16}$$

Now, using the general solution for a cubic equation to solve (15) for *t*, and choosing the real root between 0 and $t_{end,spline3}$:

$$t = \frac{-a_{ComMin} - 2[\frac{3}{4}(-j_{ComMax}^{2}(\Delta x_{join} - \Delta x))]^{\frac{1}{3}}}{j_{ComMax}} \qquad (17)$$

Finally, plugging *t* into equation (14), the desired relative velocity in terms of vehicle spacing is:

$$\Delta \dot{x} = -2[\frac{9}{16} j_{ComMax}(\Delta x_{join} - \Delta x)^{2}]^{\frac{1}{3}} \qquad (18)$$

This yields the desired velocity for the back vehicle when it is advancing to reach the desired join spacing:

$$v_{des,advance} = v_{lead} + 2[\frac{9}{16} j_{ComMax}(\Delta x_{join} - \Delta x)^{2}]^{\frac{1}{3}} \qquad (19)$$

A similar trajectory is derived for when the back vehicle is closer than the desired join spacing:

$$v_{des,retreat} = v_{lead} + 2[\frac{9}{16} j_{ComMin}(\Delta x_{join} - \Delta x)^{2}]^{\frac{1}{3}} \qquad (20)$$

The vehicle spacing at the beginning of SPLINE3 is found by plugging *t*=0 into equation (15):

$$\Delta x_{start,spline3} = \Delta x_{join} - \frac{a_{ComMin}^{3}}{6j_{ComMax}^{2}} \qquad (21)$$

The spacing at the end of SPLINE3 comes directly from boundary condition (9), and thus the SPLINE3 zone desired trajectory is completely determined:

$$\Delta x_{end,spline3} = \Delta x_{join} \qquad (22)$$

10.3. Desired Trajectory in DECEL Zone

The general equation for comfortable acceleration trajectory is:

$$\Delta \ddot{x} = -a_{ComMin} \qquad (23)$$

Boundary Conditions:

$$\Delta x_{end,decel} = \Delta x_{start,spline3} \qquad (24)$$

$$\Delta \dot{x}_{end,decel} = \Delta \dot{x}_{start,spline3} \qquad (25)$$

10.4. Desired Trajectory in SPLINE2 Zone

The general equation for comfortable jerk trajectory is:

$$\Delta \dddot{x} = -j_{ComMin} \qquad (26)$$

Boundary Conditions:

$$\Delta\dot{x}_{start,spline2} = \Delta\dot{x}_{end,safe2} \equiv \mathrm{V} \tag{27}$$

$$\Delta\ddot{x}_{start,spline2} = 0 \tag{28}$$

$$\Delta\ddot{x}_{end,spline2} = -a_{ComMin} \tag{29}$$

The value of V can be determined from the equation for the SAFE2 zone:

$$\mathrm{V} = (a_{max} - a_{min}) \cdot d - \Delta v_{allow} + \Delta v_{buff} \tag{30}$$

So far, the SPLINE2 and DECEL zones each lack one boundary condition for solving their respective desired velocity equations. The two lacking boundary conditions are supplied by constraints at the intersection of the SPLINE2 and DECEL regions. The two desired velocity equations are thus solved simultaneously.

Boundary Conditions:

$$\Delta x_{start,decel} = \Delta x_{end,spline2} \equiv \mathrm{S} \tag{31}$$

$$\Delta\dot{x}_{start,decel} = \Delta\dot{x}_{end,spline2} \tag{32}$$

The value of S will come out of subsequent calculations.

Starting with the DECEL zone, the relative velocity and spacing are found by integrating equation (23):

$$\Delta\dot{x} = -a_{ComMin} \cdot t + c_1 \tag{33}$$

$$\Delta x = -\tfrac{1}{2} a_{ComMin} \cdot t^2 + c_1 t + c_2 \tag{34}$$

Using boundary condition (31):

$$c_2 = \mathrm{S} \tag{35}$$

Then using boundary condition (24), one equation for *c₁* is:

$$-\tfrac{1}{2} a_{ComMin} \cdot t_{end,decel}^2 + c_1 t_{end,decel} + \mathrm{S} = \Delta x_{join} - \frac{a_{ComMin}^3}{6 j_{ComMax}^2} \tag{36}$$

A second equation for c_1 comes from boundary condition (25). But first, from (18):

$$\Delta\dot{x}_{start,spline2} = -\frac{a_{ComMin}^2}{2 j_{ComMax}} \tag{37}$$

Thus:

$$-a_{ComMin} \cdot t_{end,decel} + c_1 = -\frac{a_{ComMin}^2}{2 j_{ComMax}} \tag{38}$$

Solving (36) and (38) simultaneously:

$$c_1 = -\sqrt{2a_{ComMin}\left[\Delta x_{join} - \frac{a_{ComMin}^3}{24\, j_{ComMax}^2} - \mathrm{S}\right]} \tag{39}$$

Equation (34) is then solved for *t*, which is inserted into equation (33). The desired velocity equation is thereby defined independently of *t*:

$$v_{des} = v_{lead} + \sqrt{c_1^2 + 2a_{ComMin}c_2} \tag{40}$$

Continuing on to solve the desired velocity in the SPLINE2 region, equation (26) is integrated using boundary conditions (28), (27), (29), and (31):

$$\Delta\ddot{x} = -j_{ComMin} \cdot t \tag{41}$$

$$\Delta\dot{x} = -\frac{1}{2}\, j_{ComMin} \cdot t^2 + \mathrm{V} \tag{42}$$

$$\Delta x = -\frac{1}{6}\, j_{ComMin} \cdot t^3 + \mathrm{V} \cdot t + \mathrm{S} + \frac{a_{ComMin}^3}{6\, j_{ComMin}^2} - \mathrm{V}\frac{a_{ComMin}}{j_{ComMin}} \tag{43}$$

Using boundary condition (32) in equations (42) and (33), the last unknown S is found:

$$-\frac{1}{2}\, j_{ComMin} \cdot \left(\frac{a_{ComMin}}{j_{ComMin}}\right)^2 + \mathrm{V} = c_1 \tag{44}$$

$$\mathrm{S} = \Delta x_{join} - \frac{a_{ComMin}^3}{24\, j_{ComMax}^2} - \frac{a_{ComMin}^3}{8\, j_{ComMin}^2} + \frac{\mathrm{V}a_{ComMin}}{2\, j_{ComMin}} - \frac{\mathrm{V}^2}{2a_{ComMin}} \tag{45}$$

Equation (43) is then solved for *t*, and inserted into equation (42). Once again, the desired velocity equation defined independently of *t*:

$$v_{des} = v_{lead} - \mathrm{V} + \tfrac{1}{2}\, j_{ComMin}\left[-\frac{2^{1/3}\mathrm{V}}{\mathrm{A}} + \frac{2^{2/3}\mathrm{A}}{-\, j_{ComMin}}\right]^2 \tag{46}$$

Where:

$$\begin{aligned}\mathrm{A} = \Bigg\{ & -27\left(-\tfrac{1}{6}\, j_{ComMin}\right)^2\left(\Delta x_{start,spline2} - \Delta x\right) \\ & + \sqrt{4\left(-\tfrac{1}{2}\mathrm{V}j_{ComMin}\right)^3 + \left[-27\left(-\tfrac{1}{6}\, j_{ComMin}\right)^2\left(\Delta x_{start,spline2} - \Delta x\right)\right]^2}\ \Bigg\}^{1/3}\end{aligned} \tag{47}$$

And by plugging *t*=0 into equation (40):

$$\Delta x_{start,spline2} = \mathrm{S} + \frac{a_{ComMin}^3}{6\, j_{ComMin}^2} - \mathrm{V}\frac{a_{ComMin}}{j_{ComMin}} \tag{48}$$

11. Results

Simulation results are shown in Figures 19 and 20. They correspond to the previous join results shown in Figures 6 and 7. Notice that with the new desired join trajectory, the back vehicle state remains nearly within the comfort limits:
(a:[-2.0, 2.0] m/s^2 , j:[-2.5, 2.5] m/s^3). The acceleration in the DECEL zone slightly exceeds the comfort limit as it tracks the desired trajectory because of the delay *d.* The tracking error in the SPLINE2 and DECEL zones has been significantly reduced. Lastly, the maneuver completion time (assuming the lead vehicle maintains constant velocity) has actually been reduced even though the magnitude of the maximum control jerk required to track the join trajectory is smaller.

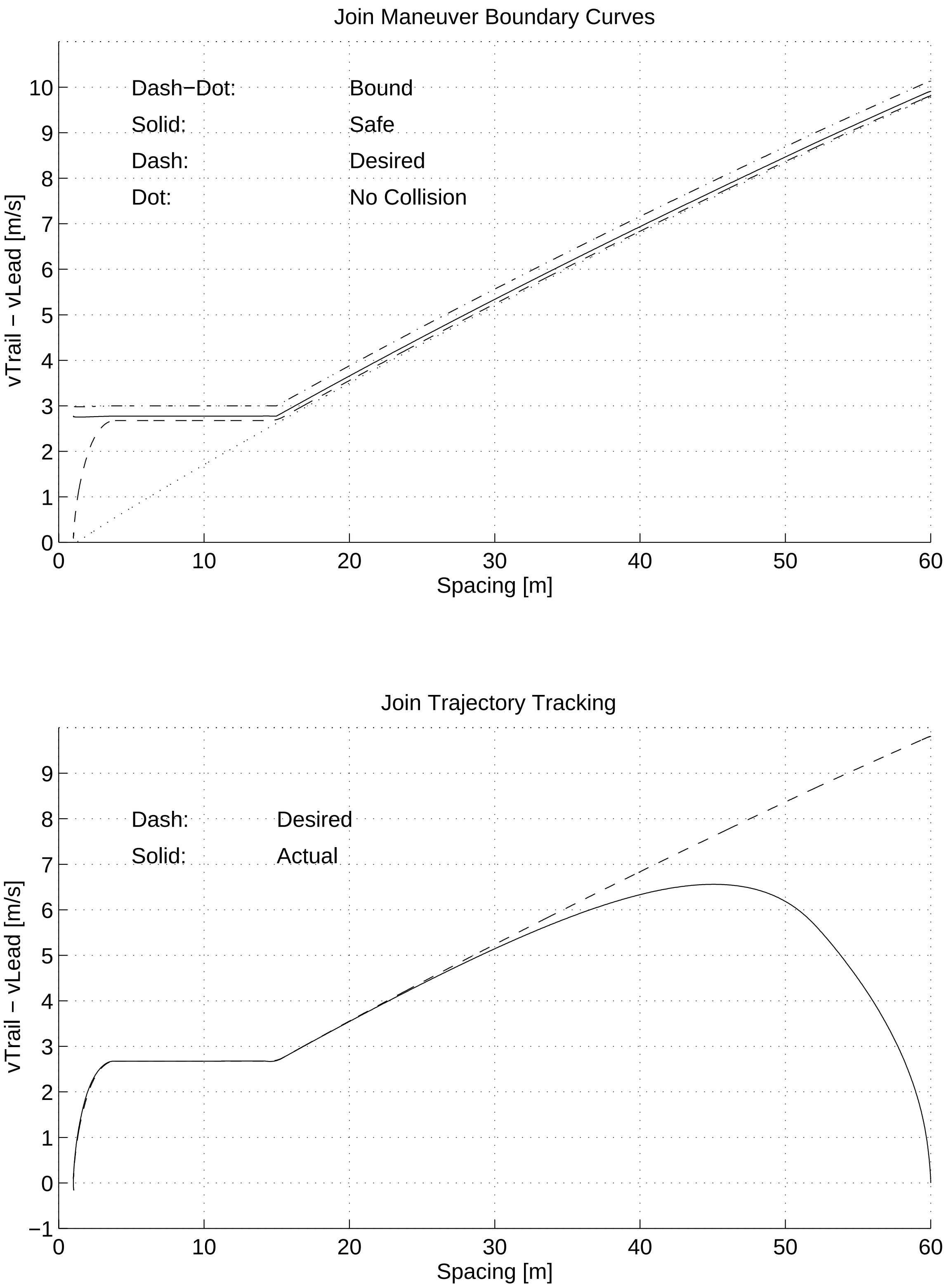


Figure 19

Improved Join Maneuver Safety Boundaries and Performance

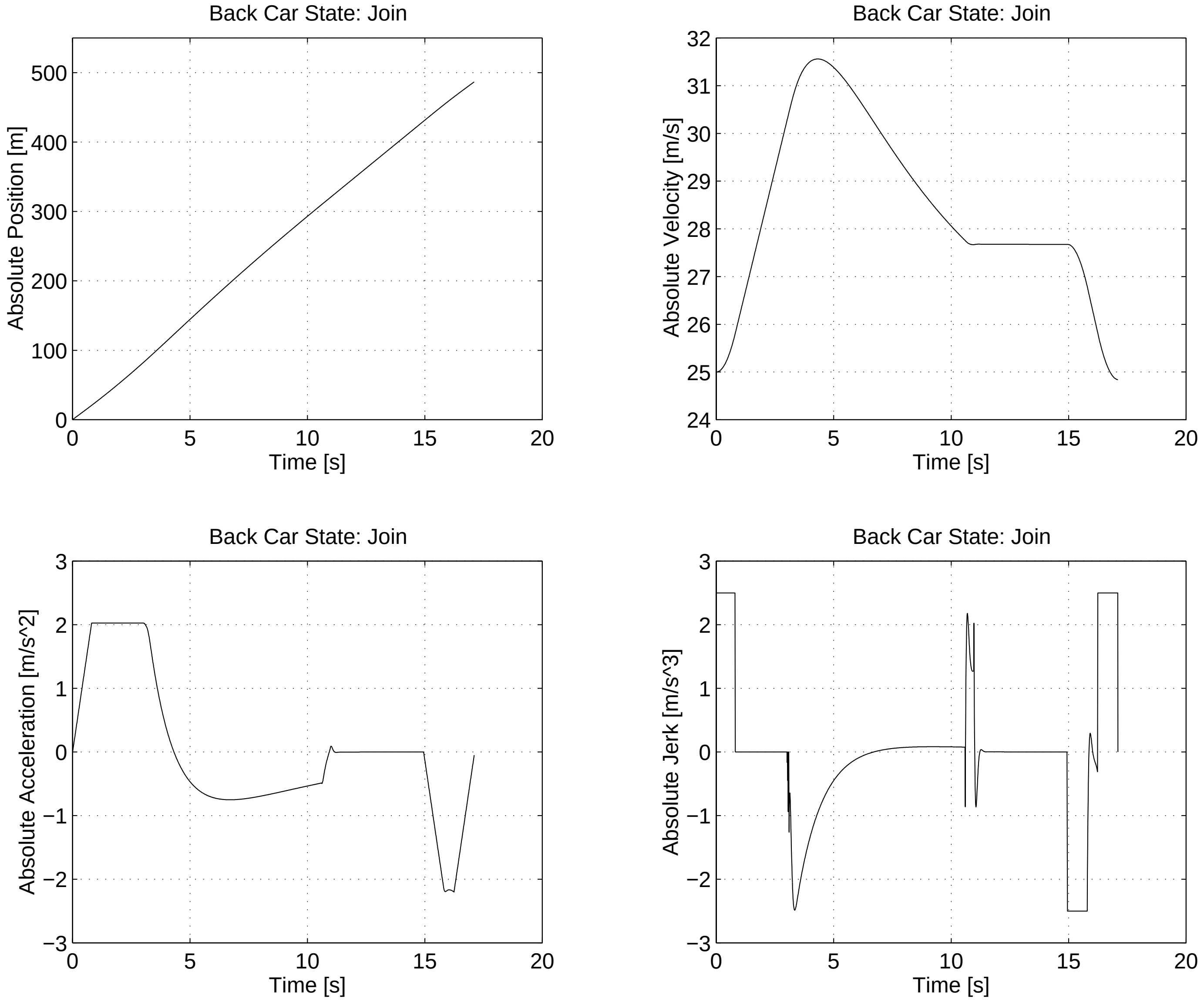


Figure 20

Improved Join Maneuver State History

Appendix A

Derivation of Desired Trajectory for Leader law

Definition of Terms:

v_{adv} Desired velocity of the back vehicle in ADVANCE zone.

v_{ret} Desired velocity of the back vehicle in RETREAT zone.

v_{slow} Highway minimum speed limit under normal circumstances.

v_{link} Advisory speed set by the Link Layer.

v_{front} Absolute velocity of the vehicle in front.

a, b, c, d Spline coefficients.

Δx Spacing between the front vehicle's rear bumper and the back vehicle's front bumper.

Δx_{SR} Spacing at the back vehicle's sensor range limit.

Δx_L Desired spacing under the Leader law.

For zone: ADVANCE

General equation for cubic spline:

$$v_{adv} = a + b(\Delta x_{SR} - \Delta x) + c(\Delta x_{SR} - \Delta x)^2 + d(\Delta x_{SR} - \Delta x)^3$$

Boundary Conditions:

$$v_{adv}\Big|_{\Delta x_{SR}} = v_{link} \qquad \left.\frac{\partial v_{adv}}{\partial \Delta x}\right|_{\Delta x_{SR}} = 0$$

$$v_{adv}\Big|_{\Delta x_{L}} = v_{front} \qquad \left.\frac{\partial v_{adv}}{\partial \Delta x}\right|_{\Delta x_{L}} = 0$$

Solve for coefficients:

$$v_{adv}\Big|_{\Delta x_{SR}} = a + b(0) + c(0) + d(0)$$

$$a = v_{link}$$

$$\left.\frac{\partial v_{adv}}{\partial \Delta x}\right|_{\Delta x_{SR}} = 0 - b - 2c(0) - 3d(0)^2$$

$$b = 0$$

$$v_{adv}\Big|_{\Delta x_{L}} = v_{link} + c(\Delta x_{SR} - \Delta x_L)^2 + d(\Delta x_{SR} - \Delta x_L)^3$$

$$\left.\frac{\partial v_{adv}}{\partial \Delta x}\right|_{\Delta x_{L}} = -2c(\Delta x_{SR} - \Delta x_L) - 3d(\Delta x_{SR} - \Delta x_L)^2$$

$$\begin{bmatrix} (\Delta x_{SR} - \Delta x_L)^2 & (\Delta x_{SR} - \Delta x_L)^3 \\ -2(\Delta x_{SR} - \Delta x_L) & -3(\Delta x_{SR} - \Delta x_L)^2 \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} v_{front} - v_{link} \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} (\Delta x_{SR} - \Delta x_L)^2 & (\Delta x_{SR} - \Delta x_L)^3 \\ -2(\Delta x_{SR} - \Delta x_L) & -3(\Delta x_{SR} - \Delta x_L)^2 \end{bmatrix}^{-1} \begin{bmatrix} v_{front} - v_{link} \\ 0 \end{bmatrix}$$

For zone: RETREAT

General equation for cubic spline:

$$v_{ret} = a + b(\Delta x_L - \Delta x) + c(\Delta x_L - \Delta x)^2 + d(\Delta x_L - \Delta x)^3$$

Boundary Conditions:

$$v_{ret}\Big|_{\Delta x_L} = v_{front} \qquad \frac{\partial v_{ret}}{\partial \Delta x}\bigg|_{\Delta x_L} = 0$$

$$v_{ret}\Big|_0 = \min(v_{slow}, v_{front}) \qquad \frac{\partial v_{ret}}{\partial \Delta x}\bigg|_0 = 0$$

Following the same derivation steps:

$a = v_{front}$

$b = 0$

$$\begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} (\Delta x_L - 0)^2 & (\Delta x_L - 0)^3 \\ -2(\Delta x_L - 0) & -3(\Delta x_L - 0)^2 \end{bmatrix}^{-1} \begin{bmatrix} \min(v_{slow}, v_{front}) - v_{front} \\ 0 \end{bmatrix}$$

Appendix B

Parameter Values Used for this Report

For the sake of repeatability, the controller gains, vehicle, and maneuver parameters are included here. These values are set in `rg`*`[maneuver_name]`*`.h`

```
/* Maneuver parameters                                                   */
dxSpline =          1.0;
dvBuff =            0.1;
dxJoinMax =         1.5;
dxJoin =            1.0;
dxJoinMin =         0.5;
dvJoinMax =         0.2;
dvJoinMin =         -0.2;
dxSplitMax =        59.0;
dxSplit =           55.0;
dxSplitMin =        51.0;
dvSplitMax =        0.1;
dvSplitMin =        -0.1;
dxLead =            35.0;
vGstopMax =         0.001;
vCstopMax =         0.001;

/* Gains                                                                 */
Lambda1 =           0.6;
Lambda2 =           25.0;
Beta =              4.0;
Gamma =             1.2;
F1 =                1.0;
F2 =                10;

/* Vehicle Limitations                                                   */
delay =             0.03;
dxSensorRange =     60.0;
vMax =              40;
aMax =              2.5;
aMin =              -5.0;
jMax =              2.5;
jMin =              -50;

/* Highway Limitations                                                   */
vFast =             35;

/* Passenger Comfort/Safety Limitations                                  */
aComMax =           2;
aComMin =           -2;
jComMax =           2.5;
jComMin =           -2.5
dvAllow =           3.0;

/* To be specified by Link Layer                                         */
vLink =             25.0;
```

Appendix C

Source Code: twocar.c

```
/* FILE: twocar.c                                                        */
/* Testing program for all two car maneuvers                             */
/* Unless noted otherwise, all units are based on: seconds, meters       */
/* INPUTS:                                                               */
/* Parameters in SPECIFY HERE: section                                   */
/* regmerge.c, regsplit.c, regleader.c, reggstop.c, regfollow.c          */
/* OUTPUTS:                                                              */
/* lstate.dat: lead platoon state trajectory                             */
/* tstate.dat: trail platoon state trajectory                            */
/* Last updated: 6/27/96 Jason Carbaugh                                  */

#include "stdio.h"
#include "math.h"
#include "regulation.h"
#include "rgInt.h"

/* Possible lead platoon behaviors                                       */
enum   {
       CONSTANT,           /*maintains initial velocity                  */
       COMFORT,            /*decelerates at comfort decel & jerk         */
       SLAM                /*decelerates at maximum decel & jerk         */
       };
/* Possible trail platoon maneuvers                                      */
enum   {
       JOIN,
       SPLIT,
       LEAD,
       FOLLOW,
       GSTOP,
       CSTOP,
       ESTOP,
       REST
       };

/* Variable declarations                                                 */
Rg_CarState **rgStateTable;      /*SmartPath state table                 */

/*SPECIFY HERE: simulation initial conditions and lead platoon behavior  */
double SimEndTime=20.0;          /*Time of end of simulation WRT start of*/
                                 /*simulation                            */
double xlead[4]={59.5,25,0,0};   /*pos,vel,acc,jerk of lead plat WRT road*/
                                 /*under lead pltn at start of simulation*/
                                 /*updated every sensorTimeStep          */
double xtrail[4]={0,25,0,0};     /*pos,vel,acc,jerk of trail plt WRT road*/
                                 /*under lead pltn at start of simulation*/
                                 /*updated every sensorTimeStep          */
double sensorTimeStep=0.01;      /*period for sensor data updates        */
                                 /*corresponds to SmartPath rgDiscr      */
double controlTimeStep=0.01;     /*period for control calculation        */
                                 /*corresponds to SmartPath rgDiscr      */
double sensorTime=0.0;           /*time for next sensor reading WRT start*/
                                 /*of simulation                         */
double controlTime=0.0;          /*time for next control signal calc WRT */
                                 /*last sensorTime                       */
double xplant[4] = {0,0,0,0};    /*pos,vel,acc,jerk of trail platoon     */
                                 /*WRT road under trail platoon at last  */
                                 /*sensorTime based on integrating contrl*/
                                 /*signal.  Updated every controlTimeStep*/
main()
{
/*Variable declarations                                                  */
double curTime = 0.0;            /*Current time WRT start of simulation  */
```

```
int LeadAct;                     /*enum of lead activity                 */
int TrailAct;                    /*enum of trail activity                */
double (*TrailLaw)();            /*pointer to longitudinal control law   */
FILE *LState;                    /*File for lead platoon state trajctry  */
FILE *TState;                    /*File for trail platoon state trajctry */

/*Open the record files                                                  */
if((LState=fopen("results/lstate.dat","wt"))==NULL)
       printf("Lead state record can’t be opened!\n");
if((TState=fopen("results/tstate.dat","wt"))==NULL)
       printf("Trail state record can’t be opened!\n");

/* Initialize SmartPath parsing table for trail platoon (cid=1)          */
rgStateTable = (Rg_CarState **) calloc(1,sizeof(Rg_CarState *));
rgStateTable[1] = (Rg_CarState *) calloc(1,sizeof(Rg_CarState));

/* Simulation begins here                                                */
for (sensorTime=0.0;sensorTime<SimEndTime;sensorTime+=sensorTimeStep)
       {
/* FORCED LEAD ACTIVITY                                                  */
       if(fabs(sensorTime-0.0)<(sensorTimeStep/10))
              {
              LeadAct =    CONSTANT;
              }

/* FORCED TRAIL ACTIVITY                                                 */
/* Start a join at beginning of simulation                               */
       if(fabs(sensorTime-0.0)<(sensorTimeStep/10))
              {
              printf("Message from twocar.c:\n");
              printf("Join initiated.  Car #1; Time = %lf\n",
                     sensorTime);
              TrailAct =   JOIN;
              InitMerge(1);
              }

/* NORMATIVE LEAD ACTIVITY STATE MACHINE (None.  All forced.)            */

/* NORMATIVE TRAIL ACTIVITY STATE MACHINE                                */
       if(TrailAct ==      JOIN)
              {
              TrailLaw =   AccelToMerge;
              if(CompleteMerge(1) == 1)
                     {
                     printf("Message from twocar.c:\n");
                     printf("Join complete. Car #1; Time = %lf\n",
                            sensorTime);
                     TrailAct =   FOLLOW;
                     InitFollow(1);
                     }
              if(CompleteMerge(1) == 2)
                     {
                     printf("Message from twocar.c:\n");
                     printf("Lead platoon out of sensor range. ");
                     printf("Car #1; Time = %lf\n",sensorTime);
                     TrailAct =   LEAD;
                     InitLead(1);
                     }
              }
       if(TrailAct ==      SPLIT)
              {
              TrailLaw =   DecelToSplit;
              if(CompleteSplit(1) == 1)
                     {
                     printf("Message from twocar.c:\n");
                     printf("Split complete. Car #1; Time = %lf\n",
                            sensorTime);
```

```
                        TrailAct =      LEAD;
                        InitLead(1);
                        }
                if(CompleteMerge(1) == 2)
                        {
                        printf("Message from twocar.c:\n");
                        printf("Lead platoon out of sensor range. ");
                        printf("Car #1; Time = %lf\n",sensorTime);
                        TrailAct =      LEAD;
                        InitLead(1);
                        }
                }
        if(TrailAct ==          GSTOP)
                {
                TrailLaw =      DecelToGstop;
                if(CompleteGstop(1) == 1)
                        {
                        printf("Message from twocar.c:\n");
                        printf("G stop complete. Car #1; Time = %lf\n",
                                sensorTime);
                        TrailAct =      REST;
                        InitRest(1);
                        }
                }
        if(TrailAct ==          CSTOP)
                {
                TrailLaw =      DecelToCstop;
                if(CompleteCstop(1) == 1)
                        {
                        printf("Message from twocar.c:\n");
                        printf("C stop complete. Car #1; Time = %lf\n",
                                sensorTime);
                        TrailAct =      REST;
                        InitRest(1);
                        }
                }
        if(TrailAct ==          ESTOP)
                {
                TrailLaw =      DecelToEstop;
                if(CompleteEstop(1) == 1)
                        {
                        printf("Message from twocar.c:\n");
                        printf("E stop complete. Car #1; Time = %lf\n",
                                sensorTime);
                        TrailAct =      REST;
                        InitRest(1);
                        }
                }
        if(TrailAct ==          FOLLOW)
                {
                TrailLaw =      DoFollow;
                }
        if(TrailAct ==          LEAD)
                {
                TrailLaw =      DoLead;
                }
        if(TrailAct ==          REST)
                {
                TrailLaw =      DoRest;
                }

/* FAILURE CONDITIONS                                                                   */
/* Terminate program if platoons crash                                                  */
        if(xlead[0]-xtrail[0] < 0.0)
                {
                printf("Message from twocar.c:\n");
                printf("Trail platoon crashed into Lead platoon. Car #1; ");
```

```
             printf("Time = %lf\n",sensorTime);
             exit(0);
             }

/* LEAD PLATOON PLANT                                                    */
/* Integrate to get the next lead platoon state                          */
       xlead[0]+=xlead[1]*sensorTimeStep;
       xlead[1]+=xlead[2]*sensorTimeStep;
       xlead[2]+=xlead[3]*sensorTimeStep;

/*Lead plant saturation simulation                                       */
/*Acceleration saturation                                                */
       if(xlead[2]>2.5) xlead[2]=2.5;
       if(xlead[2]<-5.0) xlead[2]=-5.0;
/*Velocity saturation                                                    */
       if(xlead[1]<0) xlead[1]=0;

/*LEAD PLATOON CONTROL                                                   */
/*Calculate control signal for lead platoon                              */
       switch(LeadAct)
             {
             case CONSTANT:
/*Lead acceleration is always 0                                          */
                    xlead[3]=    0;
                    break;
             case COMFORT:
/*Apply brakes at jerk comfort threshold...                              */
                    xlead[3] =   -2.5;
/*...unless the deceleration comfort level is exceeded                   */
                    if(xlead[2] < -2)
                           xlead[3]=0;
                    break;
             case SLAM:
/*Apply brakes at maximum jerk...                                        */
                    xlead[3]=    -50;
/*...unless maximum deceleration is exceeded                             */
                    if(xlead[2] <= -5.0)
                           xlead[3]=0;
                    break;
             }

/*LEAD PLATOON DATA STORAGE                                              */
       fprintf(LState,"%lf %lf %lf %lf %lf\n",
       sensorTime, xlead[0], xlead[1], xlead[2], xlead[3]);

/*TRAIL PLATOON PLANT                                                    */
/*Readjust xplant reference point since it moves every sensorTimeStep    */
       xplant[0]=0.0;
       xplant[1]=xtrail[1];
       xplant[2]=xtrail[2];
       xplant[3]=xtrail[3];

/*This loop is the analog of the one in regapply.c which calls           */
/*LongLawProc(xest,now,cid) every rgDiscr seconds.  Instead of using     */
/*kmer_step, this is just a plain integration to find the trail state    */
/*based on the control jerk sent from AccelToMerge() in regmerge.c       */
       for (controlTime=0.0;controlTime<sensorTimeStep;
       controlTime+=controlTimeStep)
             {
/*integrate to get the next trail platoon state                          */
             xplant[0]+=xplant[1]*controlTimeStep;
             xplant[1]+=xplant[2]*controlTimeStep;
             xplant[2]+=xplant[3]*controlTimeStep;

/*Trail plant saturation simulation                                      */
/*Acceleration saturation                                                */
             if(xplant[2]>2.5) xplant[2]=2.5;
             if(xplant[2]<-5.0) xplant[2]=-5.0;
/*Velocity saturation                                                    */
             if(xplant[1]<0) xplant[1]=0;

/*TRAIL PLATOON CONTROL                                                  */
             curTime=sensorTime+controlTime;
             xplant[3]=(*TrailLaw)(xplant, curTime, 1);
             }

/*Update trail platoon state available for sensors                       */
       xtrail[0]+=xplant[0];
       xtrail[1]=xplant[1];
       xtrail[2]=xplant[2];
       xtrail[3]=xplant[3];

/*TRAIL PLATOON DATA STORAGE                                             */
       fprintf(TState,"%lf %lf %lf %lf %lf\n",
       curTime,xtrail[0],xtrail[1],xtrail[2],xtrail[3]);

       rgStateTable[1]->inc_distance=xplant[0];

       }

printf("Message from twocar.c:\n");
printf("End of simulation time reached. Time = %lf\n",sensorTime);

/*Close data files and end program                                       */
fclose(LState);
fclose(TState);
return(0);
}

/*SmartPath function emulators begin here                                */
double Sm_Clock(void)
{
return(sensorTime);
}

double CarAheadVel(int cid)
{
if((xlead[0] - xtrail[0])<=60.0)
       return(xlead[1] - xtrail[1]);
else
       return(0.0);
}

double CarAheadDist(int cid)
{
if((xlead[0] - xtrail[0])<=60.0)
       return(xlead[0] - xtrail[0]);
else
       return(60.0);
}
```

Appendix D

Sample Maneuver Module Code: regmerge.c

```
/* FILE: regmerge.c                                                      */
/* Contains the functions needed to carry out a Safe Merge               */
/*      1. InitMerge                                                     */
/*      2. SafeToMerge                                                   */
/*      3. SafeToAmerge                                                  */
/*      4. AccelToMerge                                                  */
/*      5. CompleteMerge                                                 */
/*                                                                       */
/* Developers: Jon Frankel, Perry Li                                     */
/* Last updated: 2/11/96 Pin-Yen Chen                                    */
/*            8/28/96 Jason Carbaugh                                     */

/* Copyright (c) 1996 The Regents of the University of California. */
/* Permission to use, copy, modify, and distribute this software and     */
/* its documentation for any purpose and without fee is hereby granted,  */
/* provided that the above copyright notice appear in all copies. The    */
/* University of California makes no representation about the            */
/* suitability of this software for any purpose. It is provided "as is"  */
/* without expressed or implied warranty.                                */


#include "stdio.h"
#include "math.h"
#include "regulation.h"
#include "rgInt.h"
#include "rgallman.h"
#include "rgmerge.h"

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* inv2by2() receives a pointer to a 2x2 matrix in *mat, and returns     */
/* its inverse in *matinv                                                */
/* INPUTS:                                                               */
/* *mat             1x4 row by row representation of a 2x2 matrix        */
/* OUTPUTS:                                                              */
/* *matinv   1x4 row by row representation of the inverse of mat         */
/*************************************************************************/
void inv2by2(double *mat, double *matinv)
{
double det;                      /* determinant of mat                   */

det=(mat[0]*mat[3])-(mat[1]*mat[2]);
matinv[0]=(1/det)*(mat[3]);
matinv[1]=(1/det)*(-mat[1]);
matinv[2]=(1/det)*(-mat[2]);
matinv[3]=(1/det)*(mat[0]);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* traj() returns the desired Trail platoon trajectory and its first     */
/* and second derivatives.                                               */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail cars                         */
/* vLead     absolute velocity of Lead car                               */
/* vTrail    absolute velocity of Trail car                              */
/* OUTPUTS:                                                              */
/* *vDes     desired velocity of Trail car                               */
/* *DvDes    first partial derivatives of vDes                     */
/* *DDvDes   second partial derivatives of vDes                          */
/*************************************************************************/
void traj(double dx, double vLead, double vTrail,
       double *vDes, double *DvDes, double *DDvDes)
{
int Zone;                        /* trail platoon state space region     */
double r;                        /* spline regressor                     */
double Dr[2];              /* deriv of r wrt dx:0 and vLead:1      */
double DDr[4];                   /* 2nd deriv of r wrt dx^2:0, dx*vLead:1*/
                                 /* vLead*dx:2, vLead^2:3                */
double c[4];                     /* spline coefficients                  */
double Dc[4];              /* deriv of c wrt vLead                 */
double DDc[4];                   /* 2nd deriv of c wrt vLead             */
double dxIntxn1;                 /* spcg at intersection of SAFE1/SAFE2  */
double dxIntxn2;                 /* spcg at intersection of SAFE2/DECEL  */

dxIntxn1 =   intxn(vLead, SPLINE1);
dxIntxn2 =   intxn(vLead, SPLINE2);

Zone =              zone(dx, dxIntxn1, dxIntxn2);

/* Calculate spline regressor and coefficients for spline regions        */
/* and derivatives of both                                               */
spline(dx, vLead, Zone, &r, c, Dr, Dc, DDr, DDc);

vdes(dx, vLead, Zone, r, c, vDes);
dvdes(dx, vLead, Zone, r, c, Dr, Dc, DvDes);
ddvdes(dx, vLead, Zone, r, c, Dr, Dc, DDr, DDc, DDvDes);
}

/************************************************************************/
/* DESCRIPTION:                                                          */
/* intxn() returns the spacing at which the desired velocity of          */
/* intersecting curves are equal.                                    */
/* INPUTS:                                                               */
/* vLead     absolute velocity of Lead platoon                       */
/* Zone             enum of spline zone that will smooth the intxn point */
/* RETURN:   spacing at intersection point                               */
/************************************************************************/
double intxn(double vLead, int Zone)
{
double dxIntxn;                  /* return value                         */

if (Zone == SPLINE1)
       {
       dxIntxn =    (2*dvAllow*vLead +
                    aMin*(aMax - aMin)*delay*delay)/(-2*aMin);
       }
else if (Zone == SPLINE2)
       {
       dxIntxn =    (dvAllow - dvBuff - (-aMin + aMax)*delay)*
                    (dvAllow - dvBuff - (-aMin + aMax)*delay)/
                    (2*aCom) + dxJoin;
       }
else
       {
       printf("Error in regmerge.c function intxn():\n");
       printf("No desired curve intersection in this Zone.\n");
       }
return(dxIntxn);
}

/************************************************************************/
/* DESCRIPTION:                                                          */
/* zone() returns the trajectory region which is applicable to the */
/* current trail platoon state.                                          */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail cars                         */
/* dxIntxn1  spacing at when SAFE1/SAFE2 rel velocities are equal        */
/* dxIntxn2  spacing at when SAFE2/DECEL rel velocities are equal        */
/* RETURN:   enum representation of current trajectory region            */
/*************************************************************************/
int zone(double dx, double dxIntxn1, double dxIntxn2)
{
int Zone;                        /* return value                         */
double dxStart1;                 /* spacing at beg of SPLINE1            */
double dxStart2;                 /* spacing at beg of SPLINE2            */
double dxEnd1;                   /* spacing at end of SPLINE1            */
double dxEnd2;                   /* spacing at end of SPLINE2            */

dxStart1 =   dxIntxn1 + dxSpline/2;
dxStart2 =   dxIntxn2 + dxSpline/2;
dxEnd1 =     dxIntxn1 - dxSpline/2;
dxEnd2 =     dxIntxn2 - dxSpline/2;

/* Choose region based on dx, then dv                                    */
if ((dx <= dxSensorRange) && (dx > dxStart1))
       {
       Zone = SAFE1;
       }
else if ((dx <= dxStart1) && (dx > dxEnd1))
       {
       Zone = SPLINE1;
       }
else if ((dx <= dxEnd1) && (dx > dxStart2))
       {
       Zone = SAFE2;
       }
else if ((dx <= dxStart2) && (dx > dxEnd2))
       {
       Zone = SPLINE2;
       }
else if ((dx <= dxEnd2) && (dx > dxJoin))
       {
       Zone = DECEL;
       }
else if ((dx <= dxJoin) && (dx > 0.0))
       {
       Zone = FOLLOW;
       }
else
       {
       printf("Error in regmerge.c function zone():\n");
       printf("Platoons should not be joining.\n");
       }

return(Zone);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* spline() calculates the coefficients and regressors of cubic splines  */
/* at intersections of the vdes curves, and their derivatives            */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail cars                         */
/* vLead     absolute velocity of the Lead platoon                       */
/* zone             traj region defined by desired spline                */
/* OUTPUTS:                                                              */
/* *r        spline regressor                                            */
/* Dr        deriv of r wrt dx:0 and vLead:1                             */
/* DDr       2nd deriv, r wrt dx^2:0,dx*vLead:1,vLead*dx:2,vLead^2:3     */
/* c         cubic spline coefficients                                   */
/* Dc        deriv of c wrt vLead                                        */
/* DDc       2nd deriv of c wrt vLead                                    */
/*********************************************************************(***/
void spline(double dx, double vLead, int zone, double *r, double *c,
             double *Dr, double *Dc, double *DDr, double *DDc)
```

```
{
double rMat[4];                  /* spline regressor derivative matrix   */
double rMatInv[4];               /* inverse of rMat                      */
double dxIntxn;                  /* veh spcg @ desired traj intersection */
double dxStart;                  /* spcg at beg of spline                */
double dxEnd;              /* spcg at end of spline                */
double vStart;                   /* vDes at beg of spline                */
double vEnd;                     /* vDes at end of spline                */
double DvStart[2];               /* DvDes at beg of spline               */
double DvEnd[2];                 /* DvDes at end of spline               */
double a;                        /* intermediate calculation result      */

/* Build matrix whose rows are derivatives of spline regression formula  */
rMat[0] =    dxSpline*dxSpline;
rMat[1] =    dxSpline*dxSpline*dxSpline;
rMat[2] =    2*dxSpline;
rMat[3] =    3*dxSpline*dxSpline;
inv2by2(rMat,rMatInv);

if(zone == SPLINE1)
       {
/* Calculate SAFE1/SAFE2 intersection point and spline endpoints         */
       dxIntxn =    intxn(vLead, SPLINE1);
       dxStart =    dxIntxn + dxSpline/2;
       dxEnd =             dxIntxn - dxSpline/2;
       *r =         dxStart - dx;
/* v and Dv to be matched at dx = dxIntxn + dxSpline/2                   */
       vdes(dxStart,vLead,SAFE1,*r,c,&vStart);
       dvdes(dxStart,vLead,SAFE1,*r,c,Dr,Dc,DvStart);
/* v and Dv to be matched at dx = dxIntxn - dxSpline/2                   */
       vdes(dxEnd,vLead,SAFE2,*r,c,&vEnd);
       dvdes(dxEnd,vLead,SAFE2,*r,c,Dr,Dc,DvEnd);
/* Solve for the regression coefficients                                 */
       c[0] =              vStart;
       c[1] =              -DvStart[0];
       c[2] =       rMatInv[0]*(vEnd - vStart + DvStart[0]*dxSpline) +
                    rMatInv[1]*(DvStart[0] - DvEnd[0]);
       c[3] =       rMatInv[2]*(vEnd - vStart + DvStart[0]*dxSpline) +
                    rMatInv[3]*(DvStart[0] - DvEnd[0]);
/* Solve for coefficient and regressor first derivatives                 */
       a =          sqrt(-aMin*dxSpline +
                    (vLead + dvAllow)*(vLead + dvAllow));

       Dc[0] =             (dvAllow + vLead)/a;
       Dc[1] =             -aMin*(vLead + dvAllow)/(a*a*a);
       Dc[2] =             rMatInv[0]*(1 - Dc[0] - Dc[1]*dxSpline) +
                    rMatInv[1]*Dc[1];
       Dc[3] =             rMatInv[2]*(1 - Dc[0] - Dc[1]*dxSpline) +
                    rMatInv[3]*Dc[1];

       Dr[0] =             -1;
       Dr[1] =             -dvAllow/aMin;
/* Solve for coefficient and regressor second derivatives                */
       DDc[0] =     -aMin*dxSpline/(a*a*a);
       DDc[1] =     aMin*(2*(vLead + dvAllow)*(vLead + dvAllow) +
                    aMin*dxSpline)/(a*a*a*a*a);
       DDc[2] =     rMatInv[0]*(-DDc[0] - DDc[1]*dxSpline) +
                    rMatInv[1]*(DDc[1]);
       DDc[3] =     rMatInv[2]*(-DDc[0] - DDc[1]*dxSpline) +
                    rMatInv[3]*(DDc[1]);

       DDr[0] =     0;
       DDr[1] =     0;
       DDr[2] =     0;
       DDr[3] =     0;
       }
else if(zone == SPLINE2)
       {
/* Calculate SAFE2/DECEL intersection point and spline endpoints         */
       dxIntxn =    intxn(vLead, SPLINE2);
       dxStart =    dxIntxn + dxSpline/2;
       dxEnd =             dxIntxn - dxSpline/2;
       *r =         dxStart - dx;
/* v and Dv to be matched at dx = dxIntxn + dxSpline/2                   */
       vdes(dxStart,vLead,SAFE2,*r,c,&vStart);
       dvdes(dxStart,vLead,SAFE2,*r,c,Dr,Dc,DvStart);
/* v and Dv to be matched at dx = dxIntxn - dxSpline/2                   */
       vdes(dxEnd,vLead,DECEL,*r,c,&vEnd);
       dvdes(dxEnd,vLead,DECEL,*r,c,Dr,Dc,DvEnd);
/* Solve for the regression coefficients                                 */
       c[0] =              vStart;
       c[1] =              -DvStart[0];
       c[2] =       rMatInv[0]*(vEnd - vStart + DvStart[0]*dxSpline) +
                    rMatInv[1]*(DvStart[0] - DvEnd[0]);
       c[3] =       rMatInv[2]*(vEnd - vStart + DvStart[0]*dxSpline) +
                    rMatInv[3]*(DvStart[0] - DvEnd[0]);
/* Solve for coefficient and regressor first derivatives                 */
       Dc[0] =             1;
       Dc[1] =             0;
       Dc[2] =             0;
       Dc[3] =             0;

       Dr[0] =             -1;
       Dr[1] =             0;
/* Solve for coefficient and regressor second derivatives                */
       DDc[0] =     0;
       DDc[1] =     0;
       DDc[2] =     0;
       DDc[3] =     0;

       DDr[0] =     0;
       DDr[1] =     0;
       DDr[2] =     0;
       DDr[3] =     0;
       }

else
       {
       *r =         0;
       Dr[0] =             0;
       Dr[1] =             0;
       DDr[0] =     0;
       DDr[1] =     0;
       DDr[2] =     0;
       DDr[3] =     0;
       c[0] =       0;
       c[1] =       0;
       c[2] =       0;
       c[3] =       0;
       Dc[0] =             0;
       Dc[1] =             0;
       Dc[2] =             0;
       Dc[3] =             0;
       DDc[0] =     0;
       DDc[1] =     0;
       DDc[2] =     0;
       DDc[3] =     0;
       }
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* vdes() returns the desired velocity (vDes) of the trail               */
/* platoon given the zone which defines the current trajectory region    */
/* of the trail platoon state                                            */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail platoons                     */
/* vLead     absolute velocity of Lead platoon                           */
/* zone             traj region in which to calculate desired velocity   */
/* RETURN:   desired velocity of trail platoon                           */
/*************************************************************************/
void vdes(double dx, double vLead, int zone, double r, double *c,
             double *vDes)
{

/* Calculate vDes for the given trajectory region                        */
if(zone == SAFE1)
       {
       *vDes =             -(aMax - aMin)*delay - dvBuff + sqrt(-2*aMin*dx +
                    vLead*vLead  + dvAllow*dvAllow -aMin*(aMax-aMin)
                    *delay*delay);
       }
else if(zone == SAFE2)
       {
       *vDes =             -(aMax - aMin)*delay + dvAllow - dvBuff + vLead;
       }
else if(zone == DECEL)
       {
       *vDes =      vLead + sqrt(2*aCom*(dx - dxJoin));
       }
else if(zone == FOLLOW)
       {
       *vDes =             vLead - sqrt(2*aCom*(dxJoin - dx));
       }
else if((zone == SPLINE1) || (zone == SPLINE2))
       {
       *vDes =             c[0]
                    + c[1]*(r)
                    + c[2]*(r*r)
                    + c[3]*(r*r*r);
       }
else
       {
       printf("Error in regmerge.c function vdes():\n");
       printf("’zone’ has taken an illegal value\n");
       }
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* dvdes calculates the partial derivatives of the vdes curves with      */
/* respect to dx (DvDes[0]) and vLead (DvDes[1])                         */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail platoons                     */
/* vLead     absolute velocity of Lead platoon                     */
/* zone             region in which to calculate desired trajectory      */
/* OUTPUTS:                                                              */
/* *DvDes    first partial derivatives of vDes                     */
/*************************************************************************/
void dvdes(double dx, double vLead, int zone, double r, double *c,
             double *Dr, double *Dc, double *DvDes)
{
double a;                  /* intermediate calculation result            */

if (zone == SAFE1)
       {
             a =    sqrt(-2*aMin*dx + vLead*vLead +
                    dvAllow*dvAllow - aMin*(aMax-aMin)*delay*delay);
       DvDes[0] =   -aMin/a;
       DvDes[1] =   vLead/a;
       }
else if (zone == SAFE2)
       {
       DvDes[0] =   0;
       DvDes[1] =   1;
       }
else if (zone == DECEL)
       {
       DvDes[0] =   aCom/sqrt(2*aCom*(dx-dxJoin));
       DvDes[1] =   1;
       }
else if (zone == FOLLOW)
       {
       DvDes[0] =   aCom/sqrt(2*aCom*(dxJoin-dx));
       DvDes[1] =   1;
       }
else if ((zone == SPLINE1) || (zone == SPLINE2))
       {
       DvDes[0] =   (c[1] + 2*c[2]*r + 3*c[3]*r*r)*Dr[0];
       DvDes[1] =   (Dc[0] + Dc[1]*r + Dc[2]*r*r + Dc[3]*r*r*r) +
                    (c[1] + 2*c[2]*r + 3*c[3]*r*r)*Dr[1];
       }
else
       {
       printf("Error in regmerge.c function dvdes:\n");
       printf("’zone’ has taken an illegal value\n");
       }
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* ddvdes calculates the 2nd partial derivatives of the vdes curves      */
/* with respect to dx^2 (DDvDes[0]), dx*dvLead (DDvDes[1]), dvLead*dx    */
/* (DDvDes[2]), and dvLead^2 (DDvDes[3])                                 */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail platoons                     */
/* vLead     absolute velocity of Lead platoon                     */
/* zone             region in which to calculate desired trajectory      */
/* OUTPUTS:                                                              */
/* *DDvDes   second partial derivatives of vDes                          */
/*************************************************************************/
void ddvdes(double dx, double vLead, int zone, double r, double *c,
             double *Dr, double *Dc, double *DDr, double *DDc,
             double *DDvDes)
{
double a;                  /* intermediate calculation result            */

if (zone == SAFE1)
       {
       a =          sqrt(-2*aMin*dx + vLead*vLead + dvAllow*dvAllow -
                    aMin*(aMax-aMin)*delay*delay);
       DDvDes[0] =  -aMin*aMin/(a*a*a);
       DDvDes[1] =  vLead*aMin/(a*a*a);
       DDvDes[2] =  vLead*aMin/(a*a*a);
       DDvDes[3] =  (a*a - vLead*vLead)/(a*a*a);
       }
else if (zone == SAFE2)
       {
       DDvDes[0] =  0;
       DDvDes[1] =  0;
       DDvDes[2] =  0;
       DDvDes[3] =  0;
       }
else if (zone == DECEL)
       {
       a =          sqrt(2*aCom*(dx - dxJoin));
       DDvDes[0] =  -aCom*aCom/(a*a*a);
       DDvDes[1] =  0;
       DDvDes[2] =  0;
       DDvDes[3] =  0;
       }
else if (zone == FOLLOW)
       {
       a =          sqrt(2*aCom*(dxJoin - dx));
       DDvDes[0] =  aCom*aCom/(a*a*a);
       DDvDes[1] =  0;
       DDvDes[2] =  0;
       DDvDes[3] =  0;
       }
else if ((zone == SPLINE1) || (zone == SPLINE2))
       {
       DDvDes[0] =         Dr[0] *
                    (2*c[2] +
                    (6*c[3]*r)) +
                           DDr[0] *
                    (c[1] +
                    (2*c[2]*r) +
                    (3*c[3]*r*r));
       DDvDes[1] =         Dr[0] *
                    (Dc[1] +
                    (Dc[2]*r + c[2]*Dr[1])*2 +
                    (Dc[3]*r*r + 2*c[3]*r*Dr[1])*3) +
                           DDr[1] *
                    (c[1] +
                    (2*c[2]*r) +
                    (3*c[3]*r*r));
       DDvDes[2] =         Dr[0] *
                    (Dc[1] +
                    (Dc[2]*r + c[2]*Dr[1])*2 +
                    (Dc[3]*r*r + 2*c[3]*r*Dr[1])*3) +
                           DDr[2] *
                    (c[1] +
                    (2*c[2]*r) +
                    (3*c[3]*r*r));
       DDvDes[3] =  (DDc[0] +
                    (DDc[1]*r + Dc[1]*Dr[1]) +
                    (DDc[2]*r*r + 2*Dc[2]*r*Dr[1]) +
                    (DDc[3]*r*r*r + 3*Dc[3]*r*r*Dr[1])) +
                           Dr[1] *
                    (Dc[1] +
                    (Dc[2]*r + c[2]*Dr[1])*2 +
                    (Dc[3]*r*r + 2*c[3]*r*Dr[1])*3) +
                           DDr[3] *
                    (c[1] +
                    (2*c[2]*r) +
                    (3*c[3]*r*r));
       }
else
       {
       printf("Error in regmerge.c function ddvdes():\n");
       printf("’zone’ has taken an illegal value\n");
       }
}

/**************************************************************************/
/* DESCRIPTION:                                                           */
/* vsafe() calculates the maximum safe velocity of the Trail platoon     */
/* such that no collision will occur if the Trail platoon slams on its   */
/* brakes as soon as it crosses this velocity limit.  In addition, */
/* a buffer dvBuff provides a factor of safety.                           */
/* INPUTS:                                                                */
/* dx         spacing between Lead and Trail platoons                     */
/* vLead      absolute velocity of Lead platoon                    */
/* RETURN:    maximum safe velocity of Trail platoon with delay           */
```

```
/*************************************************************************/
double vsafe(double dx, double vLead)
{
double vSafe1;                   /* maximum safe Trail plat vel in SAFE1 */
double vSafe2;                   /* maximum safe Trail plat vel in SAFE2 */
double vSafe;              /* maximum safe Trail plat vel          */

vSafe1 =     -(aMax - aMin)*delay - dvBuff + sqrt(-2*aMin*dx +
             vLead*vLead  + dvAllow*dvAllow -aMin*(aMax-aMin)
             *delay*delay);
vSafe2 =     -(aMax - aMin)*delay + dvAllow -dvBuff + vLead;
vSafe =             Max(vSafe1,vSafe2);
return(vSafe);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* vbound() calculates the maximum safe velocity of the Trail platoon    */
/* with no delay.  That is, an initial state outside vBound() will */
/* certainly result in a collision if the lead platoon slams on its      */
/* brakes, even if the Trail platoon is already slamming on its brakes   */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail platoons                     */
/* vLead     absolute velocity of Lead platoon                     */
/* RETURN:   maximum safe velocity of Trail platoon w/o delay            */
/*************************************************************************/
double vbound(double dx, double vLead)
{
double vBound1;                  /* maximum safe Trail plat vel in SAFE1 */
double vBound2;                  /* maximum safe Trail plat vel in SAFE2 */
double vBound;                   /* return value                         */

vBound1 =    sqrt(-2*aMin*dx + vLead*vLead + dvAllow*dvAllow);
vBound2 =    dvAllow + vLead;
vBound =     Max(vBound1,vBound2);
return(vBound);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* vnocoll() calculates the maximum velocity of the Trail platoon        */
/* with delay that insures that no collision occurs.               */
/* When the Trail platoon velocity is above this value, the Trail  */
/* platoon is allowed to exceed acceleration and jerk comfort            */
/* constraints to avoid collisions below dvAllow. dvBuff adds a factor   */
/* of safety.                                                            */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail platoons                     */
/* vLead     absolute velocity of Lead platoon                     */
/* RETURN:   maximum velocity of Trail platoon w/delay for no impact     */
/*************************************************************************/
double vnocoll(double dx, double vLead)
{
double vNoColl;                  /* return value                         */

vNoColl =    -(aMax - aMin)*delay - dvBuff + sqrt(-2*aMin*dx +
             vLead*vLead - aMin*(aMax-aMin)*delay*delay);

return(vNoColl);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* region() determines the state space region of the Trail platoon */
/* desiring to execute or continue a join maneuver                       */
/* INPUTS:                                                               */
/* dx        spacing between Lead and Trail platoons                     */
/* vLead      absolute velocity of Lead platoon                    */
/* vTrail     absolute velocity of Trail platoon                         */
/* RETURN:    enum representation of current state space region          */
/*************************************************************************/
int region(double dx, double vLead, double vTrail)
{
double vSafe;
double vBound;
double vNoColl;
int Region;                      /* return value                         */

vSafe =       vsafe(dx, vLead);
vBound =      vbound(dx, vLead);
vNoColl =     vnocoll(dx, vLead);

if (dx > dxSensorRange)
       Region =      TOO_FAR;
else if ((dx > dxJoinMax) && (dx <= dxSensorRange))
       {
       if (vTrail <= vNoColl)
              Region =      NORMAL;
       else if((vTrail <= vSafe) && (vTrail > vNoColl))
              Region =      NOCOMFORT;
       else if((vTrail <= vBound) && (vTrail > vSafe))
              Region =      BRAKE;
       else
              Region =      UNSAFE;
       }
else if ((dx >= dxJoinMin) && (dx <= dxJoinMax))
       {
       if (((vLead - vTrail) <= dvJoinMax) &&
              ((vLead - vTrail) >= dvJoinMin))
              Region =      FINISHED;
       else
              {
              if (vTrail <= vNoColl)
                     Region =      NORMAL;
              else if((vTrail <= vSafe) && (vTrail > vNoColl))
                     Region =      NOCOMFORT;
              else if((vTrail <= vBound) && (vTrail > vSafe))
                     Region =      BRAKE;
              else
                     Region =      UNSAFE;
              }
       }
else if ((dx > 0.0) && (dx < dxJoinMin))
       {
       if (vTrail <= vNoColl)
              Region =      NORMAL;
       else if((vTrail <= vSafe) && (vTrail > vNoColl))
              Region =      NOCOMFORT;
       else if((vTrail <= vBound) && (vTrail > vSafe))
              Region =      BRAKE;
       else
              Region =      UNSAFE;
       }
else
       Region =      CRASHED;

return(Region);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* SafeToMerge() checks whether the initial state of the joining trail   */
/* platoon is in the safe region                                         */
/* INPUTS:                                                               */
/* cid       car identification number of Trail platoon leader           */
/* RETURN:   0 for unsafe, 1 for safe                                    */
/*************************************************************************/
int SafeToMerge(int cid)
{
int safe;
int Region;
double vTrail;
double vLead;
double vSafe;
double dx;

/* printf("Checking safe to merge %d\n",cid); */

vTrail =     rgStateTable[cid]->speed;
vLead =             vTrail + CarAheadVel(cid);
dx =         CarAheadDist(cid);

Region =     region(dx,vLead,vTrail);

if((Region == UNSAFE) || (Region == CRASHED))
       safe = 0;
else
       safe = 1;

return(safe);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* InitMerge() initializes the state variables used for a join           */
/* INPUTS:                                                               */
/* cid       car identification number of Trail platoon leader           */
/* OUTPUTS:                                                              */
/* rgStateTable[cid]->(various)                                          */
/*************************************************************************/
void InitMerge(int cid)
{
rgStateTable[cid]->Region =      NORMAL;
rgStateTable[cid]->DLimit =      0;
rgStateTable[cid]->xLeadSens =   0.0;
rgStateTable[cid]->vLeadSens =   0.0;
rgStateTable[cid]->aLeadSens =   0.0;
rgStateTable[cid]->jLeadSens =   0.0;
rgStateTable[cid]->xRef =  0.0;
rgStateTable[cid]->r =           0.0;
rgStateTable[cid]->rDot =  0.0;

rgStateTable[cid]->reg_coord =   NOT_SET;

}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* AccelToMerge() is the longitudinal control law for a join maneuver    */
/* The jerk of the Trail platoon is controlled.  The Trail platoon is    */
/* modelled as a second order system with a pure delay.                  */
/* INPUTS:                                                               */
/* *xest     Trail platoon state (x, v, a)                               */
/* time             time since beginning of simulation [sec]             */
/* cid       car identification number of Trail platoon leader           */
/* RETURN:   jerk of Trail platoon                                       */
/*************************************************************************/
double AccelToMerge(double *xest, double time, int cid)
{
double tCtrl[2];                 /* time of current and last control     */
double tCtrlInc;                 /* time between current & last control  */
double tSens[2];                 /* time of current and last sensor samp */
double tSensInc;                 /* time between current & last snsr smp */
double xRef;                     /* ref posn of xLead and xTrail         */
double xRefInc;                  /* ref posn increment                   */
double xLead;                    /* Lead platoon position wrt xRef */
double xLeadSens[2];             /* Current & last sensed position */
double vLead;                    /* Lead platoon absolute velocity*/
double vLeadSens[2];             /* Current & last sensed Lead velocity  */
double aLeadHat;                 /* estimated accel of Lead platoon      */
double aLeadSens[2];             /* Current & last sensor derived accel  */
double aLeadHatDotHat;           /* est of deriv of est’d acc of Lead pl */
double jLeadSens[2];             /* Current & last sensor derived jerk   */
double r[2];                     /* R.O. observer indep var for acc est  */
double rDot[2];                  /* time deriv of r                      */
double g[2];                     /* q sensitivity factors                */
double q;                        /* non-lin tuning function of observer  */
double xTrail;                   /* Trail platoon postion wrt xRef */
double vTrail;                   /* Trail platoon absolute velocity      */
double aTrail;                   /* Trail platoon acceleration           */
double jTrail;                   /* return value                         */
double vDes;                     /* desired velocity of Trail platoon    */
double DvDes[2];                 /* deriv of vDes wrt dx and vLead       */
double DDvDes[4];                /* 2nd deriv of vDes wrt dx and vLead   */
double aDes;                     /* desired accel of Trail platoon       */
double jDes;                     /* time deriv of des acc of Trail plat  */
double dxSens;                   /* sensed spacing between Lead & Trail  */
double dvSens;                   /* sensed relative vel btwn Lead & Trail*/
int Region;                      /* state space region of Trail platoon  */
int DLimit;                      /* highest deriv of posn from sens info */

/* Retrieve last Trail platoon state table...                      */
/* ...variables static between sensor time indices                         */
tSens[1] =   rgStateTable[cid]->tSens;
xLeadSens[1] =      rgStateTable[cid]->xLeadSens;
vLeadSens[1] =      rgStateTable[cid]->vLeadSens;
aLeadSens[1] =      rgStateTable[cid]->aLeadSens;
jLeadSens[1] =      rgStateTable[cid]->jLeadSens;
xRef =       rgStateTable[cid]->xRef;
DLimit =     rgStateTable[cid]->DLimit;
/* ...variables dynamic between sensor time indices                        */
tCtrl[1] =   rgStateTable[cid]->tCtrl;
r[1] =       rgStateTable[cid]->r;
rDot[1] =    rgStateTable[cid]->rDot;
/* ...current sensor readings                                              */
tSens[0] =   Sm_Clock();
xRefInc =    rgStateTable[cid]->inc_distance;
dxSens =     CarAheadDist(cid);
dvSens =     CarAheadVel(cid);

/* ...reset saturation flag */
Saturation = 0;

/* Calculate sensor and control time steps                                 */
tSensInc =   tSens[0] - tSens[1];
tCtrl[0] =   time;
tCtrlInc =   tCtrl[0] - tCtrl[1];
tFromSens =  tCtrl[0] - tSens[0];

/* On sensor info arrival, update Lead platoon state...                    */
if(time == tSens[0])
       {
/* Calculate Trail platoon state wrt road at the beginning of Join.        */
/* Trail platoon considers its position to be zero when sensors update.   */
       if (DLimit >= 1)
              {
              xRefInc =    rgStateTable[cid]->inc_distance;
              }
       else
             {
             xRefInc =     0.0;
             }
       xRef +=             xRefInc;
       xTrail =     xRef + xest[0];
       vTrail =     xest[1];
       aTrail =     xest[2];
       xLeadSens[0] =      dxSens + xTrail;
       vLeadSens[0] =      dvSens + vTrail;
       if (DLimit >= 1)
             {
             aLeadSens[0] =      (vLeadSens[0] - vLeadSens[1])/tSensInc;
             r[0] =        r[1] + rDot[1]*tCtrlInc;
             }
       else
             {
             aLeadSens[0] =      0.0;
             r[0] =        -(F1*xLeadSens[0] + F2*vLeadSens[0]);
             }
       if (DLimit >= 2)
             jLeadSens[0] =      (aLeadSens[0] - aLeadSens[1])/tSensInc;
       else
             jLeadSens[0] =      0.0;

       xLead =      xLeadSens[0];
       vLead =             vLeadSens[0];
/* Allow one higher derivative to be calculated at the next sensor read  */
       if (DLimit < 2)
             DLimit +=     1;
       else
             DLimit =      2;

/* Store updated Trail platoon state table                               */
/* variables static between sensor time indices                          */
       rgStateTable[cid]->tSens = tSens[0];
       rgStateTable[cid]->xLeadSens =   xLeadSens[0];
       rgStateTable[cid]->vLeadSens =   vLeadSens[0];
       rgStateTable[cid]->aLeadSens =   aLeadSens[0];
       rgStateTable[cid]->jLeadSens =   jLeadSens[0];
       rgStateTable[cid]->r =           r[0];
       rgStateTable[cid]->xRef =        xRef;
       rgStateTable[cid]->DLimit =      DLimit;
       }
else
       {
/* Calculate Trail platoon state wrt road at the beginning of Join.      */
       xTrail =     xRef + xest[0];
       vTrail =     xest[1];
       aTrail =     xest[2];
/*... otherwise, estimate the state of Lead platoon between samples.     */
       xLead =             xLeadSens[1] +
                    (vLeadSens[1]*tFromSens) +
                    (aLeadSens[1]*tFromSens*tFromSens)/2 +
                    (jLeadSens[1]*tFromSens*tFromSens*tFromSens)/6;
       vLead =             vLeadSens[1] +
                    (aLeadSens[1]*tFromSens) +
                    (jLeadSens[1]*tFromSens*tFromSens)/2;
       r[0] =       r[1] + rDot[1]*tCtrlInc;
       }

/* Determine region based on platoon states                              */
Region =     region(xLead - xTrail, vLead, vTrail);

/* Estimate acceleration of Lead platoon using reduced observer          */
aLeadHat =   r[0] + (F1*xLead + F2*vLead);
```

```
/* Get description of desired Trail platoon trajectory                   */
/* Backstep through desired trajectories to get control input            */
traj(xLead - xTrail,vLead,vTrail,&vDes,DvDes,DDvDes);

/* Calculate desired Trail platoon trajectory velocity                   */
/* Saturation Conditions                                                 */
if (vDes > Min(vFast,vMax))
        {
        vDes =          Min(vFast,vMax);
        DvDes[0] =      0.0;
        DvDes[1] =      0.0;
        DDvDes[0] =     0.0;
        DDvDes[1] =     0.0;
        DDvDes[2] =     0.0;
        DDvDes[3] =     0.0;
        Saturation =    1;
        }
else
        vDes =          vDes;

/* Calculate desired Trail platoon trajectory acceleration               */
        aDes =  -Lambda1*(vTrail - vDes) +
                DvDes[0]*(vLead - vTrail) +
                DvDes[1]*aLeadHat;
/* Saturation Conditions                                                 */
switch (Region)
{
case NORMAL:
        if(aDes > aCom)
                aDes =          aCom;
                Saturation =    1;
        else if(aDes < -aCom)
                aDes =          -aCom;
                Saturation =    1;
        else
                aDes =          aDes;

        if(vDes >= Min(vFast,vMax))
                if (aDes > 0.0)
                        aDes =                  0.0;
                        Saturation =    1;
                else
                        aDes =          aDes;
        else
                aDes =          aDes;
        break;
case NOCOMFORT:
        aDes =  -Lambda1*(vTrail - vDes) +
                DvDes[0]*(vLead - vTrail) +
                DvDes[1]*aLeadHat;
        if(aDes > aMax)
                aDes =          aMax;
                Saturation =    1;
        else if(aDes < aMin)
                aDes =          aMin;
                Saturation =    1;
        else
                aDes =          aDes;

        if(vDes >= Min(vFast,vMax))
                if (aDes > 0.0)
                        aDes =                  0.0;
                        Saturation =    1;
                else
                        aDes =          aDes;
        else
```

```
             aDes =       aDes;
       break;
default:
/* The rest of the cases do not use aDes, don’t worry about saturation   */
       aDes = 0.0;
       break;
}

/* Calculate desired Trail platoon trajectory jerk                       */
/* Reduced Observer (estimating aLead only)                              */
/* Calculate gain of effect of (aLead - aLeadHat) on dynamics of         */
/* (vTrail - vDes) and (aTrail - aDes)                                   */
g[0] =       -DvDes[1];
g[1] =       -DvDes[0] -
             DvDes[1]*(Lambda1 + F2) -
             DDvDes[2]*(vLead - vTrail) -
             DDvDes[3]*(aLeadHat);
/* Calculate Reduced Observer tuning function                            */
q =          (g[0]*(vTrail - vDes)*Beta +
             g[1]*(aTrail - aDes))*Gamma;

/* Calculate Reduced Observer state dynamics, estimate of aLeadHatDot    */
rDot[0] =    -F2*r[0] - F1*F2*xLead - (F2*F2+F1)*vLead + q;
aLeadHatDotHat=     rDot[0] + F1*vLead + F2*aLeadHat;

/* Calculate desired Trail platoon trajectory jerk                       */
jDes =       (vLead - vTrail)*
             (DDvDes[0]*(vLead - vTrail) + DDvDes[2]*aLeadHat +
             DvDes[0]*Lambda1) +
                    (aLeadHat)*
             (DDvDes[1]*(vLead - vTrail) + DDvDes[3]*aLeadHat +
             DvDes[1]*Lambda1) +
                    (aTrail)*
             (-Lambda1 - DvDes[0]) +
                    (aLeadHatDotHat)*
             DvDes[1];
/* Saturation Conditions                                                 */
switch (Region)
{
case NORMAL:
       if(jDes > jCom)
             jDes = jCom;
             Saturation = 1;
       else if(jDes < -jCom)
             jDes = -jCom;
             Saturation = 1;
       else
             jDes = jDes;

       if(aDes >= aCom)
             if(jDes > 0.0)
                    jDes =       0.0;
                    Saturation = 1;
             else
                    jDes =       jDes;
       else if(aDes <= -aCom)
             if(jDes < 0.0)
                    jDes =       0.0;
                    Saturation = 1;
             else
                    jDes =       jDes;
       else
             jDes =       jDes;

       if(vDes >= Min(vFast,vMax))
             if(jDes > 0.0)
                    jDes =       0.0;
```

```
                Saturation = 1;
        else
                jDes = jDes;
else
        jDes = jDes;
break;
case NOCOMFORT:
        if(jDes > jMax)
                jDes = jMax;
                Saturation = 1;
        else if(jDes < -jCom)
                jDes = -jCom;
                Saturation = 1;
        else
                jDes = jDes;

        if(aDes >= aMax)
                if(jDes > 0.0)
                        jDes =          0.0;
                        Saturation = 1;
                else
                        jDes =          jDes;
        else if(aDes <= aMin)
                if(jDes < 0.0)
                        jDes =          0.0;
                        Saturation = 1;
                else
                        jDes =          jDes;
        else
                jDes =          jDes;

        if(vDes >= Min(vFast,vMax))
                if(jDes > 0.0)
                        jDes =          0.0;
                        Saturation = 1;
                else
                        jDes = jDes;
        else
                jDes = jDes;
        break;
default:
/* The rest of the cases do not use jDes, don’t worry about saturation   */
        jDes = 0.0;
        break;
}

/* Calculate control jerk                                                */
switch (Region)
{
case TOO_FAR:
        jTrail =        0.0;
        printf("Warning in regmerge.c:\n");
        printf("Lead platoon is out of sensor range.\n");
        break;
case NORMAL:
        jTrail =        -Lambda2*(aTrail - aDes) -
                        Beta*(vTrail - vDes) +
                        jDes;
/* Saturation Conditions for comfort in NORMAL region                    */
        if(jTrail > jCom)
                jTrail =        jCom;
                Saturation = 1;
        else if(jTrail < -jCom)
                jTrail =        -jCom;
                Saturation = 1;
        else
                jTrail =        jTrail;
```

```
       if(aTrail >= aCom)
             if (jTrail > 0.0)
                    jTrail =     0.0;
                    Saturation = 1;
             else
                    jTrail =     jTrail;
       else if(aTrail <= -aCom)
             if (jTrail < 0.0)
                    jTrail =     0.0;
                    Saturation = 1;
             else
                    jTrail =     jTrail;
       else
             jTrail =     jTrail;

       if(vTrail >= Min(vMax,vFast))
             if (jTrail > 0.0)
                    jTrail =     0.0;
                    Saturation = 1;
             else
                    jTrail =     jTrail;
       else
             jTrail =     jTrail;

       break;
case NOCOMFORT:
       jTrail =     -Lambda2*(aTrail - aDes) -
                    Beta*(vTrail - vDes) +
                    jDes;
/* Saturation Conditions for comfort in NOCOMFORT region                 */
       if(jTrail > jMax)
             jTrail =     jMax;
             Saturation = 1;
       else if(jTrail < -jCom)
             jTrail =     -jCom;
             Saturation = 1;
       else
             jTrail =     jTrail;

       if(aTrail >= aMax)
             if (jTrail > 0.0)
                    jTrail =     0.0;
                    Saturation = 1;
             else
                    jTrail =     jTrail;
       else if(aTrail <= aMin)
             if (jTrail < 0.0)
                    jTrail =     0.0;
                    Saturation = 1;
             else
                    jTrail =     jTrail;
       else
             jTrail =     jTrail;

       if(vTrail >= Min(vMax,vFast))
             if (jTrail > 0.0)
                    jTrail =     0.0;
                    Saturation = 1;
             else
                    jTrail =     jTrail;
       else
             jTrail =     jTrail;

       break;
case BRAKE:
       jTrail =     jMin;
```

```
/* Saturation Conditions for vehicle capabilities in BRAKE region        */
       if(aTrail <= aMin)
             jTrail =     0.0;
             Saturation = 1;
       else
             jTrail =     jTrail;

       if(vTrail <= 0.0)
             jTrail =     0.0;
             Saturation = 1;
       else
             jTrail =     jTrail;
       break;
case UNSAFE:
       printf("Error in regmerge.c:\n");
       printf("Unsafe impact is emminent.\n");
       jTrail =     jMin;
/* Saturation Conditions for vehicle capabilities in UNSAFE region */
       if(aTrail <= aMin)
             jTrail =     0.0;
             Saturation = 1;
       else
             jTrail =     jTrail;

       if(vTrail <= 0.0)
             jTrail =     0.0;
             Saturation = 1;
       else
             jTrail =     jTrail;
       break;
case FINISHED:
       jTrail =     0.0;
       break;
case CRASHED:
       printf("Error in regmerge.c:\n");
       printf("Unsafe impact has occured.\n");
       jTrail =     jMin;
/* Saturation Conditions for vehicle capabilities in CRASHED region      */
       if(aTrail <= aMin)
             jTrail =     0.0;
             Saturation = 1;
       else
             jTrail =     jTrail;

       if(vTrail <= 0.0)
             jTrail =     0.0;
             Saturation = 1;
       else
             jTrail =     jTrail;
       break;
}

/* Store updated Trail platoon state table...                            */
/* ...variables dynamic between sensor time indices                      */
rgStateTable[cid]->tCtrl =       tCtrl[0];
rgStateTable[cid]->r =           r[0];
if(Saturation)
        rgStateTable[cid]->rDot =       rDot[0] - q;
else
        rgStateTable[cid]->rDot =   rDot[0];

/* ...variables for flags to other functions                             */
rgStateTable[cid]->Region =      Region;

return(jTrail);
}
```

```
/*************************************************************************/
/* DESCRIPTION:                                                          */
/* SafeToAmerge() is an obsolete function used in regAutoAL.c.           */
/* INPUTS:                                                               */
/* cid       car identification number of Trail platoon leader           */
/* RETURN:   0 for unsafe, 1 for safe                                    */
/*************************************************************************/
int SafeToAmerge(int cid)
{
/* regmerge will not initiate an abort if it is not safe                 */
return(1);
}

/*************************************************************************/
/* DESCRIPTION:                                                          */
/* CompleteMerge() is called by regAutoAL.c to determine if the join     */
/* manuever is complete or if it cannot be completed.              */
/* INPUTS:                                                               */
/* cid       car identification number of Trail platoon leader           */
/* RETURN:   integer encoded join completion status                      */
/*************************************************************************/
int CompleteMerge(int cid)
{
int complete;              /* return value                               */

if (rgStateTable[cid]->Region == FINISHED)
       complete =   1;
else if (rgStateTable[cid]->Region == TOO_FAR)
       complete =   2;
else
       complete =   0;

return (complete);
}
```

Appendix E

Source Code: sortcar.m

```
%File: sortcar.m
%by Jason Carbaugh Jan. 17, 1996
%Sorts out data from a specified car (N) from a filename.state SmartPath data
file
%and puts the filtered data into filename.state.carN
%Revisions:

CIDCOL = 2;  %Data table column which contains the Car ID number
COLQTY = 13; %Number of columns in SmartPath state data table

InputFile = input(’What is the name of the SmartPath state data file? ’,’s’);
FID = fopen(InputFile,’r’);
[StateAll, ElementQty] = fscanf(FID,’%f’);
StateAll = (reshape(StateAll,COLQTY,ElementQty/COLQTY))’;
fclose(FID);
SortCarID = input(’What is the Car ID of the vehicle you wish to sort out? ’);
OutputRow=1;
for InputRow = 1:size(StateAll,1),
       if (StateAll(InputRow, CIDCOL) == SortCarID);
             for ColIndex = 1:COLQTY,
             StateSort(OutputRow,ColIndex) = StateAll(InputRow,ColIndex);
             end
       OutputRow=OutputRow+1;
       end
end
OutputFile = deblank((str2mat((InputFile)’,(’.car’)’,(num2str(SortCarID))’))’);
FID = fopen(OutputFile,’w’);
fprintf(FID,’%f %d %d %f %f %d %f %f %f %f %f %f %f\n’,(StateSort)’);
fclose(FID);
```

References

[1] Eskafi, Farokh, Delnaz Khorramabadi and Pravin Varaiya: *SmartPath: An Automated Highway System Simulator*, Technical Report PATH Memorandum 92-3. Institute of Transportation Studies, Partners for Advanced Transit and Highways, University of California, Berkeley (1992).

[2] Li, Perry, Luis Alvarez and Roberto Horowitz: *AHS Safe Control Laws for Platoon Leaders*, To appear in IEEE Transaction on Control Systems Technology (1997).

[3] Varaiya, Pravin and Steven Shladover: *Sketch of an IVHS Systems Architecture*, Technical Report UCB-ITS-PRR-91-3, Institute of Transportation Studies, University of California, Berkeley (1991).

[4] Eskafi, Farokh: *Modeling and Simulation of the Automated Highway System*, PhD dissertation (1996).

[5] Figure from: Varaiya, Pravin: *Smart Cars on Smart Roads: Problems of Control*, IEEE Transactions on Automatic Control, vol. AC-38, no. 2, pp. 195-207 (1993).

[6] Lygeros, John, Dattaprodh Godbole and Mireille Brouke: *Design of an Extended Architecture for Degraded Modes of Operation of IVHS*, Technical Report UCB-ITS-PWP-95-3, Institute of Transportation Studies, Partners for Advanced Transit and Highways, University of California, Berkeley (1995).

[7] Eskafi, Farokh and Delnaz Khorramabadi: *SmartPath User’s Manual* (1993).

[8] Deshpande, Akash, Aleks Gollu and Luigi Semenzato. SHIFT Reference Manual. PATH Report. University of California, Berkeley (1996).