Models of Vehicular Collision: Development and Simulation with Emphasis on SafetyV: MEDUSA: Theory, Examples, User's Manual, Programmer's Guide and Code

Oliver M. O'Reilly, Panayiotis Papadopoulos, Gwo-Jeng Lo, Peter C. Varadi

California PATH Research Report
UCB-ITS-PRR-99-32

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.

Report for MOU 309

August 1999

ISSN 1055-1425

Models of Vehicular Collision: Development and Simulation with Emphasis on Safety

REPORT – June 1999

Submitted to: PATH (MOU 309)

Oliver M. O'Reilly (PI)
Panayiotis Papadopoulos (PI)
Gwo-Jeng Lo
Peter C. Varadi

Department of Mechanical Engineering
University of California, Berkeley

Abstract

This document constitutes a final report for MOU309. The report contains a User's Manual, Programmer's Guide, source code and underlying theory for the program MEDUSA. This program is capable of simulating both the normative driving dynamics and collision dynamics of an arbitrary number of vehicles. Its range of validity lies in the assumed nature of the vehicular collision, and it is recommended for use in studying low relative velocity impact scenarios at large time-scales.

A significant portion of this report is devoted to presenting the vehicle and road models. The former model is composed of tire, collision, suspension, and sprung mass models. It is also augmented by a collision detection algorithm. The road model is sufficiently general to encompass variations in banking and sloping of the highways.

The report also contains the simulation results of several representative vehicular collision scenarios. These results are supplemented by the input files for MEDUSA and the visualization program SMARTPATH. Several features of these examples are also discussed in the light of the improvements to MEDUSA that they motivated.

Keywords: IVHS America, Vehicle Dynamics, Collision Dynamics, Safety, Computer Simulation, Animation and Simulation.

Executive Summary

This document constitutes a final report for MOU309. The report contains a User's Manual, Programmer's Guide, source code and theory for the program MEDUSA. This program is capable of simulating both the normative driving dynamics and collision dynamics of an arbitrary number of vehicles. Its range of validity lies in the assumed nature of the vehicular collision, and it is recommended for use in studying low relative velocity impact scenarios at large time-scales.

A significant portion of this report is devoted to presenting the vehicle and road models. The former model is composed of tire, collision, suspension, and sprung mass models. It is also augmented by a collision detection algorithm. The road model is sufficiently general to encompass variations in banking and sloping of the highways.

The program MEDUSA is an *ANSI-C* based simulation package which is designed to simulate the normative and collision dynamics of an arbitrary number of vehicles. One of the most attractive features of MEDUSA is its open architecture (including dynamic data management), which facilitates the incorporation of different vehicle models, tire models, etc.. A precursor to the present version of this code was developed under MOU232. The present version has a number of new features. These include: an improved tire model, the ability to simulate vehicles moving on banked and sloped roadways, a more realistic contact algorithm and the ability to incorporate dissipation during a collision,

As in the earlier versions of MEDUSA, the vehicular models are partially based on the theory of a Cosserat point. This theory was developed by M. B. Rubin and extended by A. E. Green and P. M. Naghdi. In the vehicular model, it is used to model the elastic deformation of the sprung mass of the vehicle. Other features of the vehicle model include tire and suspension models. These are supplemented by a collision detection algorithm. In the present version, the lateral surface of the vehicle is modeled using a superellipsoid, which provide a more realistic representation of the vehicle geometry than regular ellipsoids.

A User's Manual for the program MEDUSA is also provided in this report to provide the reader with requisite background on using the program. Complementing this Manual is a section where several representative simulations are introduced and analyzed. This section is written in a style that hopefully also provides a simple tutorial on the capabilities of MEDUSA. The animations of the data provided by MEDUSA were obtained using SMARTPATH, and the default output of MEDUSA is set to conveniently interface with SMARTPATH.

In order to provide the user with avenues for potentially improving the model, a Programmer's Guide along with the source code for the program is also included in this report. The source code differs from the older version in its incorporation of the new features discussed above. It is also shorter than earlier versions because of our efforts to streamline the code.

The reader interested in obtaining these updated versions of MEDUSA should contact either Professor O. M. O'Reilly (oreilly@me.berkeley.edu) or Professor P. Papadopoulos (panos@me.berkeley.edu).

Contents

0 Conventions **1**

1 The Vehicle, Road and Contact Models **2**

1.1 Introduction . . . 2
1.2 The Vehicle Model . . . 2
1.2.1 The Chassis . . . 2
1.2.2 Suspension and Tire Forces . . . 5
1.2.3 Differential Equations of Motion . . . 10
1.2.4 Energy . . . 11
1.3 The Road Model . . . 11
1.3.1 Interpolating the Road . . . 12
1.3.2 The Road Plane . . . 13
1.3.3 The Vehicle and the Road Section . . . 13
1.4 Contact Constraints and Contact Forces . . . 14
1.5 Contact Detection . . . 15
1.5.1 A New Vehicle Geometry . . . 16
1.5.2 Minimum Distance Searching Scheme . . . 18
1.5.3 Unique Contact Point Detection . . . 19
1.6 Time Integration . . . 21
1.7 Collisions and Dissipation . . . 22

2 User's Manual **23**

2.1 Introduction . . . 23
2.2 The Three Input Files . . . 24
2.3 Running the Program . . . 26
2.4 Output Files . . . 28

3 Simulations using MEDUSA **29**

3.1 Animation with SMARTPATH . . . 29
3.2 An Offset Collision . . . 30
3.3 A Wave Problem . . . 32
3.4 Merging of Vehicles . . . 34
3.5 An Evading Maneuver . . . 37

3.6 A Car on a Banked Road 39

4 Programmer's Guide 41

4.1 Introduction 41
4.2 Generalities 42
4.3 The Files *common.h* and *common.c* 43
4.3.1 Programming Tools for Vectors and Matrices 43
4.3.2 Structures 45
4.4 The Function `main()` 48
4.5 The Initialization Module 48
4.6 Time Integration 50
4.7 The Vehicle Model 51
4.8 The Road Model: *road.c* 52
4.9 The Determination of Contact: *contact.c* 52
4.10 Data Output 54
4.11 Adding User Supplied Code 54

Bibliography 55

A The Variable Metric Method 58

B Line Search and Backtracking 61

C A Sample of the File *model.dat* 63

D Structure Definitions 66

D.1 The Vehicle Model Structure 66
D.2 The Simulation Structure 67

E Function Dependencies 68

F The MEDUSA Source Code 71

Chapter 0

Conventions

The summation convention over repeated indices is used for the indices $i, j, n, m = 1, 2, 3$, i.e.,

$$X^i\,\mathbf{d}_i \;=\; \sum_{i=1}^{3} X^i\,\mathbf{d}_i \quad . \tag{0.1}$$

In all other cases, summation is explicitly stated. The notation $x_{(\beta)}$ is used to denote a quantity x belonging to the vehicle β.

To enhance readability of the text, we will use a few notational conventions: Filenames such as *vehicle.c* or *medusa* will appear slanted. Elements of the *ANSI-C* source code such as functions, numbers and variables appear as typed, e.g., `main()`, `3.1415`, `dummy`. The *C* source code is presented as, e.g.,

```
#include <stdio.h>
void main()
{
   printf("Hello World!");
}
```

Input and output from the user screen appears in the same form, e.g.,

```
>> medusa -e
No endtime specified. Type medusa -h for help
```

The prompt `>>` is used to indicate the user input on the command line.

Chapter 1

The Vehicle, Road and Contact Models

1.1 Introduction

In this section, we discuss the vehicle model, the road model, and the contact detection algorithm used by MEDUSA. Although most of these developments were discussed in our earlier reports, there are some significant refinements and improvements. For instance, the tire model is substantially enhanced compared to tire model that was used in earlier versions. Secondly, it is now possible to simulate vehicles moving on curved or banked roads. Finally, the contact detection algorithm employs a different, and more realistic vehicle geometry, than the earlier ellipsoidal model. This chapter closes with a discussion of the numerical time-integration routines that are used in the simulations and some comments on dissipation.

1.2 The Vehicle Model

In this section of the report, the vehicle model is discussed. This model has been substantially revised in comparison to our earlier reports. The vehicle model previously used by MEDUSA is discussed in O'Reilly, Papadopoulos, Lo and Varadi [20, 21]. In response to simulation results, a new tire model was implemented. This accounts for both lateral and longitudinal tire forces, and leads to more realistic results. Furthermore, the road-vehicle interaction now allows curved or banked roads.

1.2.1 The Chassis

In a reference configuration of the chassis, we define a convected Cartesian coordinate system with coordinates X^i $(i = 1, 2, 3)$ and orthonormal basis vectors $\mathbf{E}_i$. The origin of this coordinate system lies at the center of mass of the chassis and the vectors $\mathbf{E}_i$ coincide with the chassis' principal axes of inertia (see Figure 1.1 for the orientation of these vectors).

Clearly, in this coordinate system, a material point of the chassis has position vector

$$\mathbf{R}^*(X^j) = \mathbf{R} + X^i\,\mathbf{E}_i \quad , \tag{1.1}$$

where $\mathbf{R}$ is the position vector of the center of mass of the chassis with respect to an inertial frame. In writing (1.1), the summation convention over repeated indices was employed (cf. equation (0.1)).

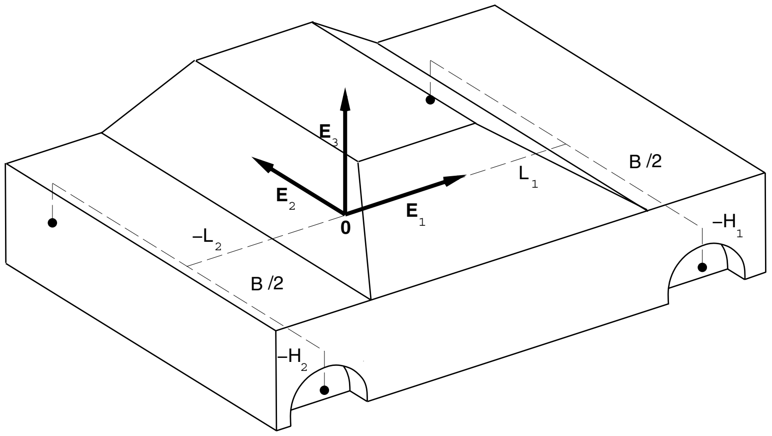

Figure 1.1: *Schematic depiction of the reference configuration of the chassis. The coordinates of the suspension assembly points are also shown.*

The chassis of the vehicle is modeled using the theory of a Cosserat point which was introduced by Rubin [27], and subsequently developed by Green and Naghdi [11]. In this model, the position vector $\mathbf{r}^*(X^i, t)$ of a material point of the chassis at time t is approximated by

$$\mathbf{r}^*(X^j, t) = \mathbf{r}(t) + X^i\,\mathbf{d}_i(t) \quad . \tag{1.2}$$

The vector $\mathbf{r}(t)$ is called the position vector of the Cosserat point. Here, it is the position vector of the center of mass of the chassis at time t and it corresponds to $\mathbf{R}$ in the reference configuration. The three vectors $\mathbf{d}_i(t)$ are called the directors of the Cosserat point. In the reference configuration, they correspond to the basis vectors $\mathbf{E}_i$.

The deformation gradient $\mathbf{F}$ associated with the motion (1.2) can be expressed as

$$\mathbf{F} = \mathbf{d}_i \otimes \mathbf{E}_i \quad , \tag{1.3}$$

where the symbol $\otimes$ denotes the usual tensor product. The position vector $\mathbf{r}^*$ can now also be written as

$$\mathbf{r}^* \;=\; \mathbf{F}(t)\,(\mathbf{R}^* \;-\; \mathbf{R}) \;+\; \mathbf{r} \quad . \tag{1.4}$$

This notation was employed by Cohen and Muncaster [4] in their theory of pseudo-rigid bodies which is closely related to the theory of a Cosserat point with three directors. This notation also indicates that a pseudo-rigid ellipsoid remains ellipsoidal in any subsequent deformation; which is consistent with classical results on homogeneous deformations which may be found in Truesdell and Toupin [30, Sections 42-46]. We used this property earlier to determine contact points of two colliding vehicles. However, it was noted that approximating the lateral surface of a vehicle using a ellipsoid lead to physically unrealistic post-collision scenarios. Later in this report, a new approximation to a vehicle's lateral surface will be discussed.

It is convenient at this point to outline some further notation. We will denote the position vectors of a material point lying on the surface σ of the vehicle in its reference and present configurations, respectively, by

$$\mathbf{R}^{\sigma} = \mathbf{R}^{*}(X^{j}_{\sigma}) \quad , \quad \mathbf{r}^{\sigma} = \mathbf{r}^{*}(X^{j}_{\sigma}, t) \quad , \tag{1.5}$$

where X^{j}_{σ} are the referential Cartesian coordinates of that surface point.

The velocity and director velocities of the Cosserat point are

$$\mathbf{v} = \dot{\mathbf{r}} \ , \quad \mathbf{w}_i = \dot{\mathbf{d}}_i \quad , \tag{1.6}$$

where a superposed dot denotes time derivative. The relevant equations of motion of the chassis are the balance of linear momentum and the three balances of director momenta:

$$m\,\dot{\mathbf{v}} = \mathbf{l}^{0} \quad , \quad my^{ij}\,\dot{\mathbf{w}}_i = \mathbf{l}^{i} - \mathbf{k}^{i} \quad . \tag{1.7}$$

In these equations, m is the mass of the vehicle and $y^{ij} = y^{ji}$ are its inertia parameters. These parameters are related to the vehicle's referential inertia tensor $\mathbf{J}_0 = J^{ij}_0 \mathbf{E}_i \otimes \mathbf{E}_j$ as follows:

$$my^{ij} = -J^{ij}_0 = 0 \quad , \quad i \neq j \quad , \tag{1.8}$$

$$\begin{pmatrix} J^{11}_0 \\ J^{22}_0 \\ J^{33}_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} my^{11} \\ my^{22} \\ my^{33} \end{pmatrix} \quad . \tag{1.9}$$

Equation (1.8) follows from the fact that $\mathbf{E}_i$ coincide with the principal axes of inertia of the chassis, and $\mathbf{R}$ is the position vector of the center of mass of the chassis in its fixed reference configuration. Vehicle inertia parameters can be derived from published data, see, e.g., Garrot [10] and Heydinger et al. [13].

The vectors $\mathbf{l}^{0}(t)$ and $\mathbf{l}^{i}(t)$ are called the applied force and the applied director forces, respectively[1]. For the vehicle model, they are calculated using

$$\mathbf{l}^{0} = \sum_{q=1}^{4} \mathbf{f}^{q} - mg\,\mathbf{E}_3 \quad , \quad \mathbf{l}^{i} = \sum_{q=1}^{4} X^{i}_{q}\,\mathbf{f}^{q} \quad . \tag{1.10}$$

[1]It is customary to use the symbol $\mathbf{n}$ to denote the applied force $\mathbf{l}^{0}$. However, we use the former symbol in this report to denote a unit outward normal.

In this equation, $g = 9.81\,[m/s]$ is the gravitational acceleration. The point forces $\mathbf{f}^q$ ($q = 1, 2, 3, 4$) are generated by the suspensions and the wheels[2]. These forces act on the suspension assembly points which are material points of the chassis with the material coordinates

$$X_q^i \left\{ \begin{array}{lrl} q=1: & (\;\;L_1\,,\;\;\;B/2\,,-H_1\,) & \text{left front} \\ q=2: & (\;\;L_1\,,-B/2\,,-H_1\,) & \text{right front} \\ q=3: & (-L_2\,,\;\;\;B/2\,,-H_2\,) & \text{left rear} \\ q=4: & (-L_2\,,-B/2\,,-H_2\,) & \text{right rear} \end{array} \right. \quad . \tag{1.11}$$

The various quantities in this equation are also depicted in Figure 1.1.

In equations (1.7), $\mathbf{k}^i$ are called the intrinsic director forces. Their function is similar to that of a stress tensor in continuum mechanics, and constitutive equations for the material response are required. In the present vehicle model, we assume a nonlinearly elastic, homogeneous, St. Venant–Kirchhoff material with Lamé constants λ and μ. The resulting constitutive equations are

$$\mathbf{k}^i = \frac{V}{2}\big(\lambda(\mathbf{d}_j \cdot \mathbf{d}_j - 3)\,\mathbf{d}_i + 2\mu(\mathbf{d}_i \cdot \mathbf{d}_n - \delta_{in})\,\mathbf{d}_n\big) \quad ; \quad \delta_{in} = \left\{ \begin{array}{l} 0 \;:\; i \neq n \\ 1 \;:\; i = n \end{array} \right. \quad , \tag{1.12}$$

where the volume V encompasses the entire chassis. The Lamé constants are related to Young's modulus E and Poisson's ratio ν by (see, e.g., Sokolnikoff [29]):

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \quad , \quad \mu = \frac{E}{2(1+\nu)} \quad . \tag{1.13}$$

1.2.2 Suspension and Tire Forces

Unlike previous versions of Medusa, the road is not assumed to be a horizontal plane. Let the local orientation of the road be given by the orthonormal vector triad $\{\mathbf{e}_i\}$, where $\mathbf{e}_3$ is the upward normal to the road plane (cf. Section 1.3.2). Since the highway's radius of curvature is large compared to the dimensions of a vehicle, we can take this triad to be the same for all four wheels (numbered $q = 1, \ldots, 4$).

It will be assumed that the wheels roll upright on the road plane so that the camber angles are negligible. The orientations of the wheel planes can now be specified with unit wheel heading vectors $\mathbf{e}_x^q$ For the rear wheels, these heading vectors $\mathbf{e}_x^3 = \mathbf{e}_x^4$ are chosen to be parallel to the projection of $\mathbf{d}_1$ onto the road plane, i.e.,

$$\mathbf{e}_x^3 = \mathbf{e}_x^4 = \frac{\mathbf{d}_1 - (\mathbf{d}_1 \cdot \mathbf{e}_3)\,\mathbf{e}_3}{||\mathbf{d}_1 - (\mathbf{d}_1 \cdot \mathbf{e}_3)\,\mathbf{e}_3||} \quad . \tag{1.14}$$

Here, $\|.\|$ denotes the length of a vector. The heading vectors of the (steered) front wheels are derived from the rear wheels using a steering angle θ:

$$\mathbf{e}_x^1 = \mathbf{e}_x^2 = \cos\theta\,\mathbf{e}_x^3 - \sin\theta\,\mathbf{e}_x^3 \times \mathbf{e}_3 \quad , \tag{1.15}$$

[2]Recall from Chapter 0 that the summation convention does not apply to the index q.

This corresponds to a planar counterclockwise rotation about $\mathbf{e}_3$. For each wheel, we can thus define a local orthonormal basis $\{\mathbf{e}^q_x, \mathbf{e}^q_y, \mathbf{e}^q_z\}$ by defining $\mathbf{e}^q_z = -\mathbf{e}_3$ and $\mathbf{e}^q_y = \mathbf{e}^q_z \times \mathbf{e}^q_x$ (cf. Figure 1.2).

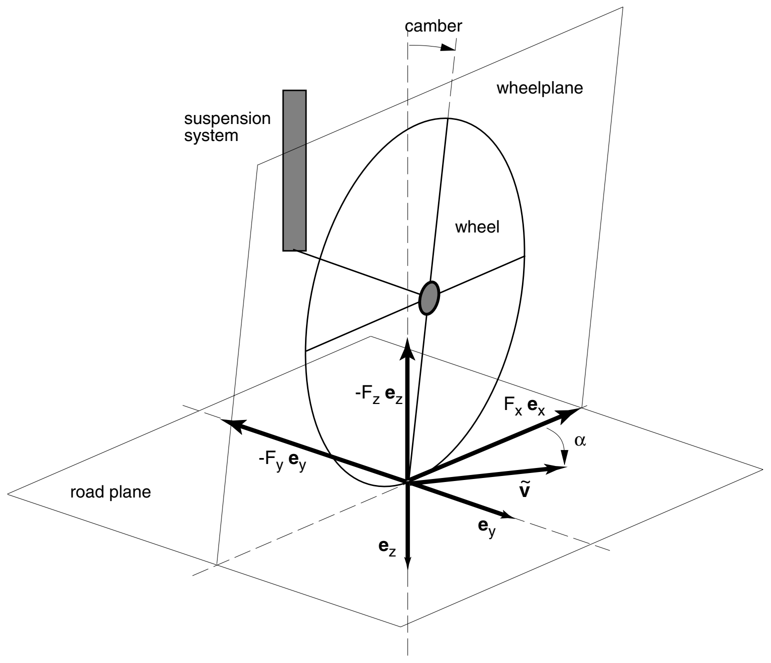
camber
wheelplane
suspension system
wheel
-F_z e_z
F_x e_x
α
-F_y e_y
road plane
ṽ
e_y
e_z

Figure 1.2: *Schematic depiction of one tire illustrating the forces and kinematic quantities defined in the text. The camber angle is shown for completeness but is assumed to be negligible.*

The wheels are assumed massless. The applied forces $\mathbf{f}^q$ in equations (1.7) are calculated as follows:

$$\mathbf{f}^q = F^q_x\,\mathbf{e}^q_x - F^q_y\,\mathbf{e}^q_y - F^q_z\,\mathbf{e}_z \quad , \tag{1.16}$$

where F^q_x and F^q_y are the longitudinal and lateral wheel forces, respectively, and F^q_z are the magnitudes of the suspension forces:

$$-F^q_z = C^q(\Delta_q - \Delta) + D^q\dot{\Delta}_q \quad , \quad C^q, D^q = \begin{cases} C_1, D_1 & : \ q = 1, 2. \\ C_2, D_2 & : \ q = 3, 4. \end{cases} \tag{1.17}$$

In this equations, Δ_q denotes the distance from a suspension assembly point (cf. Figure 1.1) to the road plane, measured along the normal $\mathbf{n}_3$, and Δ denotes a reference length which is assumed to be the same for all four suspensions. The parameters $\{C_1, D_1\}$ and $\{C_2, D_2\}$ are the linear spring and damping coefficients of the front and rear suspensions, respectively.

Next, we need to calculate the velocities of the wheel centers $\tilde{\mathbf{v}}_q$. They are the projections of the suspension assembly point velocities into the road plane:

$$\tilde{\mathbf{v}}_q = \mathbf{v}_q - (\mathbf{v}_q \cdot \mathbf{e}_3)\,\mathbf{e}_3 \quad , \quad \mathbf{v}_q = \mathbf{v} + X^i_q\,\mathbf{w}_i \quad . \tag{1.18}$$

We now provide the formulae to calculate F_x^q and F_y^q for each wheel. We have adopted a particular tire model from Allen et al. [1] for this purpose. Negligible camber thrust will be assumed. This model extends the popular CALSPAN tire model to provide tire responses over a full range of maneuvering conditions. The governing model equations represent polynomial curve fits of measured data and are computationally efficient compared to more elaborate models such as the magic tire model of Pacejka [25]. A further advantage is the availability of published parameters and tire data by the CALSPAN corporation, see Schuring [28]. However, the formulae have certain weaknesses which we will address in several remarks at the end of this section. In the interest of notational brevity, and without the risk of confusion, we will drop the explicit index q for the remainder of this section.

To proceed, we define two standard kinematic quantities in tire modeling, namely the (longitudinal) slip s (cf. Allen et al. [1]) and the (side) slip angle α (cf. Figure 1.2):

$$s = 1 - \frac{R\omega}{\tilde{\mathbf{v}} \cdot \mathbf{e}_x} \quad , \quad \alpha = \arctan\left(\frac{\tilde{\mathbf{v}} \cdot \mathbf{e}_y}{\tilde{\mathbf{v}} \cdot \mathbf{e}_x}\right) \quad . \tag{1.19}$$

In this equation, $R\omega$ denotes the circumferential speed of the tire, and R is the tire radius.

We now record the formulae for the tire contact patch length a_{p0}, the lateral and longitudinal stiffness coefficients K_s and K_c, respectively, and the peak tire/road coefficient of friction:

$$\begin{aligned} a_{p0} &= \frac{0.0768\sqrt{F_z F_{ZT}}}{T_W\,(T_p+5)} \quad , \\ K_s &= \frac{2}{a_{p0}^2}\,(A_0 + A_1 F_z - \frac{A_1}{A_2}\,F_z^2) \quad , \\ K_c &= \frac{2}{a_{p0}^2}\,F_z\,(CS/FZ) \quad , \\ \mu_0 &= (B_1 F_z + B_3 + B_4\,F_z^2)\,\frac{SN_P}{SN_T} \quad . \end{aligned} \tag{1.20}$$

The parameters that appear in these equations are listed below. For large slips, the tire looses traction and starts to slide. This transition is captured by the coefficients K_c' and μ:

$$\begin{aligned} K_c' &= K_c + (K_s - K_c)\sqrt{\sin^2\alpha + s^2\,\cos^2\alpha} \quad , \\ \mu &= \mu_0\,(1 - K_\mu\,\sqrt{\sin^2\alpha + s^2\,\cos^2\alpha}) \quad . \end{aligned} \tag{1.21}$$

The fact that the longitudinal and lateral forces do not develop independently is captured by the composite slip σ. The force saturation function reflects the fact that the tire forces do not increase linearly with slip:

$$\begin{aligned} \sigma &= \frac{\pi\,a_{p0}^2}{8\mu_0 F_z}\sqrt{K_s^2\,\tan^2\alpha + K_c^2\,\frac{s^2}{(1-s)^2}} \quad , \\ f &= \frac{c_1\sigma^3 + c_2\sigma^2 + 4/\pi\,\sigma}{c_1\sigma^3 + c_3\sigma^2 + c_4\sigma + 1} \quad . \end{aligned} \tag{1.22}$$

Finally, the longitudinal and lateral wheel forces F_x^q and F_y^q are calculated:

$$F_x = \mu F_z \frac{-f K_c' s}{\sqrt{K_s^2 \tan^2\alpha + K_c' s^2}} \quad , \quad F_y = \mu F_z \frac{f K_s \tan\alpha}{\sqrt{K_s^2 \tan^2\alpha + K_c' s^2}} \quad . \tag{1.23}$$

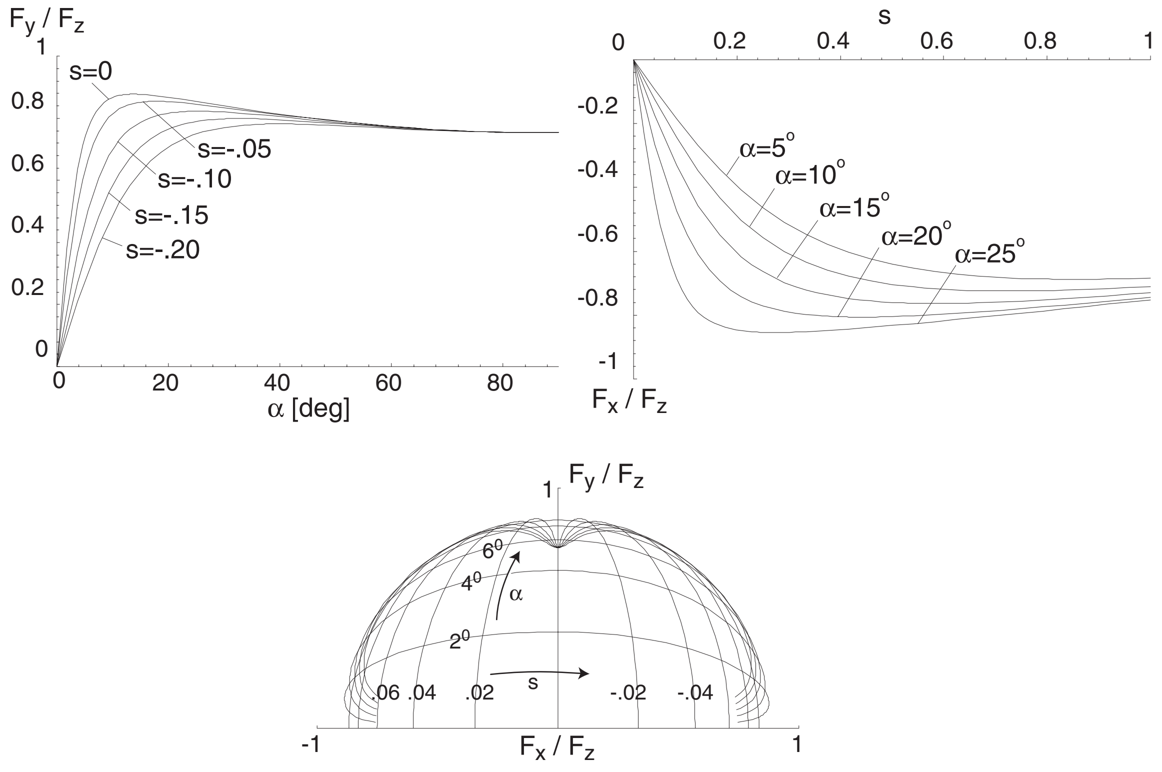

Figure 1.3: *Goodyear 185SR14 tire force diagrams and carpet plot for* $F_Z = 2500\,[N]$.

There is always the concern that the equations presented may be erroneous due to type setting errors, or that errors are introduced when the tire model is coded. A good way to examine the validity of the equations is by way of characteristic force diagrams. They exhibit standard patterns that can reveal errors. For the sake of completeness, we therefore present the force diagrams for a Goodyear 185SR14 tire in Figure 1.3. These plots were obtained with the following set of parameters:

$$A_0 = 7092.7808\,[N] \quad , \quad A_1 = 11.94\,[.] \quad , \quad A_2 = 13571.0848\,[N] \quad ,$$

$$B_1 = -2.5446429 * 10^{-5}\,[N^{-1}] \quad , \quad B_3 = 1.007\,[.] \quad , \quad B_4 = -5.291374 * 10^{-11}$$

$$CS/FZ = 18\,[.] \quad , \quad SN_T = 85\,[.] \quad , \quad SN_P = 85\,[.] \quad . \tag{1.24}$$

These are the well-known CALSPAN parameters listed in Schuring [28]. The parameters SN_T and SN_P are the test skid number and the pavement skid number, respectively. Next, there are three parameters which must be used with non-SI units in the formulas: the tire design load at operating pressure $F_{ZT}\,[lbs]$, the tread width $[in]$ and the tire pressure $T_p\,[psi]$. In particular for our example,

$$F_{ZT} = 1160\,[lbs] \quad , \quad T_W = 5\,[in] \quad , \quad T_p = 28\,[psi] \quad . \tag{1.25}$$

The remaining parameters are form factors for a bias ply tire (see Allen et al. [1]):

$$K_\mu = 0.2\,[.] \quad , \quad c_1 = 0.535\,[.] \quad , \quad c_2 = 1.05\,[.] \quad , \quad c_3 = 1.15\,[.] \quad , \quad c_4 = 0.8\,[.] \quad . \tag{1.26}$$

We close this section with a number of remarks regarding the coding of the tire model and the simulation program MEDUSA:

Remark 1: Equation $(1.23)_1$ is singular for $s = 1$. Numerically, equations (1.23) are singular if both $s = 0$ and $\alpha = 0$. In the algebraic limit however, F_x and F_y can always be calculated. This is a weakness of the tire model in computations, and appropriate exception routines have been programmed.

Remark 2: It should be clear from Figure 1.3_1, that the lateral tire force F_y needs to satisfy the symmetries $F_y(\alpha) = -F_y(-\alpha)$ and $F_y(\pi/2 - \alpha) = F_y(\pi/2 + \alpha)$. In order to satisfy the latter symmetry, we implemented $|\alpha| > \pi/2 \rightarrow \alpha := \pi - \alpha$ in the computer code.

Remark 3: The curve fit underlying this tire model fails to be accurate for s much smaller than -1, and especially if the tire is sliding backwards (i.e., $s > 1$). In a computer simulation, these maneuvering conditions cannot be excluded a priori. In order to still obtain reasonable tire forces in that range, we implemented $|s| > 1 \rightarrow s := (-1)$ in the computer code.

Remark 4: In reality, vehicle vibrations are absorbed in the compliance of the suspensions and tires. Since tire compliance is absent in most tire models, these vibrations appear as noise in the tire forces. More precisely, they affect the slips (1.19). To amend this problem, we suppress the noise by assuming that the lateral responses of the tires are delayed. This time lag can be described by a first order differential equation with a parameter τ (see Allen et al. [1]). The four delayed or lagged slip angles α_{lag} are now governed by:

$$\tau\,\dot{\alpha}_{lag} + \alpha_{lag} = \alpha \quad . \tag{1.27}$$

In the MEDUSA-code, these lagged side slip angles α_{lag} are substituted for α in the tire force calculations.

1.2.3 Differential Equations of Motion

For the sake of computational efficiency, it is convenient to define the vector components of $\mathbf{r}$, $\mathbf{d}_i$, $\mathbf{k}^i$ (given by equation (1.12)) and $\mathbf{f}^q$ (given by equation (1.16)) with respect to the basis $\{\mathbf{E}_i\}$ and to introduce the generalized position vector $\mathbf{z}_1$, the generalized velocity vector $\mathbf{z}_2$, the generalized slip angle vector $\boldsymbol{\alpha}$, the intrinsic force component vector $\mathbf{k}$, the applied force component vector $\mathbf{f}$ and the body force component vector $\mathbf{u}$ as follows:

$$r_j = \mathbf{r}\cdot\mathbf{E}_j \quad , \quad d_{ij} = \mathbf{d}_i\cdot\mathbf{E}_j \quad , \quad k^i_j = \mathbf{k}^i\cdot\mathbf{E}_j \quad , \quad f^q_j = \mathbf{f}^q\cdot\mathbf{E}_j \quad , \tag{1.28}$$

$$\begin{aligned}
(\mathbf{z}_1) &= (r_1\,,\, r_2\,,\, r_3\,,\, d_{11}\,,\, d_{12}\,,\, d_{13}\,,\, d_{21}\,,\, d_{22}\,,\, d_{23}\,,\, d_{31}\,,\, d_{32}\,,\, d_{33})^T \ ,\\
(\mathbf{z}_2) &= (v_1\,,\, v_2\,,\, v_3\,,\, w_{11}\,,\, w_{12}\,,\, w_{13}\,,\, w_{21}\,,\, w_{22}\,,\, w_{23}\,,\, w_{31}\,,\, w_{32}\,,\, w_{33})^T \ ,\\
(\boldsymbol{\alpha}) &= (\alpha_1\,,\, \alpha_2\,,\, \alpha_3\,,\, \alpha_4)^T \ ,\\
(\mathbf{k}) &= \left(0\,,\,0\,,\,0\,,\, k^1_1\,,\, k^1_2\,,\, k^1_3\,,\, k^2_1\,,\, k^2_2\,,\, k^2_3\,,\, k^3_1\,,\, k^3_2\,,\, k^3_3\right)^T \ ,\\
(\mathbf{f}) &= \left(f^1_1\,,\, f^1_2\,,\, f^1_3\,,\, f^2_1\,,\, f^2_2\,,\, f^2_3\,,\, f^3_1\,,\, f^3_2\,,\, f^3_3\,,\, f^4_1\,,\, f^4_2\,,\, f^4_3\right)^T \ ,\\
(\mathbf{u}) &= (0\,,\,0\,,\,-mg\,,\,0\,,\,0\,,\,0\,,\,0\,,\,0\,,\,0\,,\,0\,,\,0\,,\,0)^T \ .
\end{aligned} \tag{1.29}$$

The equations (1.7) can now be written as a set of first-order ordinary differential equations:

$$\begin{aligned}
\dot{\mathbf{z}}_1 &= \mathbf{z}_2 \quad ,\\
\dot{\mathbf{z}}_2 &= \mathbf{M}^{-1}\left[-\mathbf{k}(\mathbf{z}_1) + \mathbf{A}\mathbf{f}(\mathbf{z}_1,\mathbf{z}_2,\boldsymbol{\alpha}_{lag},t) + \mathbf{u}\right] \ = \ \mathbf{g}(\mathbf{z}_1,\mathbf{z}_2,\boldsymbol{\alpha}_{lag},\mathbf{u},t) \quad ,\\
\dot{\boldsymbol{\alpha}}_{lag} &= \tau^{-1}\boldsymbol{\alpha}_{lag} + \boldsymbol{\alpha} \quad ,
\end{aligned} \tag{1.30}$$

where the inertia matrix $\mathbf{M}$ and the influence matrix $\mathbf{A}$ are given by

$$(\mathbf{M}) = \begin{pmatrix} m\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & my^{11}\mathbf{I} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & my^{22}\mathbf{I} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & my^{33}\mathbf{I} \end{pmatrix} \ , \ (\mathbf{A}) = \begin{pmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{I}\\ X^1_1\mathbf{I} & X^1_2\mathbf{I} & X^1_3\mathbf{I} & X^1_4\mathbf{I}\\ X^2_1\mathbf{I} & X^2_2\mathbf{I} & X^2_3\mathbf{I} & X^2_4\mathbf{I}\\ X^3_1\mathbf{I} & X^3_2\mathbf{I} & X^3_3\mathbf{I} & X^3_4\mathbf{I} \end{pmatrix} \ , \tag{1.31}$$

respectively. In these equations, $\mathbf{I}$ is the 3×3 identity matrix, the coefficients my^{ii} (no sum on i) are calculated from equation (1.9) and the coordinates X^i_q are defined in equation (1.11) and Figure 1.1.

The program code of MEDUSA makes repeated use of the vector

$$(\mathbf{z}) = \left(\mathbf{z}_1^T\,,\, \mathbf{z}_2^T\,,\, \boldsymbol{\alpha}_{lag}^T\right)^T \tag{1.32}$$

which has 28 components. This vector will be referred to as the state vector of the system (1.30).

1.2.4 Energy

The total energy E of the vehicle model consists of the kinetic energy T, the stored elastic energy $m\psi$ of the Cosserat point, the energies V^q stored in the suspension springs and the gravitational potential U:

$$E = T + m\psi + \sum_{q=1}^{4} V^q + U \quad . \tag{1.33}$$

Using the notation from equations (1.17), (1.29) and (1.30), the energy terms are defined as follows:

$$T = \frac{1}{2}\mathbf{z}_2 \cdot (\mathbf{M}\mathbf{z}_2) \quad , \quad U = mgr_3 \quad ,$$

$$m\psi = \frac{V}{2}\left(\lambda\varepsilon_{jj}^2 + 2\mu\varepsilon_{mn}\varepsilon_{mn}\right) \quad , \quad \varepsilon_{ij} = \frac{1}{2}(\mathbf{d}_i \cdot \mathbf{d}_j - \delta_{ij}) \quad ,$$

$$V^q = \frac{1}{2}C^q\left((r_3 + X_q^i d_{i3} - \Delta s\right)^2 \quad , \quad C^q = \begin{cases} C_1 & : \; q = 1,2 \\ C_2 & : \; q = 3,4 \end{cases} \quad . \tag{1.34}$$

Note that the expression for $m\psi$ is consistent with equation (1.12) (see O'Reilly, Papadopoulos, Lo and Varadi [20]).

1.3 The Road Model

In order to study the response of vehicular motion due to excitation from a highway, the flat road assumption originally used in MEDUSA was generalized. In particular, an interpolation method is now used to model sloping and banking of the road. First, we consider the road to be a continuous fixed curve in three-dimensional space. To each point of this curve, a continuously changing banking angle is associated. Using the banking angle and the tangent vector to the curve, the normal of the road plane can be calculated.

The curve is specified by supplying a number of its points in three-dimensional space. In order to avoid the oscillatory behavior that is characteristic of high-degree polynomial interpolation, the cubic spline function (see, e.g., Bartels *et al.* [3]) was chosen as a tool to approximate the road geometry. This selection also has the advantage of not requiring information on derivatives at each of the interior points. The cubic spline interpolation ensures not only a continuous curve passing through each point but also the continuity of first as well as second derivatives at the joints of each successive segment.

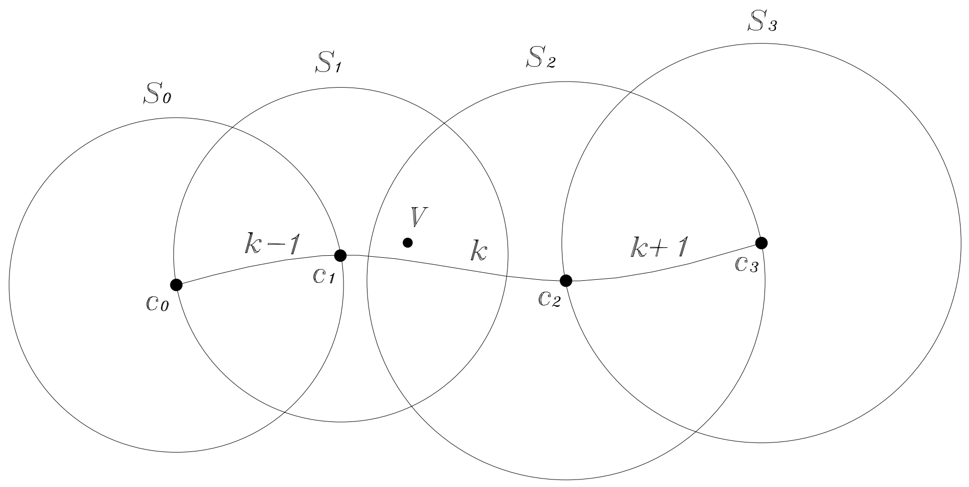

Figure 1.4: *Road spline, spheres and mass center of a vehicle (V)*

1.3.1 Interpolating the Road

We now briefly review the interpolation method. Suppose that one wants to interpolate $h+1$ points in space which have Cartesian coordinates, x_i, y_i, z_i, $i = 1 \ldots, h+1$. Each of the h segments between these points can be represented parametrically as $(X_i(s), Y_i(s), Z_i(s))$. Since $X_i(s)$ is determined solely by the x-coordinates of the points, so too are $Y_i(s)$ and $Z_i(s)$. In the interests of brevity, we will only discuss $Y_i(s)$ which is assumed to be a cubic polynomial specified by four coefficients:

$$Y_i(s) = a_i + b_i\, s + d_i\, s^2 + e_i\, s^3 \quad , \quad i = 1 \ldots, h \quad . \tag{1.35}$$

Here $s \in [0,\ 1]$ is a real-valued parameter. Clearly, we have $4h$ unknown constants to determine. At each of the $h-1$ interior points, we have four conditions representing continuity of zeroth, first and second derivatives:

$$Y_{i-1}(1) = y_i \quad , \quad Y'_{i-1}(1) = Y'_i(0) \quad , \quad Y_i(0) = y_i \quad , \quad Y''_{i-1}(1) = Y''_i(0) \quad . \tag{1.36}$$

Since we also require that $Y_1(0) = y_0$ and $Y_m(1) = y_m$, there are $4h-2$ conditions from which to determine $4h$ unknowns. Thus, two more conditions are needed to ensure that $Y_i(s)$ is a unique interpolating spline.

To overcome this indeterminacy, we employ the first four points specified by the user to construct a cubic polynomial which passes through them. Using the cubic polynomial, we then calculate the first and second derivatives of Y, say, with respect to s at the fourth point. This provides the two new conditions to remove the indeterminacy.

1.3.2 The Road Plane

After one has calculated the road spline, the unit tangent vector $\mathbf{e}_1$ at each point of the spline can be calculated. Using the banking angle and the tangent vector, the road normal vector $\mathbf{e}_3$ can be calculated. Finally, the orthonormal triad $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ can be determined. This triad was referred to earlier in Section 1.2.2.

1.3.3 The Vehicle and the Road Section

After the continuous spline curve is formed, a procedure of building the correspondence between a vehicle and the curve is needed. The reason is that road information such as elevation and the orientation of road plane is needed by the vehicle suspension models. Suppose a vehicle at time t lies entirely in segment k. Either, the vehicle remains in segment k or moves to one of its neighboring segments, $k-1$ or $k+1$, for the next time step. We construct four spheres, S_0, S_1, S_2, S_3, centered at points c_0, c_1, c_2, c_3, with position vectors (see Figure 1.4)

$$\begin{aligned}
\mathbf{r}_0 &= x_{c_0}\mathbf{E}_1 + y_{c_0}\mathbf{E}_2 + z_{c_0}\mathbf{E}_3 \quad , \\
\mathbf{r}_1 &= x_{c_1}\mathbf{E}_1 + y_{c_1}\mathbf{E}_2 + z_{c_1}\mathbf{E}_3 \quad , \\
\mathbf{r}_2 &= x_{c_2}\mathbf{E}_1 + y_{c_2}\mathbf{E}_2 + z_{c_2}\mathbf{E}_3 \quad , \\
\mathbf{r}_3 &= x_{c_3}\mathbf{E}_1 + y_{c_3}\mathbf{E}_2 + z_{c_3}\mathbf{E}_3 \quad .
\end{aligned} \tag{1.37}$$

The radii R_0, R_1, R_2, R_3, of the spheres are

$$\begin{aligned}
R_0 &= ||\mathbf{r}_0 - \mathbf{r}_1|| \quad , \\
R_1 &= \min(||\mathbf{r}_0 - \mathbf{r}_1||, ||\mathbf{r}_1 - \mathbf{r}_2||) \quad , \\
R_2 &= \min(||\mathbf{r}_1 - \mathbf{r}_2||, ||\mathbf{r}_2 - \mathbf{r}_3||) \quad , \\
R_3 &= ||\mathbf{r}_2 - \mathbf{r}_3|| \quad .
\end{aligned} \tag{1.38}$$

If the spheres S_1 and S_2 do not overlap, a fifth sphere S_4 is added whose center is at the midpoint of the straight line connecting $\mathbf{r}_1$ and $\mathbf{r}_2$ with a radius R_4 which equal to $\frac{1}{2}||\mathbf{r}_2 - \mathbf{r}_1||$. Next, we find which sphere the center of mass of the vehicle lies in. This gives us the segment of the road that the vehicle is assumed to move on at the instant of interest.

Suppose that the mass center of a vehicle is inside S_0 or S_3, then the corresponding curve segment is $k-1$ or $k+1$. Otherwise, there are three cases to consider. In the first of these the vehicle lies in the intersection of S_1 and S_2. Secondly, the vehicle may lie in S_4. The last case arises when the vehicle is only in S_1 or S_2. For the third case where the vehicle lies only in S_1 the vector from $\mathbf{r}_1$ to $\mathbf{r}$ is calculated. The dot-product of $\mathbf{r} - \mathbf{r}_1$ and $\mathbf{t}_{c_1}$, tangent vector of the road spline at point c_1, is calculated. If this dot product is positive, then the vehicle lies in the kth segment, otherwise it is defined to lie in the k-1th segment. For the third case where the vehicle lies only in S_2 the vector from $\mathbf{r}_2$ to $\mathbf{r}$ is calculated. The dot product of $\mathbf{r} - \mathbf{r}_2$ and $\mathbf{t}_{c_2}$ is calculated. If this dot product is positive, then the vehicle lies in the k+1-st segment, otherwise it is defined to lie in the k-th segment. In either of these third cases, if the dot product is zero, then the vehicle is directly over a node point. Clearly, for the first and second cases, the vehicle remains in the kth segment.

If a vehicle is not inside any one of the five spheres, it is considered out of road range. Notice that a virtual flat road segment is added ahead of the first nodal point to prevent vehicles accidentally moving out of road range at the beginning stages of a simulation. Once the segment of the curve where the vehicle lies is obtained, the corresponding point of a vehicle on the road curve is defined to be the closest point of the vehicle to that segment.

1.4 Contact Constraints and Contact Forces

When two vehicles come into contact, contact forces associated with the constraint of impenetrability (see O'Reilly and Varadi [24]) prevent the vehicles from interpenetrating. During contact, the position vectors and directors of the vehicles depend on each other and so do the individual equations of motion. For simplicity, rather than algebraically eliminating the dependent kinematic quantities from the equations of motion using the constraint equations, we adopt a numerical scheme in MEDUSA based on a normality assumption using Lagrange multipliers. Here, the normality assumption presumes frictionless contact. This numerical approximation allows the surfaces of the vehicles to overlap by a small amount. The resulting contact forces are then reminiscent physically to those resulting from the compression of a linearly elastic spring.

Consider now the situation depicted in Figure 1.5. Assume we have determined two points $\mathbf{r}^{\sigma}_{(1)}$ and $\mathbf{r}^{\sigma}_{(2)}$ (cf. equation (1.5)) on the surfaces of the respective vehicles which we may call contact points. Let $\mathbf{n}_{(1)}$ be the outward unit normal to the surface of vehicle one. The distance function

$$\begin{aligned} \phi_1 &= (\mathbf{r}^{\sigma}_{(2)} - \mathbf{r}^{\sigma}_{(1)}) \cdot \mathbf{n}_{(1)} \\ &= \left(\mathbf{r}_{(2)} + X^i_{\sigma(2)}\, \mathbf{d}_{(2),i} - \mathbf{r}_{(1)} - X^i_{\sigma(1)}\, \mathbf{d}_{(1),i}\right) \cdot \mathbf{n}_{(1)} = \hat{\phi}_1(\mathbf{z}_{(1),1}, \mathbf{z}_{(2),1}) \end{aligned} \tag{1.39}$$

quantifies separation ($\phi_1 > 0$), contact ($\phi_1 = 0$) or penetration ($\phi_1 < 0$). The sign of a second function ϕ_2 indicates the relative normal velocity of the contact points:

$$\begin{aligned} \phi_2 &= (\overset{*}{\mathbf{r}}{}^{\sigma}_{(2)} - \dot{\mathbf{r}}^{\sigma}_{(1)}) \cdot \mathbf{n}_{(1)} \\ &= \hat{\phi}_2(\mathbf{z}_{(1),1}, \mathbf{z}_{(1),2}, \mathbf{z}_{(2),1}, \mathbf{z}_{(2),2}) \ . \end{aligned} \tag{1.40}$$

Here $\overset{*}{\mathbf{r}}{}^{\sigma}_{(2)}$ is the rate of change of the position vector of the particle which occupies $\mathbf{r}^{\sigma}_{(2)}$ at time t. These two functions generate constraint conditions as they should be zero when the vehicles are in contact.

During times of contact (defined by $\phi_1 \leq 0$), we modify the equations of motion (1.30) of the two vehicles $\beta = 1, 2$ as follows:

$$\begin{aligned} \dot{\mathbf{z}}_{(\beta),1} &= \mathbf{z}_{(\beta),2} + \mathbf{c}_{(\beta),1} \ , \\ \dot{\mathbf{z}}_{(\beta),2} &= \mathbf{g}_{(\beta)} + \mathbf{c}_{(\beta),2} \ , \\ \dot{\boldsymbol{\alpha}}_{(\beta),lag} &= \tau^{-1}_{(\beta)} \boldsymbol{\alpha}_{(\beta),lag} + \boldsymbol{\alpha}_{(\beta)} \ . \end{aligned} \tag{1.41}$$

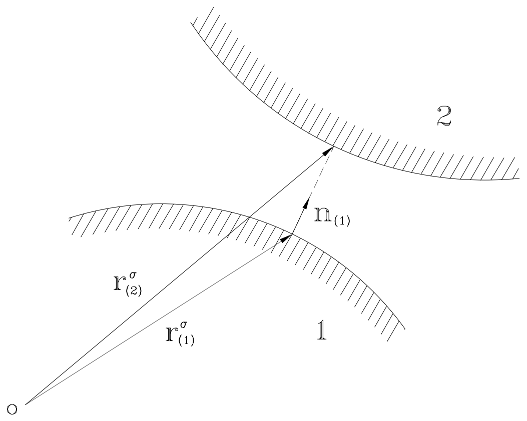

Figure 1.5: *Schematic depiction of the kinematical quantities involved in describing the contact between two bodies.*

We will refer to $\mathbf{c}_{(\beta),1}$ and $\mathbf{c}_{(\beta),2}$ as the (generalized) contact forces. They are calculated from the constraints (1.39) and (1.40) using a normality prescription:

$$\mathbf{c}_{(\beta),1} = \gamma_1 \frac{\partial \hat{\phi}_1}{\partial \mathbf{z}_{(\beta),1}} , \qquad \mathbf{c}_{(\beta),2} = \gamma_2 \frac{\partial \hat{\phi}_2}{\partial \mathbf{z}_{(\beta),2}} . \tag{1.42}$$

In this equation, the factors γ_1 and γ_2 are Lagrange multipliers that are determined by the motion. We will approximate them using a numerical scheme that will be explained in Section 1.6. In the interest of brevity, we do not write the forces (1.42) in a component form similar to (1.29). Note however that the contact forces are functions of $X^i_{\sigma(\beta)}$ and $\mathbf{n}_{(1)}$.

We also remark that a vehicle may be in contact with several other vehicles at a given instant. In this case, the above procedure is repeated for each pair of vehicles, and requires proper book-keeping of the various contact constraints and contact forces.

1.5 Contact Detection

In the previous section, we assumed *a priori* knowledge of the position vectors $\mathbf{r}^{\sigma}_{(\beta)}$ ($\beta = 1, 2$) and the surface normal vector $\mathbf{n}_{(1)}$ at the (potential) point of contact. In this section, we are going to outline a procedure that yields these quantities uniquely. Although the ideas presented here apply to general contact problems involving convex surfaces, we will restrict our discussion to special surfaces known as superellipsoidal surfaces which is discussed in Section 1.5.1.

Note that if the vehicles were modeled as spheres of radii $\rho_{(\beta)}$, the condition

$$||\mathbf{r}_{(2)} - \mathbf{r}_{(1)}|| \leq \rho_{(1)} + \rho_{(2)} \tag{1.43}$$

would be sufficient to determine whether the two vehicles are in mutual contact. The general situation is however far more complicated. Nevertheless, this simple idea can still be used as a preliminary fast test for contact while the vehicles are relatively far apart[3].

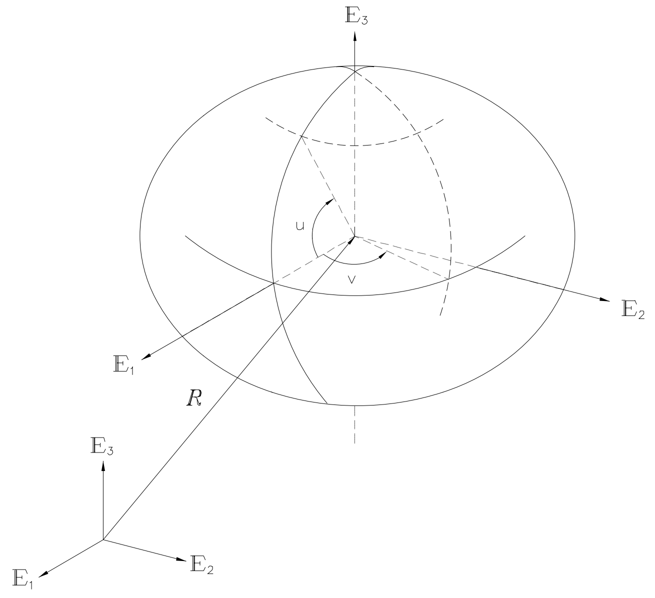

Figure 1.6: *Parametric representation of the vehicle superellipsoid in the reference configuration.*

1.5.1 A New Vehicle Geometry

In a previous report O'Reilly, Papadopoulos, Lo and Varadi [22], the vehicle geometry was approximated by an ellipsoid. The related contact information was determined using information form the surfaces of contacting ellipsoids. An obvious drawback arising from the differences between the outer surfaces of an actual vehicle and an ellipsoid is that it can result in different post-contact motions in some situations, e.g., in the offset head-tail collision of

[3]In MEDUSA, this idea has been implemented with $\rho = \max(A, B, C)$, where A, B and C are defined in equation (1.44).

two vehicles. This defect was partially amended by an improved contact-detection algorithm discussed in O'Reilly, Papadopoulos, Lo and Varadi [23]. However, the geometric mismatch between the actual vehicle and an ellipsoid remained, especially near and at the corners of a vehicle.

The outer surface of a vehicle is similar to a box, however a convex surface is required by the contact-detection algorithm. Thus, for the purpose of fixing the mismatch in geometric shape, the vehicle geometry is modeled here as a type of superquadric, specifically a superellipsoid. We recall that the surface of an ellipsoid has the parametric representation

$$\mathbf{R} = A\cos(u)\cos(v)\mathbf{E}_1 \ + \ B\cos(u)\sin(v)\mathbf{E}_2 \ + \ C\sin(u)\mathbf{E}_3 \quad , \tag{1.44}$$

where A, B and C are the lengths of the semi-axes and $\mathbf{E}_i$, $i = 1..,\ 3$ are the directions which are parallel to the ellipsoid in the reference configuration. The parameters $u \in [-\pi/2, \pi/2]$, $v \in [0, 2\pi)$ are curvilinear surface coordinates. The parametric form of a superquadric, specifically a superellipsoid, is defined as

$$\mathbf{R} = A\cos(u)^{\epsilon_1}\cos(v)^{\epsilon_2}\mathbf{E}_1 \ + \ B\cos(u)^{\epsilon_1}\sin(v)^{\epsilon_2}\mathbf{E}_2 \ + \ C\sin(u)^{\epsilon_1}\mathbf{E}_3 \quad , \tag{1.45}$$

where ϵ_1, ϵ_1 are the squareness parameters in the longitudinal and lateral directions, respectively (see, e.g., Barr [2])[4]. In the version of Medusa discussed here, the superellipsoid is used to model the outer (lateral) surface of the vehicle.

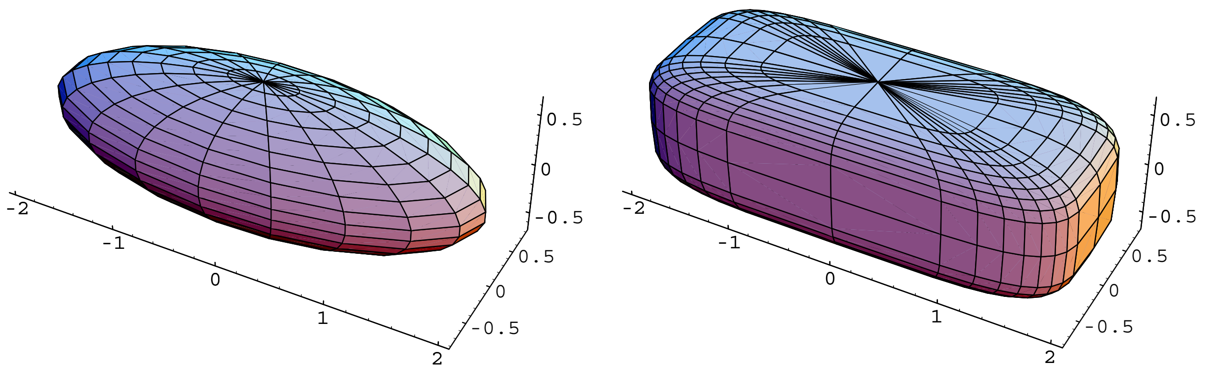

Figure 1.7: *Comparison of an ellipsoid (*$\epsilon_1 = \epsilon_2 = 1.0$*) and a superellipsoid (*$\epsilon_1 = \epsilon_2 = 0.4$*).*

Under the action of the deformation gradient $\mathbf{F}(t)$ defined in equation (1.3), the material surface σ defined in equation (1.44) subsequently deforms into a surface which is described by equation (1.4). In Medusa, the program uses the ellipsoid to determine the deformation of the vehicle, and the deformed ellipsoid is then used to generate a superellipsoidal surface

[4]The parameters ϵ_1 and ϵ_1 are chosen to be equal to 0.4 in the Medusa simulations (see Figure 1.7) to best fit the shape of an actual vehicle.

which is then used by the contact detection algorithm. The corresponding superellipsoid can be constructed using a unique mapping in which the ellipsoid and the superellipsoid share the same semi-axes and principal directions and the two parameters ϵ_1 and ϵ_2 reshape the outer surface of the original ellipsoid.

Finally, the tangent vectors and the outward surface normal vector in the present configuration are given by

$$\mathbf{a}_u = \frac{\partial \mathbf{r}^\sigma}{\partial u} \quad , \quad \mathbf{a}_v = \frac{\partial \mathbf{r}^\sigma}{\partial v} \quad , \quad \mathbf{n} = \frac{\mathbf{a}_v \times \mathbf{a}_u}{||\mathbf{a}_v \times \mathbf{a}_u||} \quad . \tag{1.46}$$

1.5.2 Minimum Distance Searching Scheme

Recall from equation (1.5) that the vectors $\mathbf{r}^\sigma_{(\beta)}$ $(\beta = 1, 2)$ denote material points on the surface of the respective superellipsoids. Consider the distance function

$$f = ||\mathbf{r}^\sigma_{(1)} - \mathbf{r}^\sigma_{(2)}|| = f(\mathbf{x}) \quad , \quad \mathbf{x} = (u_{(1)}, v_{(1)}, u_{(2)}, v_{(2)}) \quad . \tag{1.47}$$

We would like to define our contact points from Section 1.4 as those points (labeled K for vehicle one and L for vehicle two) for which f attains a global minimum. However, before we embark on finding such a minimum, we should consult Figure 1.8. It is intuitively clear that we will not able to find unique points K and L when the vehicles are interpenetrating, in which case $\min(f) = 0$ on the intersection curve. This case needs some additional work which is outlined in Section 1.5.3.

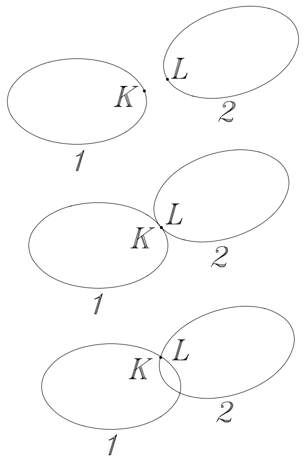

Figure 1.8: *The contact situations between two bodies*

In order to find the points K and L on the respective superellipsoids that minimize the distance function f, we start with two *arbitrary* points K_1 and L_1 on the respective surfaces (see also Figure 1.9). Keeping K_1 fixed, we find a new point L_2 on the surface of the second vehicle which (locally) minimizes f. Keeping L_2 fixed, we find a new point K_2 which again (locally) minimizes f, and so forth. In this manner, we obtain a sequence L_2, K_2, L_3, ...

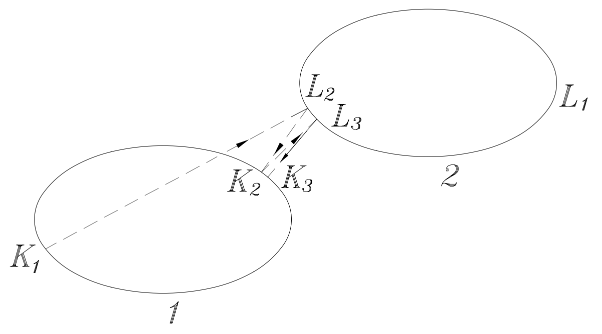

Figure 1.9: *From initial guesses K_1 and L_1 a sequence of points L_2, K_2, L_3, ... is calculated which minimizes the distance function f defined in the text.*

which converges to two points K and L (which will not be unique if there is a curve of intersection).

The algorithm employed in MEDUSA to perform the series of local minimizations of the distance function f is a *variable metric* method whose details are discussed in Appendix A. Note that every one of these minimization steps finds a unique (local) minimum of f. This is due to the fact that, in each step, we minimize the distance from a point to a convex surface (for further details, see Kowalik [16]).

Now consider Figure 1.8 once more. Having previously found the points K and L and the corresponding position vectors $\mathbf{r}^{\sigma}_{(\beta)}$, it is clear that the associated normal vectors $\mathbf{n}_{(\beta)}$ defined in equation (1.46) satisfy[5] $\mathbf{n}_{(1)} \doteq -\mathbf{n}_{(2)}$ only if the superellipsoids are not penetrating. We can therefore specify the following no-contact condition:

$$(\mathbf{r}^{\sigma}_{(2)} - \mathbf{r}^{\sigma}_{(1)}) \cdot \mathbf{n}_{(1)} > 0 \quad , \quad \mathbf{n}_{(1)} \doteq -\mathbf{n}_{(2)} \quad . \tag{1.48}$$

If this test fails, then the vehicles are in contact and additional work needs to be done to determine the unique points of contact for intersecting superellipsoids. We now turn to this matter.

1.5.3 Unique Contact Point Detection

The basic idea of this scheme is to apply a suitable perturbation to K and L which were obtained using the previous minimum distance searching iteration. For the case of interest, the former points may lie anywhere on the curve of intersection of the two superellipsoids. Hence, the points of contact that we seek are the two points of maximum penetration along their common normal[6].

[5]Up to the order of numerical accuracy in finite arithmetic.

[6]Clearly, this is one of many possible ways to define unique contact points. However, it appears that the detection of the maximum penetration points is the easiest method.

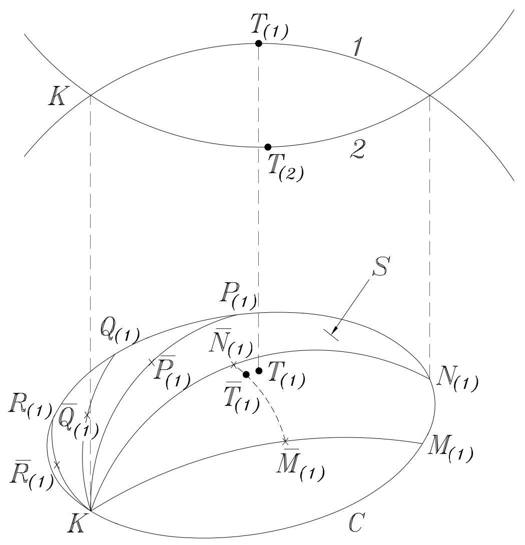

Figure 1.10: *Unique contact point detection scheme*

Recall that the equation of the superellipsoid for each vehicle in its reference configurations has the form (1.45) which can be rewritten as

$$(\mathbf{R}^{\sigma}_{(\beta)} - \mathbf{R}_{(\beta)}) \cdot \mathbf{K}_{(\beta)}(\mathbf{R}^{\sigma}_{(\beta)} - \mathbf{R}_{(\beta)}) = 1 \quad , \quad \beta = 1, 2 \quad , \tag{1.49}$$

where $(\mathbf{K}_{(\beta)}) = \text{diag}(1/A^2_{(\beta)},\ 1/B^2_{(\beta)},\ 1/C^2_{(\beta)})$ is a second order diagonal tensor. With the help of (1.4), the equation of the superellipsoid in the current configuration can be expressed as

$$(\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) \cdot \hat{\mathbf{K}}_{(\beta)}(\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) = 1 \quad , \quad \beta = 1, 2 \quad , \tag{1.50}$$

where $\hat{\mathbf{K}}_{(\beta)} = \mathbf{F}^{-T}_{(\beta)}\mathbf{K}_{(\beta)}\mathbf{F}^{-1}_{(\beta)}$[7]. Thus, the functions of the superellipsoids 1 and 2 in the present configuration are defined as follows:

$$\hat{f}_{(\beta)} = (\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) \cdot \hat{\mathbf{K}}_{(\beta)}(\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) - 1 \quad , \quad \beta = 1, 2 \quad . \tag{1.51}$$

Clearly, $\hat{f}_{(1)} = 0$ for a point on the surface $\sigma_{(1)}$ of superellipsoid 1 and $\hat{f}_{(1)}$ is greater (less) than 0 for a point located outside (inside) of superellipsoid 1.

[7]The principal directions and principal semi-axes of the superellipsoid in the current configuration can be determined by solving the eigenvectors and eigenvalues of $\hat{\mathbf{K}}_{(\beta)}$.

Consider the curve $\mathcal{C}$ which is the intersection of two superellipsoids in three-dimensional Euclidean space and the point K computed using the previous iteration. The point corresponding to the maximum penetration of superellipsoid 1 into the superellipsoid 2, denoted as $T_{(1)}$, is the point on the surface of superellipsoid 1 whose position vector minimizes the function $\hat{f}_{(2)}$. Similarly, the point $T_{(2)}$ can also be used as the maximum penetration point of superellipsoid 2 into superellipsoid 1. The pair of points, $T_{(1)}$ and $T_{(2)}$, will serve as the contact points in the case when the penetration has occurred between two superellipsoids.

The procedure for unique contact point detection is commenced by circling around point K using a small perturbations on the surface of superellipsoid 1. Suppose one picks five distinct points, denoted by $M_{(1)}$, $N_{(1)}$, $P_{(1)}$, $Q_{(1)}$ and $R_{(1)}$, which lie on the curve $\mathcal{C}$. At these points, $\hat{f}_{(2)}$ will be zero. Five curves can be plotted from the point K to each of these points by linearly interpolating between their $u-v$ coordinates. The midpoints of each curve from K to $M_{(1)}$, $N_{(1)}$, $P_{(1)}$, $Q_{(1)}$ and $R_{(1)}$ are denoted by $\bar{M}_{(1)}$, $\bar{N}_{(1)}$, $\bar{P}_{(1)}$, $\bar{Q}_{(1)}$ and $\bar{R}_{(1)}$, respectively. If $\bar{N}_{(1)}$ is the point where $\hat{f}_{(2)}$ is minimal among the five midpoints, and if $\hat{f}_{(2)}$ decreases along the curve $\bar{N}_{(1)}\bar{M}_{(1)}$, then $T_{(1)}$, the point of maximum penetration, can be found by first searching along the curve connecting $\bar{N}_{(1)}$ and $\bar{M}_{(1)}$ and then by searching along the curve connecting K and $\bar{T}_{(1)}$. The corresponding point of maximum penetration, $T_{(2)}$, of superellipsoid 2 into superellipsoid 1 can be obtained using a similar procedure.

1.6 Time Integration

Classical explicit time integration methods have proven to be unsatisfactory when solving the equations of motion (1.30) and (1.41) even when used with an adaptive step-size control. Essentially, the required step size of integration is far too small to be of practical use. The reason for this lies in the intrinsic director forces $\mathbf{k}^i$ defined in equation (1.12), which produce very high frequency modes of oscillation. Implicit integration methods on the other hand can use much larger time steps at the expense of solving (implicit) systems of algebraic equations.

In this version of MEDUSA, we employ a simple explicit predictor-corrector integration scheme based on the forward Cauchy-Euler method. We outline here the integration scheme for equation (1.41) when two vehicles are in contact. The corresponding schemes for equation (1.30), and for any additional vehicles, are easily inferred. To simplify the notation, we suppress the vehicle index β here:

$$\begin{aligned}
\tilde{\mathbf{z}}_{1,k+1} &= \mathbf{z}_{1,k} + \Delta t\, \mathbf{z}_{2,k} \ , \\
\gamma_{1,k+1} &= \gamma_{1,k} + p_1\, \phi_1(\tilde{\mathbf{z}}_{1,k+1}) \ , \\
\gamma_{2,k+1} &= \gamma_{2,k} + p_2\, \phi_2(\tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k}) \ , \\
\mathbf{z}_{1,k+1} &= \mathbf{z}_{1,k} + \Delta t\, [\mathbf{z}_{2,k} + \mathbf{c}_1(\gamma_{1,k+1}, \tilde{\mathbf{z}}_{1,k+1})] \ , \\
\mathbf{z}_{2,k+1} &= \mathbf{z}_{2,k} + \Delta t\, [\mathbf{g}(\tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k}, \boldsymbol{\alpha}_{lag,k}, t) + \mathbf{c}_2(\gamma_{2,k+1}, \tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k})] \ , \\
\boldsymbol{\alpha}_{lag,k+1} &= \boldsymbol{\alpha}_{lag,k} + \Delta t\, \left[\tau^{-1}\, \boldsymbol{\alpha}_{lag,k} + \boldsymbol{\alpha}(\tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k})\right] \ , \qquad (1.52)
\end{aligned}$$

where quantities of the form x_k are approximations of $x(t_k)$. In these incremental equations, Δt is the step size. Equation $(1.52)_1$ uses the forward Cauchy-Euler method to calculate $\tilde{\mathbf{z}}_{1,k+1}$ which serves as a predictor of $\mathbf{z}_{1,k+1}$. All of the other equations of (1.52) use this prediction. Equations $(1.52)_{4,5,6}$ are the forward Cauchy-Euler integrations steps of equation (1.41). The use of the predictor $\tilde{\mathbf{z}}_{1,k+1}$ mainly affects the function $\mathbf{g}$ in $(1.52)_5$ since it contains the stiffest part of the equations of motion (i.e., the intrinsic director forces).

Equations $(1.52)_{2,3}$ compute approximations to the Lagrange multipliers of equation (1.42) by penalizing the constraint functions ϕ_1 and ϕ_2. The penalty parameters p_1 and p_2 are constants that must be properly chosen. When a vehicle is not in contact with another, then clearly $\mathbf{c}_1 = \mathbf{0}$ and $\mathbf{c}_2 = \mathbf{0}$. We then also set $\gamma_1 = \gamma_2 = 0$, which serves as initial conditions for $(1.52)_{2,3}$ whenever contact is initiated.

We close this chapter with a remark concerning Δt and the transitions to and from equations (1.30) and equations (1.41) in Medusa. Whenever $\tilde{\mathbf{z}}_{1,k+1}$ predicts contact between a pair of vehicles in the next time step, the step size Δt is reduced in order to keep the penetration of the vehicles small. The step size is however not reduced below some minimal value which is hardwired into the code (see Programmer's Guide and Appendix F). At this point, the integration continues with the fixed smallest step size. Once contact is lost (i.e., $\phi_1 > 0$), the Lagrange multipliers γ_1 and γ_2 are reset to zero. If contact does not reoccur during a certain (hardwired) time interval, the step size is increased again to the maximal step size which is a quantity provided by the user.

1.7 Collisions and Dissipation

During an actual collision, dissipation can be present due to the irreversible deformation of the vehicle. For collision with relative velocity up to approximately $5\,[mp/h]$, dissipation is exclusively due to the bumper design. For higher relative velocities or side impacts, dissipation occurs due to the irreversible deformation of the body structure.

To incorporate irreversible deformations and their associated dissipation into the vehicle model, it suffices to augment the contact forces $\mathbf{c}_1$ and $\mathbf{c}_2$ (see (1.42)) with rate-dependent terms. This is conveniently accomplished by selecting the penalty coefficients p_1 and p_2 (see $(1.52)_{2,3}$) so that they effectively act as viscocity parameters. There coefficients may be chosen independently for frontal and lateral impacts to differentiate between bumpers and the weaker side structure. We refer the reader to Section 4.7 of the Programmer's Guide for pertinent details.

Chapter 2

User's Manual

2.1 Introduction

The MEDUSA-code has the capability of simulating arbitrarily many vehicles driving on a road free of obstacles. The program allows vehicular collisions with moderately high relative velocities. Consequently, MEDUSA can be used to qualitatively investigate collision scenarios that occur within platoons of vehicles. In this chapter, we explain how to run MEDUSA. This task involves no changes in the program code of MEDUSA. For details on such changes, we refer the reader to the Programmer's Guide in Chapter 4. There, possible modifications of the mathematical models of the vehicles and the road are discussed. The reader is also referred to Chapter 3, where several examples are discussed in detail.

Note that whenever we use the terms 'vehicle model' or simply 'model' in this chapter, we mean a set of parameters that is required by the underlying mathematical model. Thus, if we say that two vehicles have different models, we mean that their model parameters are different. Their mathematical model is however the same since it is hardwired into the program code.

The usage of MEDUSA is quite simple. The user provides the vehicle models, the initial positions, orientations and velocities of the vehicles and the road data in three separate input files. These files are written in a plain text format whose contents are explained in Section 2.2. The simulation is run by typing a line similar to

```
>>medusa -t1.0
```

at the command prompt of the operating system[1]. The simulation is controlled with command line options (such as `-t1.0` in the example above). These options are explained in Section 2.3. MEDUSA writes the simulation data into several plain text files that can be read and graphically presented by other programs such as MATLAB, MATHEMATICA or the SMARTPATH-animator. We discuss this in Section 2.4.

[1]In this chapter, we will assume that the command prompt is >> and that the executable is named *Medusa*.

2.2 The Three Input Files

There are three input files that the user needs to provide to run a simulation which by default are called *model.dat*, *platoon.dat*, and *road.dat*. Out of these, only the file *platoon.dat* may have an alternative name. It can be specified when running the program (see Section 2.3).

The structure of these plain text files is defined as a sequence of mandatory keywords, or tokens. If keywords are missing or misspelled, a run-time error message is generated. The keywords for each file are listed in their proper order below. Sample files can be found in Chapter 3 and Appendix C for reference. Comments can be placed anywhere using the symbol `%`. The rest of the line is then ignored. Note that MEDUSA is case-insensitive, i.e., the keywords `MODEL`, `Model` or `MoDeL` are considered identical. In this chapter however, we will use capital letters to highlight keywords. Small italic letters are to be replaced by a number.

model.dat This file provides MEDUSA with a database of vehicle models. The models in this file are listed sequentially and are numbered starting from one. Later, models will be assigned to vehicles simply by choosing the appropriate model number. The file *model.dat* must contain at least one set of model parameters.

`NUMBER_OF_MODELS` m This keyword appears only once at the top of the file. MEDUSA will subsequently read m vehicle models from the file or generate an error message if less than m models are found.

`MODEL` m Beginning of model m. The first model in the file is number `1`, the second `2` and so on. An error is generated if the sequence is different.

For each model, the data is grouped for the Cosserat point, the suspension, the tire model and the contact model. An additional data group defines an equilibrium of the vehicle which will be used later to define initial conditions for the simulation.

`COSSERAT_POINT` This keyword groups data pertaining to the Cosserat point:

`MASS` x The mass of the vehicle $[kg]$.

`IX` x `IY` y `IZ` z The principal moments of inertia J_0^{11}, J_0^{22} and J_0^{33} of equation (1.9) $[kg\,m^2]$.

`E` e `NU` u `VOLUME` v Young's modulus $[N/m^2]$, Poisson's ratio and the material volume of the chassis $[m^3]$.

`SUSPENSION` This keyword groups data pertaining to the suspension models:

`L1` u `L2` v `B` x `H1` y `H2` z These coordinates are defined in Figure 1.1 $[m]$.

`SPRING_REF` x The unstretched length of the suspension springs (Δs in equation (1.17)) $[m]$.

`C1` x `C2` y Front and rear spring constants $[N/m]$.

`D1` x `D2` y Front and rear viscous damping coefficients $[Ns/m]$.

`TIRE` x Tire lag parameter (τ in equation (1.27)).

`CONTACT A1` x `A2` y `A3` z The three semi-axes of the super-ellipsoid that approximates the vehicle's outer geometry.

`EQUILIBRIUM` This data group defines a reference state (typically an equilibrium) of the vehicle. The initial conditions of the vehicle are defined relative to this state.

- `R3` r Height of the vehicle's center of mass above ground $[m]$
- `D11` x `D12` y `D13` z Components of director 1 at the equilibrium [.]
- `D21` x `D22` y `D23` z Components of director 2 at the equilibrium [.]
- `D31` x `D32` y `D33` z Components of director 3 at the equilibrium [.]

Additional models are added starting again with the keyword `MODEL` and then following the above sequence of keywords. If the reference state for the `EQUILIBRIUM`-section is not known, it can easily be obtained by simulating a single vehicle on a straight horizontal road for a period of time. The vehicle will settle into the desired state after all of its vibrations are damped out.

platoon.dat This file contains the description of the vehicle platoon. As mentioned earlier, a different filename may be specified when running the program (see Section 2.3).

`NUMBER_OF_VEHICLES` v This keyword appears only once at the top and Medusa will subsequently read the definitions for v vehicles from the file or generate an error message if less vehicles are found.

`VEHICLE_HAS_MODEL` m The vehicle is assigned the model m from the database *model.dat*.

`INITIALLY_WITH` Defines initial conditions for the vehicle:

- `X` x `Y` y The initial position of the vehicle's center of mass $[m]$.
- `ORIENTATION` **u** Defines the initial direction in which the vehicle is heading $[deg]$. The angle is measured about $\mathbf{E}_3$ counter-clockwise from $\mathbf{E}_1$. For example, the value $\frac{\pi}{2}$ corresponds to a heading in the $\mathbf{E}_2$ direction.
- `SPEED` v Initial speed of the vehicle $[m/s]$.
- `TIRE_SPEED` v Circumferential tire speed of the driven front wheels in $[m/s]$ ($R\omega$ in equation $(1.19)_1$). The rear wheels roll free. If v is zero, the front wheels roll free, too.
- `STEERING` x Constant steering angle [deg]. The value `0` causes the vehicle to drive straight. A positive value makes the vehicle turn to the left (counter-clockwise, as viewed from the driver's perspective.

Medusa numbers the vehicles so that they can be identified in the output data. The first vehicle in the platoon description file is number 1 and so forth.

road.dat This is the road description file for MEDUSA. It is used to construct a fixed road curve in three-dimensional space, and, at each point of this curve, a road plane. Data points are chosen by the user to best approximate the road of interest.

It is important that this file should contains at least four data points (see Section 1.3). Furthermore, the distance between any two neighboring data points should be several times larger than a vehicle's greatest dimension. We also note that if the data points are chosen to be too closely spaced, then the spline interpolation method used to generate a road will lead to spurious elevation changes.

`NUMBER_OF_POINTS:` n The number of data points used to define the road.

`XYZ_COORDINATES_&_ANGLE:` The Cartesian coordinates $[m]$ and banking angles $[deg]$ of each road point.

`END_OF_FILE` No more data points after this keyword.

2.3 Running the Program

After one has written the model database file *model.dat*, the platoon description file (e.g., *platoon.dat*) and the road description file *road.dat*, suppose that one wishes to run a simulation. Try:

```
>>medusa
No endtime specified. Type medusa -h for help
```

The program stopped execution because no simulation parameters were specified. MEDUSA suggests one uses its help feature:

```
>>medusa -h

The command line options are:

  -h  prints this list
  -dx.xxx  set fixed stepsize to x.xxx [s] (default: 5e-05)
  -ffile  parameter file (default: platoon.dat)
  -tx.xx  simulation ends at x.xx [s] (mandatory)
  -sx.xx  save data point every x.xx [s] (default: 0.01)
```

We will explain these options in a moment. Note that the option letters are case-sensitive and they each start with a dash.

Some of the options have additional parameters. There may be no space between the option letter and the parameter. If options are misspelled, a run-time error message is produced. The sequence of options does not matter. Number parameters may be written in the normal floating point format, e.g., `0.015` or `1.5E-2` are equally valid.

`-d`*x* Step size of the time integrator. Without this option, the stepsize is set to the default as indicated by the help feature. Note that at times when vehicles are in contact with each other, the stepsize is reduced to a pre-programmed value.

`-f`*name* By default, *platoon.dat* is the platoon description file. Through this option, a different file may be chosen.

`-h` This prints the help screen above.

`-s`*x* Data points are stored in fixed time intervals. Without this option, a default value is used.

`-t`*x* The simulation runs for *x* seconds. This parameter is mandatory.

We now look at a few examples. Assume that we have written the files *model.dat* and *platoon.dat* and that we would like to simulate the vehicles for ten seconds with a fixed stepsize of 10^{-5} seconds. We wish to obtain data every 0.2 seconds. The command that achieves this is:

```
>>medusa -t10 -s.2 -d1e-5
```

Assume now that we have named the platoon description file *crash_it*. We wish to simulate the vehicles for one second with the default integration stepsize:

```
>>medusa -fcrash_it -t1
```

At execution time, MEDUSA displays some information on the screen. If one runs the program using the sample *platoon.dat* from Chapter 3, the screen will look something like this:

```
>>medusa -t10

The simulation for 2 vehicles will stop after 10 [s]
       - stepsize: 5e-0.5 [s]
       - save data point every 0.01 [s]

         t=2.1300
```

A counter at the bottom of the screen will show the progress of the simulation by displaying time in seconds. We will explain the format of the output files in further detail in the next section.

2.4 Output Files

As output, the simulation produces several plain text files: *path.asc*, *director.asc*, *velocity.asc*, *energy.asc* and *oop.asc*. These files contain the numerical simulation data as columns of floating point numbers so that the files can easily be imported into other software packages such as MATLAB or MATHEMATICA. The first column in each file records elapsed time, the remaining column record certain quantities at a given time.

path.asc This file contains the simulation data in a form suitable for import into the SMART-PATH animator. We refer to Eskafi, Khorramabadi and Varaiya [8] for an explanation of the data format.

director.asc Following the time column, there are twelve data columns for vehicle one, twelve for vehicle 2, and so forth. Out of these twelve columns, the first three record the three components $\mathbf{r}\cdot\mathbf{E}_i$ $(i = 1, 2, 3)$ of the vehicle's position vector $\mathbf{r}$. The remaining columns record the vehicle's three directors $\mathbf{d}_i$, each with three components $\mathbf{d}_i \cdot \mathbf{E}_j$ (cf. equation (1.28)).

velocity.asc This file has the same structure as *director.asc*, with the exception that instead of position vectors and directors for all vehicles, velocity $\mathbf{v}$ and director velocities $\mathbf{w}_i \cdot \mathbf{E}_j$ are recorded.

energy.asc Following the time column, there is one column for each vehicle that records the vehicle's total energy.

oop.asc This file exists mostly for the purpose of debugging the program. It is empty unless a programmer wants to use it as a scratch file (see Programmer's Guide in the next chapter).

Chapter 3

Simulations using Medusa

An accident on a busy automated highway is a highly complex dynamic event which may involve many vehicles. Even when only a few vehicles participate actively in collisions, many more would carry out computer controlled emergency maneuvers. Medusa was conceived to simulate the dynamics of all these vehicles as part of an automated vehicle simulation package. As a stand-alone program, i.e., open-loop without controller input, Medusa can be used to study the isolated events that make up the complete scenario. The examples presented in this chapter represent such events. They were also chosen to demonstrate the capabilities of Medusa.

3.1 Animation with SmartPATH

The five simulated scenarios included in this chapter were animated with the SmartPATH-animator. The results are shown in the form of screen snapshots. The Medusa input file *platoon.dat* is provided for each case. This also helps illustrate the explications in the User's Manual. The Medusa input file *model.dat* is the same for all simulations and is provided in Appendix C. With the exception of the last example in this chapter, the road is assumed to be plane.

For plane roads, the Medusa input file *road.dat* is

```
NUMBER_OF_POINTS:
4
XYZ_COORDINATES_&_ANGLE:
0.       0.0   0.0   0.0
100.     0.0   0.0   0.0
500.     0.0   0.0   0.0
1000.    0.0   0.0   0.0
```

The SmartPATH-animator requires three input files in order to create an animation (see Eskafi, Khorramabadi and Varaiya [8] for details). First, there is the simulation data file which is provided by MEDUSA (it requires the filename extension **.state*). Then, there is a file **.cars* which is of the form

```
3
1 simple1
2 simple1
3 simple1
```

This file specifies the number of vehicles and their types for the animation. Here, `3` on the first line is the number of vehicles, `simple1` is the type. The vehicle type is defined in the last of the three input files which has the extension **.config*. This file also defines the highway for the animation. All the examples use the same file:

```
SECTION CARTYPE
CARTYPE  simple1
LENGTH 4.0
WIDTH 1.6
HEIGHT 1.3
FILE esprit.flt (90 0 0) (0.774 0.015 0.000 1.00)
//
SECTION HIGHWAY
LANEWIDTH 4
HIGHWAY HW1
LANE 1 MANUAL NONE_BARRIER
LANE 2 MANUAL NONE_BARRIER
GEOMETRY LINE 500 0
//
PIN HW1 -50 4 .2 0
```

3.2 An Offset Collision

The offset collision of two vehicles is probably the most common scenario expected to trigger a chain reaction on a highway. This elementary case was also used to debug MEDUSA since many errors in the vehicle model show up as visible inconsistencies in the simulation data. For example, an error in the calculation of the contact normal would obviously cause the vehicles to swerve in the wrong directions.

The offset collision is staged by letting a slower moving vehicle be rear-ended by a faster one. Both vehicles are in neutral gear, i.e., their front tires are rolling freely. We achieve this by setting `TIRE_SPEED` to zero in the file *platoon.dat*:

```
% Example: Offset Collision

NUMBER_OF_VEHICLES 2
```

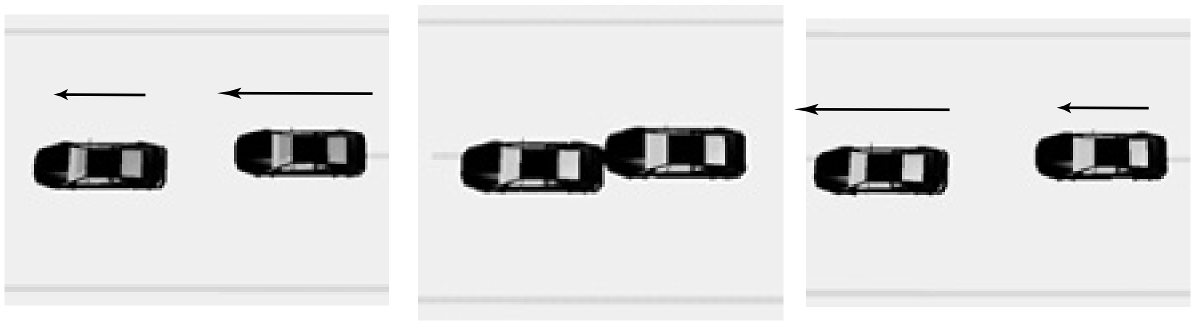

Figure 3.1: *An offset collision between two vehicles. From left to right: approach, collision and separation.*

```
% Vehicle 1
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 0.0                    % X coordinate of center of mass [m]
  Y 0.0                    % Y coordinate of center of mass [m]
  ORIENTATION 0.0          % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0              % forward speed [m/s]
  TIRE_SPEED 0.0           % in [m/s], 0.0 -> free rolling
  STEERING 0.0             % steer angle [deg]

% Vehicle 2
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 6.0                    % X coordinate of center of mass [m]
  Y 0.4                    % Y coordinate of center of mass [m]
  ORIENTATION 0.0          % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  20.0              % forward speed [m/s]
  TIRE_SPEED 0.0           % in [m/s], 0.0 -> free rolling
  STEERING 0.0             % steer angle [deg]
```

Clearly, the lateral offset is 0.4 $[m]$.

The simulation results depicted in Figure 3.1 were obtained by running MEDUSA for 10.0 seconds. The vehicles are heading towards the left. Figure 3.1_1 shows the two vehicles approaching, Figure 3.1_2 shows the collision and Figure 3.1_3 shows the vehicles separating. The arrows indicate the exchange of linear momentum during the collision.

We observe that both vehicles maintain their principal heading after the collision. This is attributable to the box-like shape of the superellipsoid which surrounds the vehicles. However, as expected, the offset in the collision causes both vehicles to drift to the left which is noticeable after some time, see Figure 3.2. As expected, the larger the offset in the collision, the stronger the drift. If the vehicles contact at their corners, the contact forces may even send the vehicles into a spin. However, in the latter case, the outcome of the

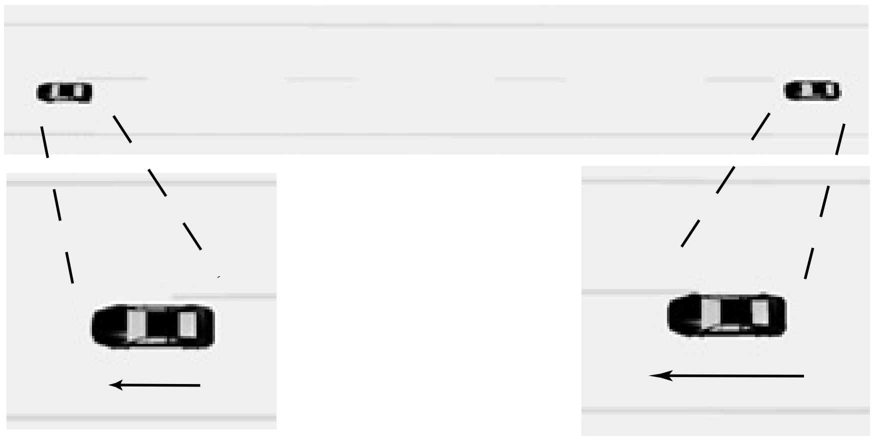

Figure 3.2: *After this offset collision, the vehicles drift towards the left side of the road.*

simulation is highly sensitive to the geometry of the vehicles' outer surfaces due to the rapid change in the surface normals at the corners. This is a well-understood issue for any vehicle accident simulation, see Fonda [9] and McHenry and McHenry [19].

3.3 A Wave Problem

One possibility to trigger a chain reaction is to cause a collision of the first vehicle in a platoon with a slower moving vehicle, see Figure 3.3. This scenario was staged by running Medusa for 20.0 seconds with the following *platoon.dat* file:

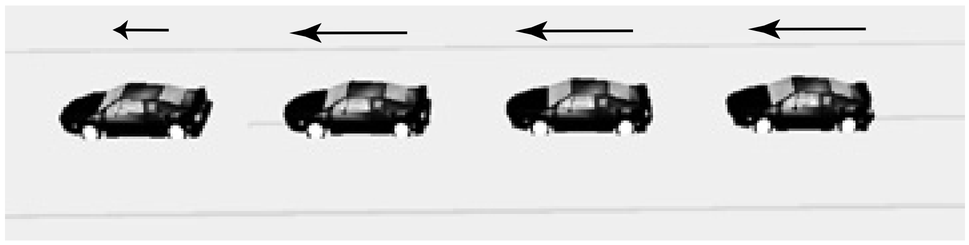

Figure 3.3: *A staged chain reaction.*

```
% Example: Chain Reaction

NUMBER_OF_VEHICLES 4

% Vehicle 1
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 0.0                  % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
```

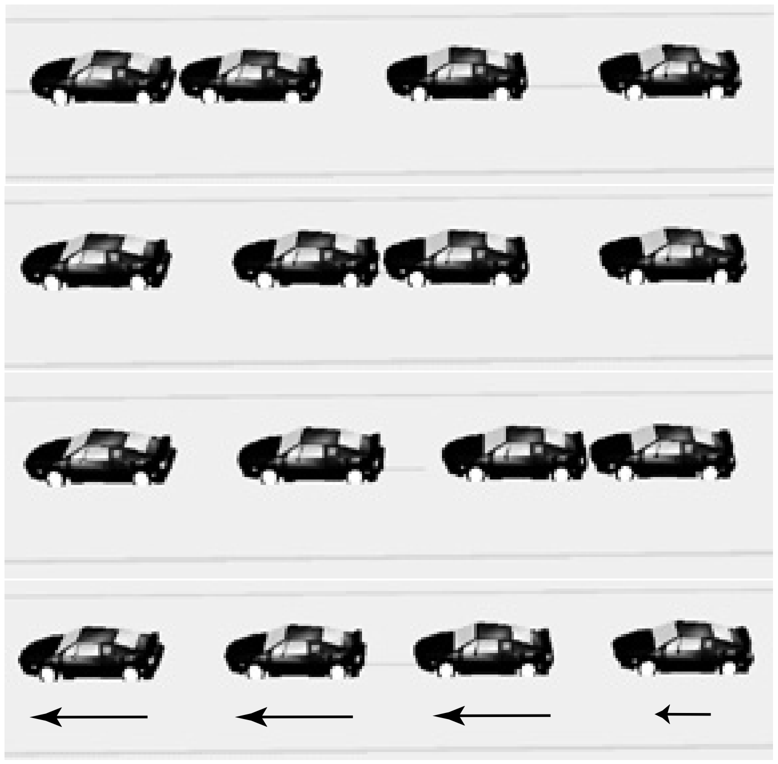

Figure 3.4: *The consecutive collisions in a chain reaction.*

```
  ORIENTATION 0.0          % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0              % forward speed [m/s]
  TIRE_SPEED 0.0           % in [m/s], 0.0 -> free rolling
  STEERING 0.0             % steer angle [deg]

% Vehicle 2
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 6.0                   % X coordinate of center of mass [m]
  Y 0.0                    % Y coordinate of center of mass [m]
  ORIENTATION 0.0          % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0              % forward speed [m/s]
  TIRE_SPEED 0.0           % in [m/s], 0.0 -> free rolling
  STEERING 0.0             % steer angle [deg]

% Vehicle 3
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 12.0                   % X coordinate of center of mass [m]
  Y 0.0                    % Y coordinate of center of mass [m]
  ORIENTATION 0.0          % heading angle [deg] (cw:"-", ccw:"+")
```

```
    SPEED  22.0              % forward speed [m/s]
    TIRE_SPEED 0.0           % in [m/s], 0.0 -> free rolling
    STEERING 0.0             % steer angle [deg]

  % Vehicle 4
  VEHICLE_HAS_MODEL 1 INITIALLY_WITH
    X 18.0                   % X coordinate of center of mass [m]
    Y 0.0                    % Y coordinate of center of mass [m]
    ORIENTATION 0.0          % heading angle [deg] (cw:"-", ccw:"+")
    SPEED  20.0              % forward speed [m/s]
    TIRE_SPEED 0.0           % in [m/s], 0.0 -> free rolling
    STEERING 0.0             % steer angle [deg]
```

If all vehicles are aligned, a sequence of collisions will ripple through the entire platoon, see Figure 3.4. In each collision, translational kinetic energy is partially converted into internal energy of the Cosserat points. Inspecting a vehicle's translational speed alone, this conversion appears therefore as dissipation. This, in return, affects the propagation of the chain reaction through the entire platoon.

Finally, the rear-end collisions of this scenario also serve to illustrate the three dimensionality of the contact algorithm. Since a vehicle's center of mass is not centered between its axles, it naturally pitches forward. When hit straight from behind by a similar vehicle, its rear is lifted up causing it to pitch forward even more. This is demonstrated in Figure 3.5 where a closeup of the initial collision in Figure 3.4_1 is shown.

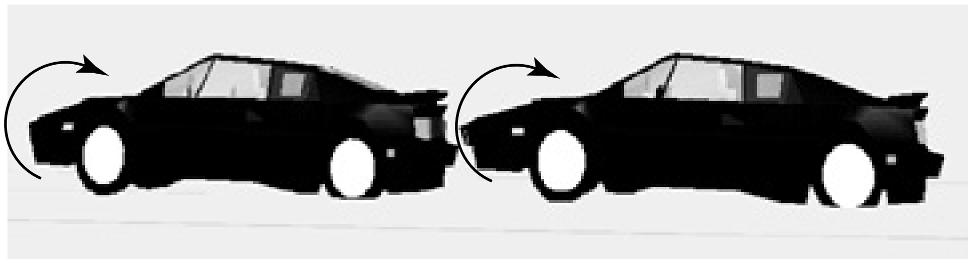

Figure 3.5: *The pitching induced by a aligned rear-end collision.*

3.4 Merging of Vehicles

One plausible collision scenario involves a rogue vehicle trying to merge into a regular platoon, see Figure 3.6. We staged such a scenario in MEDUSA with the assumption that the steering angle of the rogue vehicle is constant (steering failure and lockup). Every vehicle is assumed to be in neutral gear so that they neither brake nor accelerate. The *platoon.dat* file for this example is

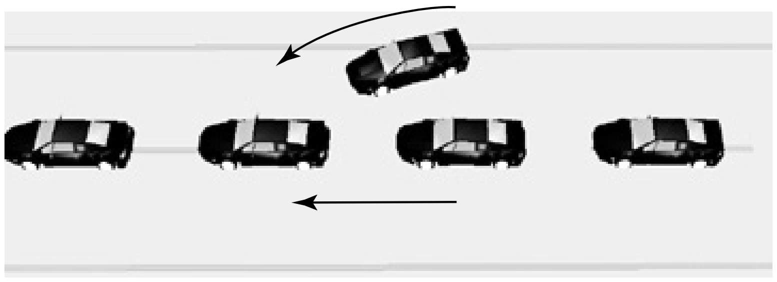
Figure 3.6: *A rogue vehicle attempting to merge into a platoon.*

```
% Example: Merging into a Platoon

NUMBER_OF_VEHICLES 5

% Vehicle 1
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 0.0                   % X coordinate of center of mass [m]
  Y 0.0                   % Y coordinate of center of mass [m]
  ORIENTATION 0.0         % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0             % forward speed [m/s]
  TIRE_SPEED 0.0          % in [m/s], 0.0 -> free rolling
  STEERING 0.0            % steer angle [deg]

% Vehicle 2
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 6.0                   % X coordinate of center of mass [m]
  Y 0.0                   % Y coordinate of center of mass [m]
  ORIENTATION 0.0         % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0             % forward speed [m/s]
  TIRE_SPEED 0.0          % in [m/s], 0.0 -> free rolling
  STEERING 0.0            % steer angle [deg]

% Vehicle 3
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 12.0                  % X coordinate of center of mass [m]
  Y 0.0                   % Y coordinate of center of mass [m]
  ORIENTATION 0.0         % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0             % forward speed [m/s]
  TIRE_SPEED 0.0          % in [m/s], 0.0 -> free rolling
  STEERING 0.0            % steer angle [deg]

% Vehicle 4
```

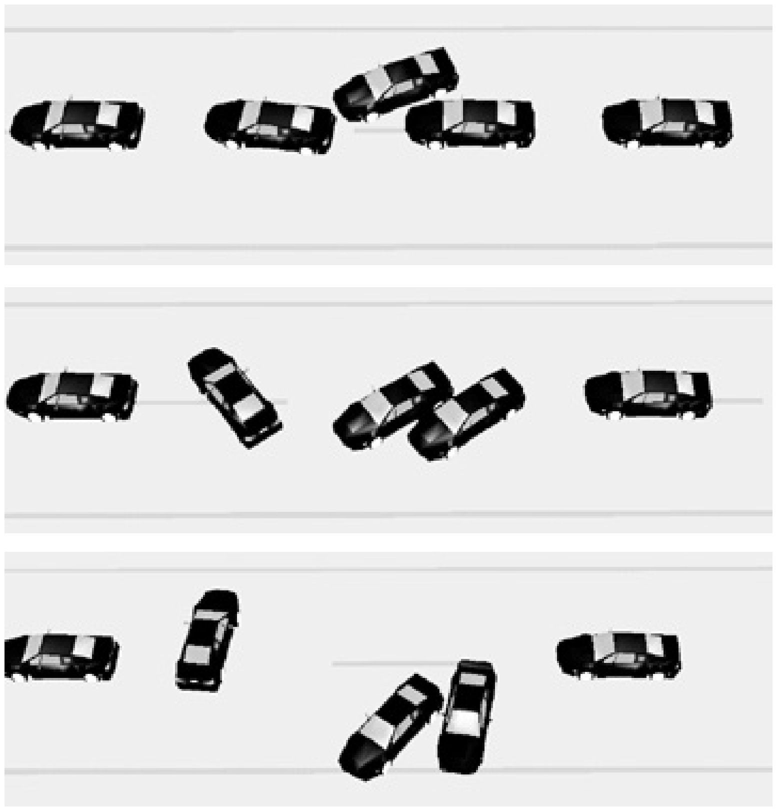

Figure 3.7: *Evolution of the merging accident scenario.*

```
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 18.0                  % X coordinate of center of mass [m]
  Y 0.0                   % Y coordinate of center of mass [m]
  ORIENTATION 0.0         % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0             % forward speed [m/s]
  TIRE_SPEED 0.0          % in [m/s], 0.0 -> free rolling
  STEERING 0.0            % steer angle [deg]

% Vehicle 5
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 6.0                   % X coordinate of center of mass [m]
  Y -4.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0         % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  25.0             % forward speed [m/s]
  TIRE_SPEED 0.0          % in [m/s], 0.0 -> free rolling
  STEERING 8.0            % steer angle [deg]
```

This example demonstrates the program's ability to simulate a large number of vehicles with multiple collisions. Indeed, the rogue vehicle first collides with the second vehicle

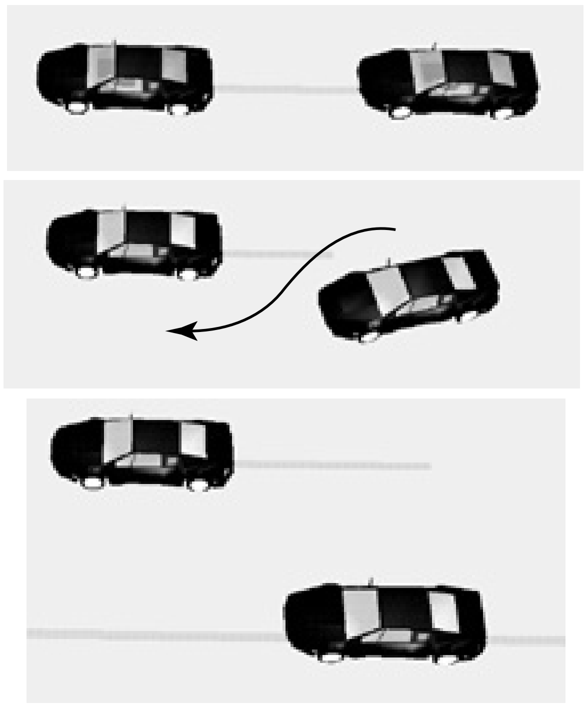

Figure 3.8: *In this scenario, a fast moving vehicle avoids a collision by changing lanes.*

in the platoon and then immediately with the third, see Figure 3.7_1. The capability of simulating simultaneous contacts with several vehicles sets MEDUSA apart from standard vehicle collision codes such as CRASH and SMAC (cf. [18, 19]).

After the initial collisions, the rogue vehicle side impacts the third vehicle in the platoon several times more while it pushes it aside, see Figures $3.7_{2,3}$. We note that during contact, the tire forces are negligible in comparison to the contact forces. However, a side impact may send a vehicle into a spin with subsequently high tire slips and slip angles. Therefore, this example illustrates the importance of an accurate tire model to cover a wide range of maneuvers (see Section 1.2.2).

3.5 An Evading Maneuver

As mentioned earlier, multiple vehicle accident scenarios involve vehicles that attempt to avoid collisions with appropriate driving maneuvers. The study of these maneuvers is the core task to computer controlled accident prevention and does not depend on collision dynamics. Here, proper tire models are extremely important.

In the example of this section, the stage is set for a potential rear-end collision, see Figure 3.8_1. The vehicle in the front is moving slower than the approaching vehicle. Reasons for this can be manifold: the slower vehicle may have motor problems or a defective clutch, or the approaching vehicle may be unable to brake. The *platoon.dat* file for this case is

```
% Example: Avoiding a Collision
% Vehicle 1
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 0.0                  % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  22.0            % forward speed [m/s]
  STEERING 0.0           % steer angle [deg]

% Vehicle 2
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 8.0                  % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [deg] (cw:"-", ccw:"+")
  SPEED  20.0            % forward speed [m/s]
  STEERING 0.0           % steer angle [deg]
```

The difficulty of running such an example in the present version of MEDUSA lies in the fact that only constant steering angles can be input using *platoon.dat*. No time history or steering control algorithms have been incorporated into MEDUSA. The steering angles needed for this scenario therefore need to be hardwired into the code. For our example, we inserted a short piece of code in the file *vehicle.c* just before the line

```
equations_of_motion(dz[cv],vehicle+cv);
```

and then recompiled the program. This code prescribes a steering angle time history for the first (faster) vehicle:

```
simulation.vehicle[1].suspension.steer_angle=
     (t<0.30) ?  0.0 :
    ((t<0.50) ?  3.0 /180*Pi :
    ((t<0.70) ?  9.0 /180*Pi :
    ((t<0.85) ?  3.0 /180*Pi :
    ((t<1.10) ?  0.0 :
    ((t<1.25) ? -3.0 /180*Pi :
    ((t<1.45) ? -7.0 /180*Pi :
    ((t<1.60) ? -3.0 /180*Pi : 0.0 )))))));
```

The simulation results are depicted in Figure 3.8. This example illustrates the difficulty of steering a vehicle in an open-loop manner. Finding the right steering inputs for this scenario involved many trial and error repetitions of the simulation. This example also demonstrates the need for an interactive driver's interface for MEDUSA so that more complex driving maneuvers may be appropriately and realistically established.

3.6 A Car on a Banked Road

The road surface of a highway is not necessary horizontal. The highway may run up- or downhill or it may be banked. For this reason, MEDUSA has a built-in road-geometry simulation capability. This final example illustrates the influence of a banked road on the vehicle dynamics. The *road.dat* file for this example is

```
NUMBER_OF_POINTS:
4
XYZ_COORDINATES_&_ANGLE:
0       0.0   0.0   5.0
30      0.0   0.0   2.5
60      0.0   0.0  -4.0
90      0.0   0.0   0.0
END_OF_FILE
```

i.e., the center line of the road remains horizontal and straight, but the road plane tilts first to the right, then to the left and finally back to horizontal.

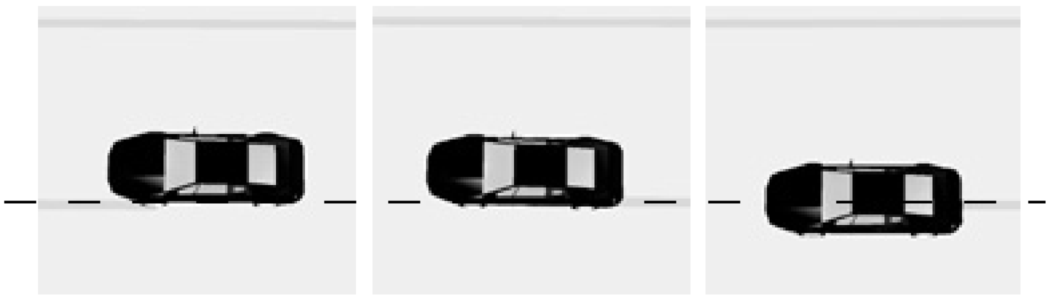

Figure 3.9: *The influence of a banked road on the vehicle dynamics. From right to left: the vehicle at various instances. The dashed line is a lateral reference.*

We simulated a single vehicle driving straight. Unlike the previous examples in this section, the front wheels are driven, i.e., they maintain the vehicle's speed (in this case $12.0\,[m/s]$). The *platoon.dat* file for this simulation is

```
% Example: Driving on a Banked Road

NUMBER_OF_VEHICLES 1

% Vehicle 1
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 0.0                  % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [deg] (cw:"-", ccw:"+")
```

```
SPEED  12.0             % forward speed [m/s]
TIRE_SPEED 12.0         % in [m/s], if 0.0, tires roll freely
STEERING 0.0            % steer angle [deg]
```

Figure 3.9 demonstrates the influence of the banked road on the path of the vehicle. Clearly, when the road tilts to the right, the vehicle moves to the right. When the road tilts left, the vehicle's sideways motion is slowed down. When the road becomes horizontal again, the vehicle pursues undisturbedly a straight path. The road tilt offsets the vehicle from the centerline of the road by about $0.5\,[m]$. We note that a driver or the automated highway controller would counteract the road influence through small steering inputs.

Chapter 4

Programmer's Guide

4.1 Introduction

Medusa is a program for the simulation of platoons of vehicles that incorporates moderate vehicle collisions. We reviewed the theory behind the various models in Medusa in Chapter 1 and explained the use of Medusa in Chapter 2. In this chapter, we describe the program code itself[1]. We presume that the reader is familiar the previous chapters and the *ANSI-C* programming language. The program consists of several functional blocks which we refer to as modules. These modules are

- initialization,
- time integration and data output,
- vehicle model,
- road model,
- contact detection.

The modules are contained in separate files and are, as far as possible, independent of each other. The complete source code is listed in Appendix F.

This guide is structured as follows. After some general guidelines and useful definitions and functions in Sections 4.2 and 4.3, we explain in Section 4.4 the function `main()` where the execution of every *C* program begins. From there, the initialization module (Section 4.5) is called which serves as the input interface to the user. Next, the time integration module is executed (Section 4.6). It integrates the equations of motion of the vehicles (Section 4.7). Details of the road module are discussed in Section 4.8. The module that detects collisions is explained in Section 4.9. During integration, data is written to output files (Section 4.10). For further details of the program, we refer to the source code in Appendix F. Section 4.11 is intended to give additional information for the programmer who wishes to make substantial changes and additions to the code.

[1]The notation that is used in this chapter is explained in Chapter 0.

4.2 Generalities

In this section we list some general comments to the code. We have attempted to make the code sufficiently clear and general so that later modifications and improvements can be easily performed. Keeping the code modular is one part of this attempt, encapsulating vehicle data in data structures is another.

MEDUSA is programmed in *ANSI-C* (see Kernighan and Ritchie [15]) which is platform independent. Macros (beginning with the `#define` keyword) are mostly written in capital letters to distinguish them visually from variables and functions.

Modularity and Global Variables

Each file of the program is considered to be an independent module of the code. Global variables are sometimes used within a file. The only global variable that is shared between all the files (using the `extern` keyword in *C*) is the variable `simulation`. It is a data structure which holds all the data essential to the simulation. If data is passed between modules, it is through this data structure or through function arguments, never through other global variables.

Data Types for Numerics

Floating point variables are all at least double precision. For intermediate calculations (such as matrix multiplications), the data type `long double` is used. Compilers that do not support this data type generate warning messages which may be ignored.

For the storage of vectors and matrices and for use in vector and matrix operations, we use the data types `Vector` and `Matrix` from the *Numerical Recipes in C* [26]. We explain these data types in detail in Section 4.3.1.

Structures and Pointers

Structures and especially pointers to structures are an efficient way to pass large amounts of different kinds of data between functions. In MEDUSA, almost all of the data used to run the simulation is stored in structures which are defined in the file *common.h*. We discuss the contents of these structures in detail in Section 4.3.2, and list them in Appendix D for reference. Here, we give a brief description of these important structures:

`vehicle_struct` This structure is used to store a vehicle's model parameters and all its state variables. It contains substructures, scalar variables, vectors and matrices. For each vehicle, it is initialized at the beginning of the simulation (cf. Section 4.5).

`simu_struct` This structure exists only once in the program and all the simulation parameters and simulation data. For example, it contains not only the integration step-size and the duration of the simulation, but also the array of `vehicle_struct` data structures.

Source Code Listings

The complete MEDUSA code is listed in Appendix F. The lines and pages are numbered. Each listing has a header containing the filename, the date of modification and a short description of the file. The global variables and function names are listed with line numbers at the top of each listing to help localize them in the code. For additional reference, Appendix E lists the function dependencies, but note that for brevity, we do not list the dependencies on functions contained in *common.h*.

4.3 The Files *common.h* and *common.c*

The header file *common.h* provides a body of definitions and function prototypes which are used as tools in all other parts of the MEDUSA program code. Every source code file contains the line

```
#include "common.h"
```

to make the definitions available. The functions corresponding to the prototypes are coded in the file *common.c* (see Appendix F). Many of these functions are adapted from *Numerical Recipes in C* [26]. We refer to that book for a more detailed discussion of them.

4.3.1 Programming Tools for Vectors and Matrices

Vectors

For many vectors with known length, we declare in the source code double precision floating point arrays such as

```
double a[N+1];
```

In this example, the array elements `a[i]` correspond to the vector components a_i ($i = 1, \ldots, N$) of an N-dimensional vector **a**. The array element `a[0]` is not used and is void. Often we write

```
double a[N+1]={0.0};
```

to explicitly set the unused `a[0]` element to zero. This may be helpful in debugging the program code.

In the *ANSI-C* programming language, pointers and one-dimensional arrays are essentially the same (see Kernighan and Ritchie [15]). For instance, in the example above, `a[1]` and `*(a+1)` denote the same array element. For this reason, we have created a new data type `Vector` in the file *common.h*:

```
typedef double *Vector;
```

This data type increases the readability of the source code and helps to identify physical vectors in the program.

The memory for arrays can also be allocated dynamically during execution time using the standard *C*-function `malloc()`. *Numerical Recipes in C* [26] offers a more convenient solution to allocate memory for vectors. We have adopted their solution and define the following function prototypes in the file *common.h*:

```
Vector vector(int n);
void free_vector(Vector);
```

We explain their use with the following short sample program:

```
#include <stdio.h>
#include "common.h"
void fun(int n)
{   /* calculate and print the squares of the first n integers: */
    Vector a=vector(n);
    int i;
    for (i=1;i<=n;i++) a[i]=i*i;
    for (i=1;i<=n;i++) printf("%d, ",a[i]);
    free_vector(a);
}
```

In this example, the function `vector(n)` allocates memory for a double precision vector of length n and returns a pointer to that memory to the variable `a`. This variable can now be used just like a normal one-dimensional array. The function `free_vector(a)` de-allocates the memory once the calculations have been performed. After this, the variable `a` is void. We note that `vector()` does not allocate memory for the array element `a[0]`. Using `a[0]` in the above example would probably lead to a runtime error.

Matrices

We also adapted the ideas proposed in the *Numerical Recipes in C* [26] to treat matrices. To denote a matrix, we have created the new data type `Matrix`:[2]

```
typedef double **Matrix;
```

Similar to the functions `vector()` and `free_vector()` for vectors, the following functions are used to allocate and free memory for a n by m matrix, respectively:

```
Matrix matrix(int n, int m);
void free_matrix(Matrix);
```

[2]Note that the declarations `Matrix`, `*Vector` and `**double` are equivalent.

The `Matrix` data type can be used just like a two-dimensional array. For example, the following (admittedly simple) code segment creates a three by three matrix A, assigns the value 2.0 to the matrix element A_{12} and eliminates the matrix again:

```
Matrix A=matrix(3,3);
A[1][2]=2.0;
free_matrix(A);
```

There is however a fundamental difference between two-dimensional arrays and the `Matrix` data type. In the example above, `A` is actually a pointer to a one-dimensional array of row vectors of data type `Vector`. The following code segment illustrates this very useful property which is sometimes used in Medusa:

```
Matrix A=matrix(3,3);
Vector v;
int i,j;
for (i=1;i<=3;i++) for (j=1;i<=3;j++) A[i][j]=3*(i-1)+j;
v=A[2];
printf("The second row of A is %d %d %d",v[1],v[2],v[3]);
free_matrix(A);
```

Here, the variable `v` points to the second row of the matrix `A`. The output from this code segment is therefore:

```
The second row of A is 4 5 6
```

Vector and Matrix Operations

There are a number of functions defined in *common.h* that handle vector and matrix operations. When using these functions, one should remember to allocate memory for the variables using the functions `vector()` and `matrix()` discussed above. Note also that, as with any dynamically allocated memory, one must be careful to free allocated memory when it is not used anymore (with `free_vector()` and `free_matrix()`).

4.3.2 Structures

The Data Type `vehicle_struct`

The `vehicle_struct` data structure helps to conveniently manage the parameters and variables pertaining to a vehicle. It is listed in Appendix D.1 for reference. Medusa makes extensive use of pointers to structures of this type. The following is a complete list of the contents of `vehicle_struct`:

`int ident`: This is a unique number identifying the vehicle.

`Vector z`: This is the vehicle's state vector from equation (1.32) that is being integrated.

`Cosserat_point`: This is a substructure containing the parameters of the Cosserat point that is used to model the chassis of the vehicle:

`double m`: The mass of the vehicle chassis.

`Matrix I`: The inertia matrix $\mathbf{M}$ in equation (1.31).

`Matrix I_inv`: The inverse of the matrix `model->Cosserat_point.I`.

`double lam, tm`: The constants $\frac{V}{2}\lambda$ and $V\mu$ of equation (1.12).

`suspension`: This substructure contains the parameters of the suspension model:

`double steer_angle`: The steering angle of the front wheels. MEDUSA reads it from `STEERING` keyword in the platoon description file (cf. Section 2.2).

`double X1[5], X2[5], X3[5]`: Coordinates of assembly points as defined in equation (1.11) and Figure 1.1. X1[1] corresponds to L_1, X2[2] to $-B/2$, X3[3] to $-H_2$, and so forth.

`double spring_ref`: The unstretched length Δs of the springs defined in equation (1.17).

`double C[5], D[5]`: The spring and damping constants of the suspension as defined in equation (1.17). For example, `C[2]` is the spring constant for suspension 2.

`Matrix infl`: The influence matrix $\mathbf{A}$ in equation (1.31).

`tire`: This substructure contains parameters related to the tire model.

`double tire.tau_inv`: This is the tire force lag parameter τ^{-1} in equation (1.27).

`double speed`: Circumferential speed of the front tires ($R\omega$ in equation (1.19)).

`int driven`: Can be `TRUE` or `FALSE`. If `FALSE`, the variable `speed` is ignored and the front tires are assumed to be rolling freely.

`road`: This substructure contains information about where the vehicle is with respect to the road:

`int segment`: The road segment number (see Section 1.3).

`double parameter`: The parameter s of the segment (see equation 1.35).

`contact`: This substructure contains information pertaining to the contact between vehicles:

`double dimension_box[4` :] Physical dimensions of the vehicle. MEDUSA reads them from `CONTACT` keyword in the model description file (see Appendix C).

`Vector force`: The resultant constraint force of all other vehicles acting on this vehicle (i.e. the contributing forces are those from equation (1.42)).

`Matrix previous` This matrix records at which surface points the vehicle was in contact with other vehicles in previous calculations. A surface point corresponds to the vector $\mathbf{x}$ in equation (1.47).

`Matrix multiplier`: This matrix is in fact a pointer to an array that contains the Lagrange multipliers γ_1 and γ_2 of the contact force equations (1.42) for each pair of vehicles. All the vehicles share this matrix.

`init`: This substructure defines a state of the vehicle relative to which the initial conditions are determined:

`double r3`: The reference value of the vertical component of the vehicle's position vector. This value is specified by the keyword `R3` in the `EQUILIBRIUM` sections of the file *model.dat* (see Section 2.2).

`Matrix F`: This matrix corresponds to the deformation gradient $\mathbf{F}$ defined in equation (1.3). It is specified in the `EQUILIBRIUM` sections of the file *model.dat*.

The Data Type `simu_struct`

The `simu_struct` data structure helps to conveniently manage all the simulation parameters and the vehicle data. It is listed in Appendix D.2 for reference. The following is a list of its contents:

`char *in_file`: This string stores the name of the platoon description file. By default it is `DEFAULT_INPUT_FILE`. The name is changed during run-time by the option `-f` (see Section 2.3).

`FILE *ofp`: This scratch file is initialized by `init()`. It exists mainly for the sake of debugging the program.

`int NofV`: The number of vehicles in the simulation. MEDUSA finds this number in the platoon description file after the keyword `NUMBER_OF_VEHICLES` (see also Section 2.2).

`int NofM`: The number of vehicle models which MEDUSA finds in the file *model.dat* after the keyword `NUMBER_OF_MODELS` (see also Section 2.2).

`vehicle_struct *model`: This array stores the vehicle models that have been read from the file *model.dat*.

`vehicle_struct *vehicle`: This is the array of vehicles that has been previously described.

`integrate`: This substructure contains parameters controlling the integration:

`double end_time`: The simulation ends after a simulated time of `end_time` seconds. This corresponds to the run-time option `-t` (see Section 2.3).

`double delta_t`: The fixed step-size of the integrator. The default value is `DELTA_T`. The value is changed by the option `-d` in Section 2.3.

`double save_delta_t`: In time intervals of length `save_delta_t`, data points are written to the output files. The default is `SAVE_DELTA_T`. The value is changed by the run-time option `-s` (see Section 2.3).

`double t`: Current time.

`road`: This substructure contains the road geometry parameters:

`int nodal_points`: Number of points that define the road.

`Matrix XIN`: The Cartesian coordinates of each road points.

`Matrix *C`: The coefficients of each road spline (see equation 1.35).

4.4 The Function `main()`

ANSI-C starts the execution of the MEDUSA program code with the function `main()` which calls the function `init()` (see Section 4.5), opens the output files, prints information pertaining to the simulation to the screen, and calls the function `integrate()` (cf. Section 4.6) which performs the actual simulation of the vehicles. After some cleanup, the program then ends.

4.5 The Initialization Module

The file *init.c* constitutes the initialization module whose main function is `init()`. In sequence, the command line options are evaluated by the function `evaluate_cmd_line()` (cf. Section 2.3), the user input files are read by `read_models()` and `read_vehicles()` (cf. Section 2.2), and the road geometry is initialized by `road_init()`. We explain these functions individually below.

Evaluation of the Command Line

In *ANSI-C*, the contents of the command line are stored in the variables `argv` and `argc` (cf. Kernighan and Ritchie [15]). The function `evaluate_cmd_line()` reads the command line parameters (such as the simulation time) from these variables and stores them in the global variable `simulation` according to the following table (cf. Section 2.3):

`-f`	`simulation.in_file`
`-d`	`simulation.integrate.delta_t`
`-s`	`simulation.integrate.save_delta_t`
`-t`	`simulation.end_time`

Reading Files

To perform the task of reading one of the input files from the disk into a buffer string, the function `read_file()` is provided. This function is blind to comments and multiple white-space characters[3]. To search a buffer string for a keyword, or token, the functions `read_expr()` and `find_token()` are provided. These two functions generate error messages if the guidelines for writing input files are violated. These guidelines are described in Section 2.2.

Evaluation of the File *model.dat*

The function `read_models()` uses `read_file()` to obtain a buffer string which contains the condensed contents of the file *model.dat*. The keyword `NUMBER_OF_MODELS` indicates the number of vehicle models (stored in `simulation.NofM`). A `vehicle_struct` array is then generated in the variable `simulation.model`.

According to the guidelines in Section 2.2, for each model `m`, the buffer string is searched for the proper sequence of keywords and the corresponding parameters. These parameters are either stored directly in the structure `simulation.model[m]` (cf. Section 4.3.2), or else they are used to calculate further quantities. In particular:

`IX, IY, IZ`: Using equations (1.8), (1.9) and (1.31), the following matrices are calculated:

```
simulation.model[m].Cosserat_point.I
simulation.model[m].Cosserat_point.I_inv
```

`E, NU, VOLUME`: Using equations (1.13), the following variables are calculated:

```
simulation.model[m].Cosserat_point.lam
simulation.model[m].Cosserat_point.tm
```

`L1, L2, B, H1, H2, SPRING_REF, C1, C2, D1, D2`: These parameters are stored in the substructure `simulation.model[m].suspension`. Using equations (1.11) and (1.31), the influence matrix `model[m].suspension.infl` is calculated.

`TIRE`: The inverse of this value is stored in `simulation.model[m].tire.tau_inv`.

`D11, ..., D33`: Components of the matrix `simulation.model[m].init.F` (cf. equation (1.3)).

[3]For a definition of white-space characters, see Kernighan and Ritchie [15].

Reading the Platoon Data File

The function `read_vehicles()` uses `read_file()` to obtain a buffer string which contains the condensed contents of the file `simulation.in_file`. In this buffer, the keyword `NUMBER_OF_VEHICLES` indicates the number `simulation.NofV` of vehicles in the platoon. An array of `vehicle_struct` structures is then generated in the variable `simulation.vehicle`.

For each vehicle v, the buffer string is searched for the keyword `VEHICLE_HAS_MODEL`, say `m`. The structure `simulation.model[m]` is then copied to `simulation.vehicle[v]`. Next, the state vector `simulation.vehicle[v].z`, `z` for short, is initialized with the initial conditions of the simulation (cf. equation (1.32)). In particular, the parameters `X` and `Y` are stored in `z[1]` and `z[2]`, while the variable `z[3]` is copied from `simulation.model[m].init.r3`. The variables `z[4]` through `z[12]` correspond to the $\mathbf{E}_i \otimes \mathbf{E}_j$-components of the initial deformation gradient $\mathbf{F}_0$ from equation (1.3). It is calculated from the reference deformation gradient $\mathbf{F}_{ref}$ (stored in the matrix `simulation.model[m].init.F`) as follows:

$$\mathbf{F}_0 = \mathbf{Q}(\theta)\,\mathbf{F}_{eq} \quad ,$$

where the rotation tensor $\mathbf{Q}(\theta)$ corresponds to a counter-clockwise rotation through an angle θ about the $\mathbf{E}_3$-axis, and θ is the `ORIENTATION` parameter.

The variables `z[13]` and `z[14]` are the initial speeds of the vehicle in the $\mathbf{E}_1$ and $\mathbf{E}_2$ direction, respectively. They are calculated from the forward speed `SPEED` and θ. The remaining states `z[15]` through `z[24]` are set to zero.

4.6 Time Integration

The file *main.c* contains the function `integrate()` which calculates the left-hand sides of equations $(1.52)_{1,4,5,6}$ over a period of time `simulation.integrate.end_time` at the time-rate `simulation.integrate.delta_t`. The function `set_constraint_forces()` supplements hereby the equations $(1.52)_{2,3}$ from which the constraint forces $\mathbf{c}_1$ and $\mathbf{c}_2$ in the right-hand side of equations $(1.52)_{4,5}$ are calculated (see Section 4.7). The right-hand sides of equations $(1.52)_{4,5,6}$ are supplied by the function `equations_of_motion()` which is discussed in Section 4.7.

To output intermediate integration steps to a file, `integrate()` calls the functions `save_data_point()` and `write_to_file()` which are discussed in Section 4.10.

It was pointed out in Section 1.6 that the step size Δt (i.e., the variable `dt`) of the integrator is reduced just before two vehicles come into contact. For this reason, we use the variable `h` to hold a copy of the state vectors of all vehicles. The function `set_constraint_forces()` returns a non-zero value when contact is detected (see Section 4.7). In that case, the integrator restores the state vectors of the vehicles from `h` and tries another integration step with a smaller Δt. The integrator continues to integrate with the current Δt if no vehicles are in contact or if Δt has already been reduced to a value which is smaller then the one given by the macro `DT_MIN`.

Before Δt can be increased again, the time integration continues with this step size for at least 100 steps. This is ensured by the counter variable `just_reduced` and reduces oscillations in the variable `dt`. After this period, Δt is increased again when no vehicles are in contact.

4.7 The Vehicle Model

The file *vehicle.c* represents the actual vehicle model. It contains three separate principal functions: `set_constraint_forces()` calculates the contact forces (1.42) that act between the vehicles, `equations_of_motion()` calculates the equations of motion (1.41) for a single vehicle and `energy()` calculates the total energy (1.33) of a single vehicle.

`set_constraint_forces()`: For each pair of vehicles with numbers `cv` and `av`, this function calls the function `detect_contact()` (see Section 4.9) to determine if they are in contact. If no contact is detected, the Lagrange multipliers (which are stored in the two-vector `vehicle[cv].contact.multiplier[av]`) of the two vehicles are set to zero.

If contact is detected, the function will return a non-zero value to indicate this occurrence. The constraint function ϕ_1 from equation $(1.39)_1$ is calculated[4]. The coordinates $X^i_{\sigma(\beta)}$ in equation $(1.39)_2$ are calculated using equations (1.2) and (1.3). They are also used to calculate ϕ_2 from equation (1.40). The Lagrange multipliers are now updated using equations $(1.52)_{2,3}$. Finally, using equation (1.42), the contact force is added to the contact force resultants `vehicle[cv].contact.force` and `vehicle[av].contact.force`, respectively.

`equations_of_motion()`: This function contains the vehicle model outlined in Section 1.1. Its input argument is a pointer to the current vehicle's data structure. The output argument is the time derivative `Vector dzdt` of the vehicle's state vector.

To simplify matters and for brevity, the vehicle's state vector is split up into the global variables `r`, `d1`, `d2`, `d3`, `v`, `w1`, `w2`, `w3` of type `Vector` which represent the position vector $\mathbf{r}$, the directors $\mathbf{d}_i$, the velocity $\mathbf{v}$ and the director velocities $\mathbf{w}_i$ of the Cosserat point which is defined in equation (1.2).

For each wheel of the vehicle, the wheel heading is calculated using equations (1.14) and (1.15). Next, the force vector $\mathbf{f}$ from equation (1.30) (see equations (1.16) and $(1.29)_5$) is calculated. It is composed of the suspension force (cf. equation (1.17)) and the tire force (cf. equations (1.20)–(1.23)) and remarks in Section 1.2.2. The tire parameters are currently those for a Goodyear 185SR14 tire, cf. equations (1.24)-(1.26).

The time derivative `Vector dzdt` of the state vector `Vector z` is now calculated using equations (1.32), (1.30) and (1.41). The intrinsic forces are calculated according to equations (1.30), (1.12) and $(1.29)_4$.

[4]The variables $\mathbf{r}^*_{(1)}$ and $\mathbf{r}^*_{(2)}$ of equation (1.39) correspond to the variables `rho1` and `rho2` in the code.

`energy()`: This function calculates the total energy E of a single vehicle using equations (1.33) and (1.34). Its input argument is a pointer to the data structure of that vehicle.

4.8 The Road Model: *road.c*

`road()`: This function called by *vehicle.c* calculates the road tangent and normal vectors at the corresponding points defined in Section 1.3. The input argument is a vehicle structure.

Two functions are programmed in this module to calculate the corresponding point on the road curve for a vehicle.

- `dist_road()`: provides the value of the distance between the mass center of a vehicle and the corresponding point on the road.
- `d_road()`: outputs the gradient of the distance function in which the coordinates are defined by the parameter of the road curve.

4.9 The Determination of Contact: *contact.c*

The module contained in the file *contact.c* does three major tasks: for a pair of vehicles, it determines contact, searches for the points of minimum-distance and determines unique contact points as discussed in Section 1.5. This module returns the contact information needed by the function `set_constraint_forces()`(see Section 4.7).

`detect_contact()` This function is called by `set_constraint_forces()`. It needs the following input arguments:

- `Vector cm1, cm2` : the position vectors of the respective centers of mass of vehicles 1 and 2 as defined in equation (1.2)[5].
- `Matrix deform1, deform2` : the deformation gradients of the two vehicles. These two matrices are reassigned to the matrices F1 and F2 which corresponds to the notation in equation (1.4).
- `Vector state` : This is the vector $\mathbf{x}$ defined in equation (1.47). It contains the $u-v$ coordinates of the most recently found contact points on each of the two vehicles.
- `vehicle_struct *car1, *car2` : These structures contain the model parameters of the two vehicles.

and the outputs

- `Vector n`: the unit outward normal at the contact point of vehicle 1.

[5]Recall that the position vector of the Cosserat point coincides with the position vector of center of mass of the chassis.

- `Vector r1, r2`: the position vectors of the contact points of the respective vehicles relative to the center of mass of vehicle 1.

Several functions are programed in this module. These are

- `info()` calculates the matrix $\hat{\mathbf{K}}$ which is discussed in equation (1.50).
- `eig()` calculates the principal lengths and directions of a superellipsoid in the current configuration.

  Two additional functions, `jacobi()` and `eigsrt()`, are needed in `eig()` which are copied from *Numerical Recipes in C* [26].
- `jacobi()` calculates the eigenvalues and eigenvectors of the matrix $\hat{\mathbf{K}}$.
- `eigsrt()` sorts the eigenvalues into descending order and rearranges the eigenvectors correspondingly.
- `pos()` calculates the position vector of the contact point using equation (1.4).
- `enorm()` calculates the unit outward normals at the contact points using equation (1.46) which is discussed in Section 1.5.1.
- `norm_angle()` normalizes the state vector which is calculated in function `minimum()` to $[0,\ 2\pi]$.
- `dist()` provides the value of the distance function at the coordinates given by the vector variable `state`.
- `d_dist1()` calculates the components of the gradient of the distance function in which the coordinates of $u_{(2)}$ and $v_{(2)}$ are fixed and $u_{(1)}$ and $v_{(1)}$ are given by the vector variable `state`.
- `d_dist2()` calculates the components of the gradient of the distance function in which the coordinates of $u_{(1)}$ and $v_{(1)}$ are fixed and $u_{(2)}$ and $v_{(2)}$ are given by the vector variable `state`.
- `d_dist()` calculates the components of the gradient of the distance function in which the coordinates are given by the `vector state`.

  The functions `dist()`, `d_dist1()`, `d_dist2()` and `d_dist()` are required by the function `minimize()`.

`pert()`: This function searches for the unique contact points of two vehicles (see Section 1.5.3).

- `func()` calculates the value of the function $\hat{f}_{(\beta)}$ (cf. equation (1.51)).

- `piksrt()` sorts the input matrix `brr`, which records the curvilinear coordinates of 2n points by their corresponding values in `arr[1..n]`. The latter vector records the values of the function $\hat{f}_{(\beta)}$ (cf. equation (1.51)). This module outputs the matrix `brr`.
- `opp()` calculates the positions of the points $M_{(\beta)}$, $N_{(\beta)}$, $P_{(\beta)}$, $Q_{(\beta)}$ and $R_{(\beta)}$, $\beta = 1, 2$ (see Figure 1.10). This function converges when the function $\hat{f}_{(2)}$ at these points is close to zero.
- `search()` calculates the point of maximum penetration by first searching along the curve connecting $\bar{N}_{(1)}$ and $\bar{M}_{(1)}$ and then by searching along the curve connecting K and $\bar{T}_{(1)}$ (see Section 1.5.3).

4.10 Data Output

The MEDUSA file *main.c* contains a simple algorithm to handle the output of data following the guidelines in Section 2.4. The state vectors of all vehicles are first saved to a buffer in time intervals `simulation.integrate.save_delta_t` by the function `save_data_point()`. When the buffer is full[6] or at the end of the simulation, it is written to the various output files by the function `write_to_file()` using the standard *ANSI-C*-function `fprintf()` (cf. Kernighan and Ritchie [15]).

4.11 Adding User Supplied Code

The user may wish to make modifications and improvements to any part of the program. We have attempted to simplify this endeavor by keeping the code as modular as possible. As shown in Section 4.3.2, all the relevant data is accessible through the global variable `simulation`. This data structure subsumes, among other quantities, a `vehicle_struct` array which represents the vehicles. Besides adding new functions to the code, most modifications are done by adding additional members to these structures (defined in the file *common.h*, see also Appendix D). This has the advantage that the basic structure of the code remains unaltered. This encapsulation of data parallels somewhat the ideas of object oriented programming in *C++* and simplifies the debugging process considerably. In the future, real object oriented extensions are envisioned.

As a final comment, we wish to point out the `FILE` pointer `simulation.ofp` which represents a scrap file. It is normally left empty, but remains in the code for the purpose of debugging a modified program.

[6]The buffer length is given by `KMAX`.

Bibliography

[1] R. W. Allen, R. E. Magdaleno, T. J. Rosenthal, D .H. Klyde and J. R. Hogue. Tire modeling requirements for vehicle dynamics simulation. SAE 950312. Society of Automotive Engineers, Warrendale PA, 1995.

[2] A. H. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications.* Vol. 1, No. 1, pp. 11-12 and pp. 15-23.

[3] R. H. Bartels, J. C. Beatty and B. A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling.* M. Kaufmann Publishers, California, 1987.

[4] H. Cohen and R. G. Muncaster. *The Theory of Pseudo-rigid Bodies.* Springer Tracts in Natural Philosophy, Vol. 33, Springer-Verlag, New York, 1988.

[5] J. W. Daniel. *The Approximate Minimization of Functionals.* Prentice-Hall, New Jersey, 1971.

[6] T. D. Day. An overview of the HVE vehicle model. SAE 950308. Society of Automotive Engineers, Warrendale PA, 1995.

[7] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, New York, 1983.

[8] F. Eskafi, D. Khorramabadi and P. Varaiya. SmartPath: An Automated Highway System Simulator. PATH Research Report UCB-ITS-TM-92-3, University of California at Berkeley, 1992.

[9] A. G. Fonda. Crush energy formulations and single-event reconstruction. SAE 900099. Society of Automotive Engineers, Warrendale PA, 1990.

[10] W. R. Garrot. Measured vehicle inertia parameters - NHTSA's data through September 1992. SAE 930897. Society of Automotive Engineers, Warrendale PA, 1993.

[11] A. E. Green and P. M. Naghdi. A thermomechanical theory of a Cosserat point with application to composite materials. *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 44, pp. 335-355, 1991.

[12] J. L. Greenstadt. Variations on variable-metric methods. *Mathematics of Computation*, Vol. 24, pp. 1-22, 1970.

[13] G. J. Heydinger, N. J. Durisek, D. A. Coovert, D. A. Guenther and S. J. Novak. The design of a vehicle inertia measurement facility. SAE 950309. Society of Automotive Engineers, Warrendale PA, 1995.

[14] W. Kortüm and R. S. Sharp, editors. *Multibody Computer Codes in Vehicle System Dynamics*, in *Vehicle System Dynamics*, Vol. 22 Supplement. Swets and Zeitlinger, Amsterdam, 1993.

[15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. 2nd ed., Prentice Hall, 1988.

[16] J. Kowalik and M. R. Osborne. *Methods for Unconstrained Optimization Problems.* American Elsevier Publishing Company, New York, 1968.

[17] B. G. McHenry and R. R. McHenry. HVOSM-87. SAE 880228. Society of Automotive Engineers, Warrendale PA, 1988.

[18] B. G. McHenry and R. R. McHenry. SMAC-97 – Refinement of the collision algorithm. SAE 970947. Society of Automotive Engineers, Warrendale PA, 1997.

[19] B. G. McHenry and R. R. McHenry. CRASH-97 – Refinement of the trajectory solution procedure. SAE 970949. Society of Automotive Engineers, Warrendale PA, 1997.

[20] O. M. O'Reilly, P. Papadopoulos, G.-J. Lo and P. C. Varadi. *Models of Vehicular Collision: Development and Simulation with Emphasis on Safety. I: Development of a Model for a Single Vehicle.* California PATH Research Report UCB-ITS-PRR 97-15, 1997.

[21] O. M. O'Reilly, P. Papadopoulos, G.-J. Lo and P. C. Varadi. *Models of Vehicular Collision: Development and Simulation with Emphasis on Safety. II: On the Modeling of Collision between Vehicles in a Platoon System.* California PATH Research Report UCB-ITS-PRR 97-34, 1997.

[22] O. M. O'Reilly, P. Papadopoulos, G.-J. Lo and P. C. Varadi. *Models of Vehicular Collision: Development and Simulation with Emphasis on Safety. III: Computer Code, Programmer's Guide and User Manual for MEDUSA.* California PATH Research Report UCB-ITS-PRR 98-10, 1998.

[23] O. M. O'Reilly, P. Papadopoulos, G.-J. Lo and P. C. Varadi. *Models of Vehicular Collision: Development and Simulation with Emphasis on Safety. IV: An Improved Algorithm for Detecting Contact Between Vehicles.* California PATH Research Report UCB-ITS-PRR 98-25, 1998.

[24] O. M. O'Reilly and P. C. Varadi. A unified treatment of constraints in the theory of a Cosserat point. *Journal of Applied Mathematics and Physics (ZAMP)*, Vol. 49, pp. 205-223, 1998.

[25] H. B. Pacejka. The role of tyre dynamics properties. In: *Smart Vehicles*. Swets & Zeitlinger, Lisse, Netherlands, 1995.

[26] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery. *Numerical Recipes in C: the Art of Scientific Computing*. 2nd ed., Cambridge University Press, 1992.

[27] M. B. Rubin. On the theory of a Cosserat point and its application to the numerical solution of continuum problems. *ASME Journal of Applied Mechanics*, Vol. 52, pp. 368-372, 1985.

[28] D. J. Schuring. Tire parameter determination, Vol. V - tire test data. NHTSA DOT HS-802 090, Nov. 1976.

[29] I. S. Sokolnikoff. *Mathematical Theory of Elasticity*. 2nd ed., Mc Graw Hill, New York, 1956.

[30] C. Truesdell, R. A. Toupin. The Classical Field Theories, in *Handbuch der Physik*. Vol. III/1, pp. 226-858, edited by S. Flügge, Springer-Verlag, Berlin, 1960.

[31] P. C. Varadi, G.-J. Lo, O. M. O'Reilly and P. Papadopoulos. A novel approach to vehicle dynamics using the theory of a Cosserat point and its application to collision analyses of platooning vehicles. *Vehicle System Dynamics*, to appear 1999.

Appendix A

The Variable Metric Method

Here, we review the variable metric method which is used to determine the contact point. The lateral surface of a superellipsoid can be rather flat. As a result, it may be computationally difficult (and expensive) to determine contact points on it. Hence, there is a pay-off between a realistic vehicle geometry and computational expense.

In Section 1.5, the distance function from an arbitrary point outside the superellipsoid to any point located on the surface of the superellipsoid is given and needs to be minimized to locate the contact points. In other words, the problem that needs to be solved is an unconstrained minimization problem [5, 7] which may be expressed as follows:

$$\min_{\mathbf{x} \in \mathbb{R}^n} f : \mathbb{R}^n \longrightarrow \mathbb{R} . \tag{A.1}$$

The distance function $f(\mathbf{x})$ can be approximated as a quadratic form using a Taylor's series expansion about $\mathbf{x}_j$:

$$f(\mathbf{x}) \doteq f(\mathbf{x}_j) + \nabla f(\mathbf{x}_j) \cdot (\mathbf{x} - \mathbf{x}_j) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_j) \cdot \mathbf{H}(\mathbf{x}_j)\,(\mathbf{x} - \mathbf{x}_j) , \tag{A.2}$$

where

$$\nabla f(\mathbf{x}_j) = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_j} , \quad \mathbf{H}(\mathbf{x}_j) = \left.\frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_j} , \tag{A.3}$$

are the gradient vector and Hessian of the function $f(\mathbf{x})$ evaluated at $\mathbf{x} = \mathbf{x}_j$, respectively. Thus, by differentiating (A.2),

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{x}_j) + \mathbf{H}\,(\mathbf{x} - \mathbf{x}_j) . \tag{A.4}$$

In Newton's method, the next iteration point $\mathbf{x}$ is computed by setting $\nabla f(\mathbf{x}) = \mathbf{0}$ and is given by

$$\mathbf{x} - \mathbf{x}_j = -\mathbf{H}^{-1}\,\nabla f(\mathbf{x}_j) . \tag{A.5}$$

A scalar function f decreases at a point $\mathbf{x}$ in the direction of $\mathbf{x}-\mathbf{x}_j$, i.e., the Newton direction in (A.5) is a descent direction if the directional derivative along this direction of the function $f(x)$ is negative

$$\nabla f(\mathbf{x}_j)\cdot(\mathbf{x} - \mathbf{x}_j) = -(\mathbf{x} - \mathbf{x}_j)\cdot\mathbf{H}\,(\mathbf{x} - \mathbf{x}_j) < 0 , \tag{A.6}$$

where use has been made of (A.5). In order to search for a local minimum of a scalar function $f(\mathbf{x})$, (A.6) implies that a necessary condition for the value of the function to decrease during a full Newton's step is that the Hessian of the function must be positive definite. Further details on line search and backtracking can be found in Appendix B.

The variable metric method proceeds by rescaling

$$\hat{\mathbf{x}} = \mathbf{T}\mathbf{x} , \tag{A.7}$$

where $\mathbf{T}$ is a nonsingular matrix with the dimension of $\mathbf{x}$. Thus in the new variable space, the quadratic approximation to f about $\hat{\mathbf{x}}_j$ is

$$f(\hat{\mathbf{x}}) = f(\mathbf{x}_j) + \nabla f(\mathbf{x}_j)\cdot\mathbf{T}^{-1}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_j) + \frac{1}{2}\mathbf{T}^{-1}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_j)\cdot\mathbf{H}\,\mathbf{T}^{-1}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_j) , \tag{A.8}$$

or

$$f(\hat{\mathbf{x}}) = f(\mathbf{x}_j) + \nabla f(\mathbf{x}_j)\cdot\mathbf{T}^{-1}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_j) + \frac{1}{2}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_j)\cdot(\mathbf{T}^{-T}\mathbf{H}\,\mathbf{T}^{-1})(\hat{\mathbf{x}} - \hat{\mathbf{x}}_j) . \tag{A.9}$$

Since the Hessian must be symmetric and positive definite, the basic idea of the algorithm is to create a symmetric and positive definite approximation to the Hessian.

A common choice of $\mathbf{T}$ is

$$\mathbf{T} = \sqrt{\mathbf{H}}. \tag{A.10}$$

Observe that the scaling that leads to an identity Hessian in (A.9) at $\hat{\mathbf{x}}_j$, i.e.,

$$\mathbf{T}^{-T}\mathbf{H}\,\mathbf{T}^{-1} = \mathbf{I} , \tag{A.11}$$

implies (A.10). Following Greenstadt's updating formula [12]

$$\begin{aligned}\hat{\mathbf{H}}_{j+1}^{-1} &= \hat{\mathbf{H}}_j^{-1} + \frac{(\hat{\mathbf{s}}_{j+1} - \hat{\mathbf{H}}_j^{-1}\,\hat{\mathbf{y}}_{j+1})\otimes\hat{\mathbf{y}}_{j+1} + \hat{\mathbf{y}}_{j+1}\otimes(\hat{\mathbf{s}}_{j+1} - \hat{\mathbf{H}}_j^{-1}\,\hat{\mathbf{y}}_{j+1})}{\hat{\mathbf{y}}_{j+1}\cdot\hat{\mathbf{y}}_{j+1}} \\ &- \frac{[\hat{\mathbf{y}}_{j+1}\cdot(\hat{\mathbf{s}}_{j+1} - \hat{\mathbf{H}}_j^{-1}\,\hat{\mathbf{y}}_{j+1})]\,\hat{\mathbf{y}}_{j+1}\otimes\hat{\mathbf{y}}_{j+1}}{(\hat{\mathbf{y}}_{j+1}\cdot\hat{\mathbf{y}}_{j+1})^2} ,\end{aligned} \tag{A.12}$$

where

$$\hat{\mathbf{s}}_{j+1} = \hat{\mathbf{x}}_{j+1} - \hat{\mathbf{x}}_j = \mathbf{T}(\mathbf{x}_{j+1} - \mathbf{x}_j) = \mathbf{T}\,\mathbf{s}_{j+1} , \tag{A.13}$$

$$\hat{\mathbf{y}}_{j+1} \; = \; \mathbf{T}^{-1}(\nabla f_{j+1} \; - \; \nabla f_j) \; = \; \mathbf{T}^{-1}\,\mathbf{y}_{j+1} \; , \tag{A.14}$$

$$\hat{\mathbf{H}}_j^{-1} \; = \; \mathbf{T}\mathbf{H}_j^{-1}\,\mathbf{T}^T \; , \tag{A.15}$$

and

$$\hat{\mathbf{H}}_{j+1}^{-1} \; = \; \mathbf{T}\mathbf{H}_{j+1}^{-1}\,\mathbf{T}^T \; . \tag{A.16}$$

Notice that $\mathbf{x}_{j+1}$ can be updated by subtracting (A.5) at $\mathbf{x}_{j+1}$ from the same equation at $\mathbf{x}_j$:

$$\mathbf{x}_{j+1} \; - \; \mathbf{x}_j \; = \; \mathbf{H}_j^{-1}\,(\nabla f_{j+1} \; - \; \nabla f_j) \; , \tag{A.17}$$

where $\nabla f_j = \nabla f(\mathbf{x}_j)$ and ∇f_{j+1} is evaluated at $\mathbf{x} = \mathbf{x}_{j+1}$. This is the result obtained by the line searches and backtracking scheme [26] along the Newton's direction from $\mathbf{x}_j$.

Using the relations (A.13) to (A.16) to transform (A.12) into the original variable space, the *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* updating formula[1] is obtained

$$\begin{aligned} \mathbf{H}_{j+1}^{-1} \; &= \; \mathbf{H}_j^{-1} + \frac{\mathbf{s}_{j+1} \otimes \mathbf{s}_{j+1}}{\mathbf{s}_{j+1} \cdot \mathbf{y}_{j+1}} - \frac{(\mathbf{H}_j^{-1}\,\mathbf{y}_{j+1}) \otimes (\mathbf{H}_j^{-1}\,\mathbf{y}_{j+1})}{\mathbf{y}_{j+1} \cdot \mathbf{H}_j^{-1}\,\mathbf{y}_{j+1}} \\ &+ \; (\mathbf{y}_{j+1} \cdot \mathbf{H}_j^{-1}\,\mathbf{y}_{j+1})\,\mathbf{u} \otimes \mathbf{u} \; , \end{aligned} \tag{A.18}$$

where

$$\mathbf{u} \; = \; \frac{\mathbf{s}_{j+1}}{\mathbf{s}_{j+1} \cdot \mathbf{y}_{j+1}} \; - \; \frac{\mathbf{H}_j^{-1}\,\mathbf{y}_{j+1}}{\mathbf{y}_{j+1} \cdot \mathbf{H}_j^{-1}\,\mathbf{y}_{j+1}} \; . \tag{A.19}$$

Since each of these updates can be derived using a scaling of the variable space that is different at every iteration, the algorithm used above is called the *variable metric* method.

[1]Another alternative formulation which is known as the *Davidon-Fletcher-Powell (DFP)* algorithm differs from the BFGS scheme only in details of their roundoff error, convergence tolerances, etc. However, it has become generally recognized that the BFGS scheme is superior in these respects.

Appendix B

Line Search and Backtracking

Recall that the descent direction used in (A.5) need not decrease the function since the quadratic approximation may not be valid if the full Newton step has been taken. The descent direction only guarantees that *initially* the function f decreases as the point moves in that direction. The strategy for proceeding from an initial guess which is estimated to lie far from the root (even be outside the convergence region of Newton's method) is the method of line searches and backtracking [7].

The idea is that given a descent direction[1], say $\mathbf{p}$, an "acceptable" $\mathbf{x}_{j+1}$ is taken along that direction. That is,

$$\mathbf{x}_{j+1} \;=\; \mathbf{x}_j \;+\; \lambda_j\;\mathbf{p}_j\;, \quad 0 < \lambda_j \leq 1\;. \tag{B.1}$$

The term "line search" refers to the procedure for choosing λ_j in the previous equation. In order to take advantage of the fast convergence of Newton's method near the solution, it is important to take a full Newton step whenever possible. Thus, we take $\lambda = 1$ in the first attempt.

A simple acceptance rule for the new point $\mathbf{x}_{j+1}$ requires that

$$f(\mathbf{x}_{j+1}) < f(\mathbf{x}_j)\;. \tag{B.2}$$

However, this condition does not guarantee that $\mathbf{x}_j$ will converge to the minimizer of f in two cases[2]. The first case arises where the decrease in function values relative to the length of steps is too small. The first case can be remedied by ensuring

$$f(\mathbf{x}_{j+1}) \leq f(\mathbf{x}_j) \;+\; \alpha\lambda_j\;\nabla f(\mathbf{x}_j)\cdot\mathbf{p}_j\;, \tag{B.3}$$

where $\alpha = 10^{-4}$ (see reference [7] for details). This condition requires that the average rate of decrease $(f(\mathbf{x}_{j+1}) - f(\mathbf{x}_j))/\lambda_j$ of f be at least some prescribed fraction (i.e., α) of the

[1]The initial descent direction in MEDUSA is $-\nabla f(\mathbf{x}_0)$ since the identity Hessian has been used as the starting matrix from which we update the Hessian in (A.18).

[2]See [7] for examples of such cases.

initial rate of decrease[3] in that direction. The second case arises when the steps are too small relative to the initial rate of decrease of f. This problem can also be remedied by the use of a backtracking strategy which we now describe. First, define

$$\psi(\lambda) \;=\; f(\mathbf{x}_j \;+\; \lambda\mathbf{p}_j)\;. \tag{B.4}$$

The idea is that if the full Newton step is not acceptable, which means that backtracking is necessary, then λ is chosen by using the most current information about ψ such that the function $\psi(\lambda)$ is minimized. Initially, we have two pieces of information concerning $\psi(\lambda)$:

$$\psi(0) \;=\; f(\mathbf{x}_j) \quad \text{and} \quad \psi^{'}(0) \;=\; \nabla f(\mathrm{x_j})\cdot \mathrm{p_j}\;. \tag{B.5}$$

Since the Newton step is always attempted first, $\psi(1) = f(\mathbf{x}_j + \mathbf{p}_j)$ is also known. Thus, $\psi(\lambda)$ can be approximated by a quadratic function:

$$\tilde{\psi}(\lambda) \;=\; [\psi(1) \;-\; \psi(0) \;-\; \psi^{'}(0)]\lambda^2 \;+\; \psi^{'}(0)\lambda \;+\; \psi(0)\;. \tag{B.6}$$

The minimum of $\psi(\lambda)$ is attained when

$$\lambda \;=\; \lambda^{*} \;=\; -\frac{\psi^{'}(0)}{2[\psi(1) \;-\; \psi(0) \;-\; \psi'(0)]}\;, \tag{B.7}$$

for which $\psi^{'}(\lambda) = 0$. It can be shown that if the full Newton step fails, i.e., (B.3) is not satisfied, then the upper bound of λ is $\lambda \leq \frac{1}{2}$. On the other hand, if $\psi(1)$ is much larger than $\psi(0)$, λ can be very small; then $\lambda \geq 0.1$ is chosen to be the lower bound.

Suppose $\psi(\lambda) = f(\mathbf{x}_j + \lambda\ \mathbf{p}_j)$, where λ is calculated from (B.7), does not satisfy (B.3). In this case, the backtracking needs to be executed again. On the second and subsequent backtracks, $\psi(\lambda)$ is approximated as a cubic function of λ, using the previous value $\psi(\lambda_1)$ and the second most recent value $\psi(\lambda_2)$,

$$\psi(\lambda) \;=\; a\,\lambda^3 \;+\; b\,\lambda^2 \;+\; \psi^{'}(0)\,\lambda \;+\; psi(0)\;, \tag{B.8}$$

where

$$\begin{pmatrix} a \\ b \end{pmatrix} \;=\; \frac{1}{\lambda_1 \;-\; \lambda_2}\begin{pmatrix} 1/\lambda_1^2 & -1/\lambda_2^2 \\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2 \end{pmatrix}\begin{pmatrix} \psi(\lambda_1) - \psi^{'}(0)\lambda_1 - \psi(0) \\ \psi(\lambda_2) - \psi^{'}(0)\lambda_2 - \psi(0) \end{pmatrix}\;. \tag{B.9}$$

Its local minimizing point is

$$\lambda \;=\; \frac{-b \;+\; \sqrt{b^2 \;-\; 3a\;\psi'(0)}}{3a}\;. \tag{B.10}$$

A long, but straightforward, calculation shows that λ in (B.10) can never be imaginary if $\alpha < \frac{1}{4}$. Since we have previously chosen α to be 10^{-4}, λ will always be real.

[3]Since λ=1 is used in the first attempt of the step-acceptance criteria, the directional derivative of f at $\mathbf{x}_j$ in the direction $\mathbf{p}_j$ is the initial rate of decrease of f.

Appendix C

A Sample of the File *model.dat*

The following is a printout of a sample file *model.dat*. It defines two vehicle models that differ in the elastic properties of the chassis, i.e., in E and ν:

```
%
% A sample of physical parameters for two vehicle models
%
NUMBER_OF_MODELS 2

% description of Model 1 starts here
MODEL 1
 COSSERAT_POINT
  MASS 1573.0  % mass of car [kg]
  Ix    479.6  % moments of inertia along principal axes [kg m^2]
  Iy   2594.6
  Iz   2782.0
  E   600.0e6  % Young's modulus [N/m^2]
  nu  0.30     % Poisson's ratio
  volume 0.42  % assumed volume of the Chassis [m^3]
 SUSPENSION
  L1 1.034     % distance from cg to front axle [m]
  L2 1.491     % distance from cg to rear axle [m]
  B  1.2       % track of axle [m]
  H1 0.0       % vertical distance from cg to front assembly pts.
  H2 0.0       % and to rear assembly points [m] (assumed)
  spring_ref 0.15  % reference length of spring [m]
  C1 40000.0   % spring constant for front wheel suspension [N/m]
  C2 40000.0   % spring constant for rear wheel suspension [N/m]
  D1 1500.0    % damping coeff. for front wheel suspension [Ns/m]
  D2 1200.0    % damping coeff. for rear wheel suspension [Ns/m]
 TIRE 0.0016   % lag parameter for tire model [s]
 CONTACT
  A1 4.0       % dimensions of a vehicle: length [m]
  A2 1.6       % width [m]
  A3 1.3       % height [m]
 EQUILIBRIUM
  R3  5.039617e-02   % vertical position of vehicle's center of mass [m]
  D11  0.9972   D12  0.0   D13 -0.0748  % director 1 [.]
  D21  0.0      D22  1.0   D23  0.0     % director 2 [.]
  D31  0.0749   D32  0.0   D33  0.9972  % director 3 [.]

% description of model 1 ends and description of Model 2 starts

MODEL 2
 COSSERAT_POINT
  MASS 1573.0  % mass of car [kg]
  Ix    479.6  % moments of inertia along principal axes [kg m^2]
  Iy   2594.6
  Iz   2782.0
  E   200.0e7  % Young's modulus [N/m^2]
  nu  0.33     % Poisson's ratio
  volume 0.42  % assumed volume of the Chassis [m^3]
 SUSPENSION
  L1 1.034     % distance from cg to front axle [m]
  L2 1.491     % distance from cg to rear axle [m]
  B  0.725     % track of axle [m]
  H1 0.0       % vertical distance from cg to front assembly pts.
  H2 0.0       % and to rear assembly points [m] (assumed)
  spring_ref 0.15  % reference length of spring [m]
  C1 17000.0   % spring constant for front wheel suspension [N/m]
  C2 40000.0   % spring constant for rear wheel suspension [N/m]
  D1 1500.0    % damping coeff. for front wheel suspension [Ns/m]
  D2 1200.0    % damping coeff. for rear wheel suspension [Ns/m]
  TIRE 0.0016  % lag parameter for tire model [s]
  CONTACT
  A1 4.5       % dimensions of a vehicle: length [m]
  A2 2.5       % width [m]
  A3 1.5       % height [m]
 EQUILIBRIUM
  R3 0.03644925    % vertical position of vehicle's center of mass [m]
  D11  0.9972   D12  0.0   D13 -0.0748  % director 1 [.]
  D21  0.0      D22  1.0   D23  0.0     % director 2 [.]
  D31  0.0749   D32  0.0   D33  0.9972  % director 3 [.]
```

```
% description of model 2 ends here
```

Appendix D

Structure Definitions

D.1 The Vehicle Model Structure

Associated with each vehicle is a data structure that holds all its model parameters, as well as its state vector and other data:

```
typedef struct {
  int ident;                 /* number that identifies the vehicle */
  Vector z;                  /* state vector */
  struct {
    double m;                /* mass of car in kg */
    Matrix I;                /* inertia matrix */
    Matrix I_inv;            /* inverse inertia matrix */
    double lam, tm;          /* elastic properties of Cosserat point */
  } Cosserat_point;
  struct {
    double steer_angle;
    double X1[5], X2[5], X3[5]; /* coordinates of assembly points in m wrt. */
                             /* Xi[0] is void. */
    double spring_ref;       /* reference length of the suspension springs in m */
    double C[5], D[5];       /* spring and damping constants of the suspension */
                             /* in N/m. C[0] and D[0] are void. */
    Matrix infl;             /* Influence matrix */
  } suspension;
  struct {
    double tau_inv;          /* lag parameter for tire model in 1/s */
    double speed;            /* tire speed in m/s */
    int driven;              /* is the tire driven or undriven? */
  } tire;
  struct {
    int segment;             /* road segment number */
    double parameter;        /* position within the segment */
  } road;
  struct {
    double dimension_box[4];/* physical dimensions of vehicles */
    Vector force;            /* resultant constraint force */
    Matrix previous;        /* previous contact points with other vehicles */
    Matrix multiplier;      /* Lagrange multipliers */
  } contact;
  struct {                  /* equilibrium state */
    double r3;
    Matrix F;
  } init;
} vehicle_struct;
```

D.2 The Simulation Structure

Medusa uses a structure of the data type **simu_struct** to store data pertaining to the simulation. This structure is listed below:

```
typedef struct {
  char *in_file;                /* name of the platoon description file */
  FILE *ofp;                    /* scratch file */
  int NofV;                     /* number of vehicles in platoon */
  int NofM;                     /* number of models in model array */
  vehicle_struct *model;        /* model array */
  vehicle_struct *vehicle;      /* vehicle array, i.e., platoon */
  struct {
    double end_time;            /* simulation runs for end_time seconds */
    double delta_t;             /* integration step-size */
    double save_delta_t;        /* time intervals for saving a data point */
    double time;                /* current time step */
  } integrate;
  struct {                      /* road related parameters */
    int nodal_points;           /* number of nodes that define the road */
    Matrix XIN;                 /* nodal coordinates */
    Matrix *C       ;           /* road coefficients */
  } road;
} simu_struct;
```

Appendix E

Function Dependencies

- `cmderror()` .... *init.c*
- `detect_contact()` .... *contact.c*
  - `info()` .... *contact.c*
  - `eig()` .... *contact.c*
  - `angle()` .... *contact.c*
  - `min_search()` .... *contact.c*
  - `norm_angle()` .... *contact.c*
  - `pos()` .... *contact.c*
  - `norm()` .... *contact.c*
  - `dist()` .... *contact.c*
  - `d_dist1()` .... *contact.c*
  - `d_dist2()` .... *contact.c*
  - `d_dist()` .... *contact.c*
  - `pert()` .... *contact.c*
- `eig()` .... *contact.c*
  - `jacobi()` .... *contact.c*
  - `eigsrt()` .... *contact.c*
- `energy()` .... *vehicle.c*
- `equations_of_motion()` .... *vehicle.c*
  - `road()` .... *road.c*
  - `tire_forces()` .... *vehicle.c*
- `evaluate_cmd_line()` .... *init.c*
  - `cmderror()` .... *init.c*
  - `print_options()` .... *init.c*

- `find_token()` .... *init.c*
- `init()` .... *init.c*
  - `evaluate_cmd_line()` .... *init.c*
  - `read_models()` .... *init.c*
  - `read_vehicles()` .... *init.c*
  - `road_init()` .... *init.c*
- `integrate()` .... *main.c*
  - `equations_of_motion()` .... *vehicle.c*
  - `save_data_point()` .... *main.c*
  - `set_constraint_forces()` .... *vehicle.c*
  - `write_to_file()` .... *main.c*
- `main()` .... *main.c*
  - `init()` .... *init.c*
  - `integrate()` .... *main.c*
- `print_options()` .... *init.c*
- `pert()` .... *contact.c*
  - `func()` .... *contact.c*
  - `piksrt()` .... *contact.c*
  - `opp()` .... *contact.c*
  - `search()` .... *contact.c*
- `read_expr()` .... *init.c*
  - `find_token()` .... *init.c*
- `read_file()` .... *init.c*
  - `nrerror()` .... *common.h*
- `read_models()` .... *init.c*
  - `find_token()` .... *init.c*
  - `read_expr()` .... *init.c*
  - `read_file()` .... *init.c*
- `read_vehicles()` .... *init.c*
  - `find_token()` .... *init.c*
  - `read_expr()` .... *init.c*
  - `read_file()` .... *init.c*
- `road_init()` .... *init.c*

- `seg_init()` .... *init.c*
- `road()` .... *road.c*
  - `dist_road()` .... *road.c*
  - `d_road()` .... *road.c*
- `save_data_point()` .... *main.c*
  - `energy()` .... *vehicle.c*
  - `write_to_file()` .... *main.c*
- `seg_init()` .... *init.c*
- `set_constraint_forces()` .... *vehicle.c*
  - `detect_contact()` .... *contact.c*
- `tire_forces()` .... *vehicle.c*
- `write_to_file()` .... *main.c*

Appendix F

The Medusa Source Code

This appendix lists the complete source code of Medusa. The pages are individually numbered starting anew with page one for each file. The files are listed in this order:

common.h	2 pages
main.c	4 pages
init.c	10 pages
vehicle.c	6 pages
road.c	5 pages
contact.c	17 pages
common.c	8 pages

```
/*****************************************************************************/
/********                                                           ********/
/********                       common.h                            ********/
/********                                                           ********/
/*****************************************************************************/
/* created by Peter Varadi, last modified April 21, 1999, 118 lines          */
/*****************************************************************************/
/* This file defines some general tools and matrix operations. It also de-    */
/* fines the global structure data types simu_struct and vehicle_struct for   */
/* storing simulation and vehicle data.                                       */
/*****************************************************************************/

#define LANEWIDTH 3.0 /* The width of a lane in meters */

/* scalar product of two 3-vectors */
#define DOT3(x,y)  (x[1]*y[1]+x[2]*y[2]+x[3]*y[3])

#define FALSE 0
#define TRUE 1
#define Pi 3.1415926535898

double max(double x, double y);
double min(double x, double y);
double square(double x);
double dt(double x[], double y[]);
void nrerror(char error_text[]);           /* error handler */

typedef double *Vector;
typedef double **Matrix;
Vector vector(int n);                         /* allocate Vector */
Matrix matrix(int nrow, int ncol);      /* allocate Matrix */
void free_vector(Vector);
void free_matrix(Matrix);

typedef struct {          /* vehicle parameters */
  int ident;                    /* number that identifies the vehicle */
  Vector z;                           /* state vector */
  struct {
       double m;                      /* mass of car in kg */
       Matrix I;                      /* inertia matrix */
       Matrix I_inv;                  /* inverse inertia matrix */
       double lam, tm;       /* elastic properties of Cosserat point */
  } Cosserat_point;
  struct {
       double steer_angle;
       double X1[5], X2[5], X3[5]; /* coordinates of assembly points in m wrt.
*/
       /* Xi[0] is void. */
       double spring_ref;    /* reference length of the suspension springs in m
*/
       double C[5], D[5];    /* spring and damping constants of the suspension
*/
                                                   /* in N/m. C[0] and D[0] are
void. */
       Matrix infl;                 /* influence matrix */
  } suspension;
  struct {
       double tau_inv;       /* lag parameter for tire model in 1/s */
       double speed;                  /* tire speed in m/s */
       int driven;           /* is the tire driven or undriven? */
  } tire;
  struct {
       int segment;          /* road segment number */
       double parameter;     /* position within the segment */
  } road;
  struct {
       double dimension_box[4]; /* physical dimensions of vehicles */
       Vector force;                  /* resultant constraint force */
       Matrix previous;      /* previous contact points with other vehicles */
       Matrix multiplier;                  /* Lagrange multipliers */
  } contact;
  struct {                           /* equilibrium state */
    double r3;
    Matrix F;
  } init;
} vehicle_struct;

typedef struct {           /* simulation parameters */
  char *in_file;           /* name of the platoon description file */
  FILE *ofp;                     /* scratch file */
  int NofV;                      /* number of vehicles */
  int NofM;                      /* number of models in model array */
  vehicle_struct *model;   /* model array */
  vehicle_struct *vehicle;/* vehicle array, i.e., platoon */
  struct {
       double end_time;      /* duration of simulation */
       double delta_t;           /* integration stepsize */
       double save_delta_t;  /* time intervals for saving a data point */
       double time;                /* current time */
  } integrate;
  struct {                 /* road related parameters */
       int nodal_points;     /* number of nodes that define the road */
       Matrix XIN;           /* nodal coordinates */
       Matrix *C       ;     /* road coefficients */
  } road;
} simu_struct;

/**************************************************************************/
/******                  Vector and Matrix operations                ******/
/**************************************************************************/
double dot(Vector x, Vector y, int n);
       /* calculates scalar product <x,v>  */
void vector_product(Vector a, Vector b, Vector c);
       /* calculates a=b x c */
void matrix_times_vector(Vector y, Matrix A, Vector x, int nrow, int ncol);
       /* calculates y=Ax */
void matrix_times_matrix(Matrix Y, Matrix A, Matrix B, int M, int N, int P);
       /* calculates Y=AB */
void matrix_transpose(Matrix Y, Matrix A, int nrow, int ncol);
/* calculates transpose of Y */
void lin_solve(Matrix A, int n, Vector b);
       /* This function solves Ax=b for x and returns x in b. A is changed. */
void matrix_inverse(Matrix A, int n, Matrix Ainv);
       /* inverse Ainv of A */
```

```
/*******************************************************************************/
/******                       other operations                           ******/
/*******************************************************************************/
void minimize(Vector, int,
                          double (*f)(Vector), void (*df)(Vector, Vector),
Vector);
      /* solves for the local minimum of the function f(x). */
double projection(Matrix *, Vector, Vector, int);
```

```
/*************************************************************************/
/********                                                         *******/
/********                      main.c                             *******/
/********                                                         *******/
/*************************************************************************/
/* created by Peter Varadi, last modified June 30, 1999, 257 lines
*/
/*************************************************************************/
/*     MEDUSA final: integrator and output functions                    */
/*************************************************************************/
/* Global variables:          line 34                                    */
/*                                                                      */
/* Functions:                                                           */
/*   - main()                 line 40                                   */
/*   - integrate()            line 108                                  */
/*   - save_data_point()      line 182                                  */
/*   - write_to_file()        line 207                                  */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "common.h"

/* The state vector of a vehicle has 24 states for the Cosserat point and 4
states for the tires: */
#define STATES 28

/* The output is buffered KMAX times before it is written to the output file: */
#define KMAX 100

/********************/
/* Global Variables: */
/********************/
extern simu_struct simulation;
static FILE *smart_path_file, *director_file, *energy_file, *velocity_file;
static Matrix data;                    /* output data is stored in a Matrix */
static vehicle_struct *vehicle;     /* array of vehicle models */
static int nv;                      /* number of vehicles in the platoon */

int main(int argc, char *argv[])
/*  The main function calls the initialization function init(), initializes the
global variables and calls the integrator. The arguments argc and argv are
passed to init(). */
{
  time_t time1, time2;

  void init(int argc, char *argv[]);
  void integrate(void);

  /*****************************/
  /* Initialize simulation run: */
  /*****************************/
  init(argc,argv);

  vehicle=simulation.vehicle; /* substitution */
  nv=simulation.NofV;
  data=matrix(1+25*nv,KMAX);

  smart_path_file=fopen("path.asc","w");
  setvbuf(smart_path_file,NULL,_IOFBF,BUFSIZ);
  director_file=fopen("director.asc","w");
  setvbuf(director_file,NULL,_IOFBF,BUFSIZ);
  energy_file=fopen("energy.asc","w");
  setvbuf(energy_file,NULL,_IOFBF,BUFSIZ);
  velocity_file=fopen("velocity.asc","w");
  setvbuf(velocity_file,NULL,_IOFBF,BUFSIZ);

  /*****************************************/
  /* Print some information to the screen: */
  /*****************************************/
  printf("\n\n\n The simulation for %d vehicles will stop after %g [s]\n", nv,
            simulation.integrate.end_time);
  printf("\n\t- stepsize: %g [s]\n",simulation.integrate.delta_t);
  printf("\t- save data point every %g [s]\n\n",
       simulation.integrate.save_delta_t);

  /*******************/
  /* run simulation: */
  /*******************/
  time1=time(NULL);            /* record when simulation started */
  integrate();
  time2=time(NULL);            /* record when simulation ended */

  /*******************/
  /* postprocessing: */
  /******************/
  printf("\n start time=%s",asctime(localtime(&time1)));
  printf("\n end time=%s",asctime(localtime(&time2)));
  printf("\n The simulation took %e seconds\n",difftime(time2,time1));

  fclose(energy_file);
  fclose(velocity_file);
  fclose(director_file);
  fclose(smart_path_file);
  free_matrix(data);
  fclose(simulation.ofp);

  return(EXIT_SUCCESS);
}

/*******************************************************************************/
/******                                                                  ******/
/******                         Integration                              ******/
/******                                                                  ******/
/*******************************************************************************/
#define DT_MIN 5.e-6

void integrate(void)
/* This function integrates the equations of motion of the vehicles from t1=0 to
      t2 with a fixed step-size. The step-size is reduced if any two vehicles
are
      in contact. */
{
	double t=0.0, t2;		/* start and end times */
	double dt,dt_max;		/* step sizes */
	Matrix dz, h;			/* intermediate storage */
	int i, cv, contact, just_reduced=-1;

	int set_constraint_forces(void);
	void equations_of_motion(Vector,vehicle_struct *);
	void save_data_point(double t);
	void write_to_file(void);

	/* Initialize: */
	t2=simulation.integrate.end_time;
	dt_max=dt=simulation.integrate.delta_t;

	dz=matrix(nv,STATES);
	h=matrix(nv,STATES);
	for (cv=1;cv<=nv;cv++) for (i=1;i<=STATES;i++) h[cv][i]=vehicle[cv].z[i];
	/* The matrix h contains now a copy of all state vectors. h is used to 
trace
		the simulation back by one step if the step size needs to be 
reduced. */
	while (t<=t2) {
		simulation.integrate.time=t;
		save_data_point(t);	/* Make sure that initial conditions are 
stored. */

		/* calculate prediction of next integration step: */
		for (cv=1;cv<=nv;cv++) {
		  for (i=1;i<=12;i++) vehicle[cv].z[i]=h[cv][i]+dt*h[cv][i+12];
		  for (i=13;i<=STATES;i++) vehicle[cv].z[i]=h[cv][i];
		}

		/* Calculate constraint forces based on this prediction: */
		contact=set_constraint_forces();

		/* If no contact has occured or if integrator runs already on the 
smallest
			step size, proceed with integration. Else reduce the step size 
and
			start current integration step all over again: */
		if (!contact || dt<DT_MIN) {
			/* Calculate equations of motion and new state vectors: */
			for (cv=1;cv<=nv;cv++) {
				equations_of_motion(dz[cv],vehicle+cv);
				for (i=1;i<=STATES;i++) vehicle[cv].z[i]=(h[cv][i] += 
dt*dz[cv][i]);
			}
			t+=dt;
			/* If there was no contact for a while and the step size is 
small, it
				is increased again: */
			if (!contact && just_reduced<0 && dt<dt_max) {
				dt*=1.1;
				printf("t=%f\t delta_t=%f\n",t,dt);
			}
			just_reduced--;
		} else {
```

```
                  dt/=1.4641;          /* decrease step size */
                  just_reduced=100;       /* keep current step size for at least
80 steps */
                  printf("t=%f\t delta_t=%f\n",t,dt);  /* print current step size
*/
            }
      }
      write_to_file();
      free_matrix(h);
      free_matrix(dz);
}

/********************************************************************************/
/******                                                                    ******/
/******                              Output                                ******/
/******                                                                    ******/
/********************************************************************************/
/* When this counter reaches KMAX, then the output buffer output_data[] is full
      and needs to be written to the disk: */
static int count=0;

void save_data_point(double t)
/* This function saves a datapoint to the output data buffer whenever enough
      time has passed since the last time a data point was saved. The input t is
      the current time. */
{
      int cv, i;
      static double t_old= -100; /* Last saved timestep. This initialization
makes
                                                 the function save the initial
conditions to the file. */
      void write_to_file(void);
      double energy(vehicle_struct *);

      /* Has enough time passed?: */
      if ((t-t_old)>=0.999999*simulation.integrate.save_delta_t) {
            t_old=t;
            count++;
            data[1][count]=t;                   /* saves current time */
            for (cv=1;cv<=nv;cv++) {       /* saves data of every vehicle */
                  for (i=1;i<24;i++) data[25*cv-24+i][count]=vehicle[cv].z[i];
/* state */
                  data[cv*25+1][count]=energy(vehicle+cv);
/* energy */
            }
            printf("t=%.4f\n",t);                         /* write current time
to screen */
            if (count==KMAX) write_to_file();     /* write to file if buffer is
full */
      }
}

void write_to_file(void)
/* This function writes the output files using fprintf() commands. The global
      counter variable count is reset. */
{
      int i, cnt, cv;
      /* Write directors, velocities and energies to individual files: */
      for (cnt=1;cnt<=count;cnt++) {
            /* time is first on every line: */
            fprintf(director_file,"%e\t",data[1][cnt]);
            fprintf(energy_file,"%e\t",data[1][cnt]);
            fprintf(velocity_file,"%e\t",data[1][cnt]);

            /* now for every car: */
            for (cv=1;cv<=nv;cv++) {
                  for (i=1;i<=12;i++) {
                        fprintf(director_file,"%e\t",data[25*cv-24+i][cnt]); /*
directors */
                        fprintf(velocity_file,"%e\t",data[25*cv-12+i][cnt]);/*
velocities */
                  }
                  fprintf(energy_file,"%e\t",data[1+cv*25][cnt]);  /* energy */

                  /* SmartPATH data: */
                  fprintf(smart_path_file,"%e\t",data[1][cnt]); /* time */
                  fprintf(smart_path_file,"%d\t0\t0\t",cv);  /* vehicle number
*/
                  for (i=1;i<=3;i++)
                        fprintf(smart_path_file,"%e\t",data[25*cv-24+i][cnt]);/*
x y z */
                  fprintf(smart_path_file,"%e\t%e\t%e\t",
                        atan2(data[25*cv-19][count],data[25*cv-20][count]),
/* heading */
                        atan2((data[25*cv-14][count]*data[25*cv-20][count]+
/* pitch */
                        data[25*cv-19][count]*data[25*cv-13][count])/
                        sqrt(data[25*cv-20][count]*data[25*cv-20][count]+
                        data[25*cv-19][count]*data[25*cv-19][count]),
                        data[25*cv-12][count]),
                        atan2((data[25*cv-19][count]*data[25*cv-14][count]-
/* roll */
                        data[25*cv-20][count]*data[25*cv-13][count])/
                        sqrt(data[25*cv-20][count]*data[25*cv-20][count]+
                        data[25*cv-19][count]*data[25*cv-19][count]),
                        data[25*cv-12][count]));
                  fprintf(smart_path_file,"0\t");  /* segment ID */
                  fprintf(smart_path_file,"%e\t",
/* speed */
                        sqrt(data[25*cv-11][count]*data[25*cv-11][count]+
                        data[25*cv-10][count]*data[25*cv-10][count]));
                  fprintf(smart_path_file,"0 0 0 0 0 0 0 0\n");
/* stuff */
            }
            /* terminate lines: */
            fprintf(director_file,"\n");
            fprintf(velocity_file,"\n");
            fprintf(energy_file,"\n");
      }
      count=0;
}
```

```
/*****************************************************************************/
/********                                                             *******/
/********                        init.c                               *******/
/********                                                             *******/
/*****************************************************************************/
/* created by Peter Varadi and Gwo_jeng Lo, modified June 30, 1999, 615 lines */
/*****************************************************************************/
/* The code in this file reads the road data, the data for the vehicle models */
/* and the data for the platoon from separate files. The command line options */
/* are evaluated and the simulation is initialized.                           */
/*****************************************************************************/
/* Functions:                                                                 */
/*    - init()               line 51                                          */
/*    - evaluate_cmd_line()  line 80                                          */
/*    - print_options()      line 123                                         */
/*    - cmderror()           line 135                                         */
/*    - read_models()        line 148                                         */
/*    - read_vehicles()      line 262                                         */
/*    - read_file()          line 354                                         */
/*    - find_token()         line 411                                         */
/*    - read_expr()          line 433                                          */
/*    - road init()          line 458                                         */
/*    - seg_init()           line 580                                         */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include "common.h"

/* The position vector and the 3 directors of a Cosserat point have 12 vector
components. The state vector of a vehicle has 24+4 states (Cosserat point plus
tire model) : */
#define N 12
#define STATES 28

#define DEFAULT_INPUT_FILE  "platoon.dat"
#define MODEL_FILE "model.dat"
#define DELTA_T 0.00005     /* default stepsize of the integrator */
#define SAVE_DELTA_T  0.01  /* default time intervals for saving a data point */
#define TOKEN_LENGTH 20     /* maximal lenght of a keyword in the input files */

simu_struct simulation={DEFAULT_INPUT_FILE};

/*****************************************************************************/
/******                                                               ******/
/******                     Main Initialization                       ******/
/******                                                               ******/
/*****************************************************************************/
void init(int argc, char *argv[])
/* Main initialization function. argc and argv are the command line options that
      are passed down directly from main(). init() initializes the simulation
      structure, evaluates the command line options, sets up the road and the
      platoon. */
{
      void evaluate_cmd_line(int argc, char *argv[]);
      void read_models(void);
      void read_vehicles(void);
      void road_init(void);

      /* Initialize the global simulation structure: */
      simulation.integrate.end_time=0.0;
      simulation.integrate.delta_t=DELTA_T;
      simulation.integrate.save_delta_t=SAVE_DELTA_T;
      simulation.ofp=fopen("oop.asc","w");
      setvbuf(simulation.ofp,NULL,_IOFBF,BUFSIZ);

      evaluate_cmd_line(argc,argv);
      read_models();
      read_vehicles();
      road_init();
}

/*****************************************************************************/
/******                                                                ******/
/******                   read command line options                    ******/
/******                                                                ******/
/*****************************************************************************/
void evaluate_cmd_line(int argc, char *argv[])
/* Command line options are passed down from main() in argc and argv. The
      options are evaluated using a switch statement. Error messages are
generated
      for missing or unknown options. Refer to a C manual for reference on the
argv
      and argc variables. */
{
      int i;
      void print_options(void);
      void cmderror(char text[]);

      /* Refer to print_options() below for an explanation of these options: */
      for (i=1;i<argc;i++) {
            if (argv[i][0]=='-')
                  switch (argv[i][1]) {
                  case 'd' : if
(!sscanf(argv[i]+2,"%lf",&simulation.integrate.delta_t))
                                          cmderror(argv[i]);
                                    break;
                  case 'f' : simulation.in_file=argv[i]+2;
                                    break;
                  case 'h' : print_options();
                                    break;
                  case 's' : if (!sscanf(argv[i]+2,"%lf",

      &simulation.integrate.save_delta_t))
                                          cmderror(argv[i]);
                                    break;
                  case 't' : if
(!sscanf(argv[i]+2,"%lf",&simulation.integrate.end_time))
                                          cmderror(argv[i]);
                                    break;
                  default : cmderror(argv[i]);
            } else {
                  printf("\n\n unknown command line option %s .",argv[i]);
                  printf(" Type %s -h for help\n\n",argv[0]);
                  exit(EXIT_FAILURE);
            }
      }
      /* An error message is generated if the simulation ending time (option -t)
      was not specified: */
      if (simulation.integrate.end_time==0.0) {
            printf("\n\nNo endtime specified. Type %s -h for help\n\n",argv[0]);
            exit(EXIT_FAILURE);
      }
}

void print_options(void)
/* Prints command line otions and exits program. */
{
      printf("\n\n The command line options are:\n\n");
      printf("   -h  prints this list\n");
      printf("   -dx.xxx  set fixed stepsize to x.xxx [s] (default:
%g)\n",DELTA_T);
      printf("   -ffile  parameter file (default: %s)\n",DEFAULT_INPUT_FILE);
      printf("   -tx.xx  simulation ends at x.xx [s] (mandatory)\n");
      printf("   -sx.xx  save data point every x.xx [s] (default:
%g)\n",SAVE_DELTA_T);
      exit(EXIT_FAILURE);
}

void cmderror(char text[])
/* Generic error handler for evaluate_cmd_line(). */
{
      printf("\n\n error in command line option %s .",text);
      printf(" Use option -h for help\n\n");
      exit(EXIT_FAILURE);
}

/******************************************************************************/
/******                                                                  ******/
/******                 Read vehicle models from file                    ******/
/******                                                                  ******/
/******************************************************************************/
void read_models(void)
/* Reads the number of models and the individual models from the file
      model.dat */
{
      vehicle_struct *v;      /* short for simulation.model */
      int i,j,k,m;
      char *buffer, *ptr;      /* ptr points to character inside buffer[] */
      char str[]="D11";

      /* variables for intermediate results: */
      double Ix, Iy, Iz, E, nu, vol, I[4][4];
      Matrix M;

      char *read_file(char *filename);
      double read_expr(char token[], char **ptr);
      void find_token(char token[], char **ptr);
      /* Read the reduced file contents into the buffer array: */
      ptr=buffer=read_file(MODEL_FILE);
      /* ptr points to start of buffer array */

      simulation.NofM=(int) read_expr("NUMBER_OF_MODELS",&ptr);

      /* Allocate memory for the models (v[0] is void): */
      v=simulation.model=(vehicle_struct *)
                  malloc((size_t) ((simulation.NofM+1)*sizeof(vehicle_struct)));
      if (!v) nrerror("allocation failure in read_models()");

      /* Read models: */
      for (m=1;m<=simulation.NofM;m++) {
            if(m!=(int) read_expr("MODEL",&ptr))
                  nrerror("Wrong Model Number in file model.dat");
            v[m].ident=m;                                           /* store
model number */
            find_token("COSSERAT_POINT",&ptr); /* Cosserat point related
material: */
            /* Read data: */
            v[m].Cosserat_point.m=read_expr("MASS",&ptr);
            Ix=read_expr("IX",&ptr);
            Iy=read_expr("IY",&ptr);
            Iz=read_expr("IZ",&ptr);
            E=read_expr("E",&ptr);
            nu=read_expr("NU",&ptr);
            vol=read_expr("VOLUME",&ptr);

            /* Calculate material constants: */
            v[m].Cosserat_point.lam=0.5*vol*E*nu/(1+nu)/(1-2*nu);
            v[m].Cosserat_point.tm=0.5*vol*E/(1+nu);

            /* Calculate inertia matrix and its inverse: */
            v[m].Cosserat_point.I=M=matrix(N,N);
            for (i=1;i<=N;i++) for (j=1;j<=N;j++) M[i][j]=0.0;       /* zero
matrix */
            M[1][1]=M[2][2]=M[3][3]=v[m].Cosserat_point.m;
            M[4][4]=M[5][5]=M[6][6]=0.5*(-Ix+Iy+Iz);
            M[7][7]=M[8][8]=M[9][9]=0.5*(Ix-Iy+Iz);
            M[10][10]=M[11][11]=M[12][12]=0.5*(Ix+Iy-Iz);

            v[m].Cosserat_point.I_inv=matrix(N,N);
            matrix_inverse(M, N, v[m].Cosserat_point.I_inv);

            find_token("SUSPENSION",&ptr);      /* Suspension related material:
*/
            /* Read data: */
            v[m].suspension.X1[1]=v[m].suspension.X1[2]=  read_expr("L1",&ptr);
            v[m].suspension.X1[3]=v[m].suspension.X1[4]= -read_expr("L2",&ptr);
            v[m].suspension.X2[2]=v[m].suspension.X2[4]=
                  -(v[m].suspension.X2[1]=v[m].suspension.X2[3]=
                        0.5*read_expr("B",&ptr));
            v[m].suspension.X3[1]=v[m].suspension.X3[2]= -read_expr("H1",&ptr);
            v[m].suspension.X3[3]=v[m].suspension.X3[4]= -read_expr("H2",&ptr);
            v[m].suspension.spring_ref=read_expr("SPRING_REF",&ptr);
            v[m].suspension.C[1]=v[m].suspension.C[2]=read_expr("C1",&ptr);
            v[m].suspension.C[3]=v[m].suspension.C[4]=read_expr("C2",&ptr);
            v[m].suspension.D[1]=v[m].suspension.D[2]=read_expr("D1",&ptr);
            v[m].suspension.D[3]=v[m].suspension.D[4]=read_expr("D2",&ptr);

            /* Calculate influence matrix: */
            v[m].suspension.infl=matrix(N,N);
            for (i=1;i<=N;i++) for (j=1;j<=N;j++)
v[m].suspension.infl[i][j]=0.0;

            I[0][0]=I[0][1]=I[0][2]=I[0][3]=1.0;
            for (i=0;i<=3;i++) {
                  I[1][i]=v[m].suspension.X1[i+1];
                  I[2][i]=v[m].suspension.X2[i+1];
                  I[3][i]=v[m].suspension.X3[i+1];
            }
            for (i=0;i<=3;i++) for (j=0;j<=3;j++) for (k=1;k<=3;k++)     /* fill
in */
                  v[m].suspension.infl[3*i+k][3*j+k]=I[i][j];

            /* Read remaining data for current model: */
            v[m].tire.tau_inv=1.0/read_expr("TIRE",&ptr);

            find_token("CONTACT",&ptr);
            v[m].contact.dimension_box[1]=read_expr("A1",&ptr);
            v[m].contact.dimension_box[2]=read_expr("A2",&ptr);
            v[m].contact.dimension_box[3]=read_expr("A3",&ptr);

            find_token("EQUILIBRIUM",&ptr);
            /* Define the equilibrium state of the vehicle: */
            v[m].init.r3=read_expr("R3",&ptr);   /* Height of center of mass */
            v[m].init.F=matrix(3,3);                         /* Deformation
gradient */
            for (i=1;i<=3;i++) {
                  str[1]='0'+i;
                  for (j=1;j<=3;j++) {
                        str[2]='0'+j;
                        v[m].init.F[j][i]=read_expr(str,&ptr);
                  }
            }
      }
      free(buffer);                  /* Free memory allocated by read_file */
}

/*****************************************************************************/
/******                                                                ******/
/******                  Read platoon data from file                   ******/
/******                                                                ******/
/*****************************************************************************/
void read_vehicles(void)
/* Read vehicle models from the file simulation.in_file. */
{
      vehicle_struct *vehicle;     /* short for simulation.vehicle */
      char *buffer, *ptr;     /* ptr points to a character inside buffer[] */
      char error_msg[]="allocation failure in read_vehicles()";
      int NofV;                            /* Number of vehicles */
      int i,j,k,m,v;
      /* intermediate variables: */
      double theta, Fij, vel;
      Matrix Q=matrix(3,3);

      char *read_file(char *filename);
      double read_expr(char token[], char **ptr);
      void find_token(char token[], char **ptr);

      /* Read the reduced file contents into the buffer array: */
      ptr=buffer=read_file(simulation.in_file);
      /* ptr points to start of buffer. */

      NofV=simulation.NofV=(int) read_expr("NUMBER_OF_VEHICLES",&ptr);

      /* Allocate an array of vehicles: */
      simulation.vehicle=vehicle=
            (vehicle_struct *) malloc((size_t)
((NofV+1)*sizeof(vehicle_struct)));
      if (!vehicle) nrerror(error_msg);

      /*******************************************************/
      /* Associate vehicles and initial conditions with a model: */
      /*******************************************************/
      for (v=1;v<=NofV;v++) {
            m=(int) read_expr("VEHICLE_HAS_MODEL",&ptr);

            /* Each vehicle is a copy of a model and all vehicles of a certain
model
            share some memory through pointers! */
            if (m>0 && m<=simulation.NofM) {
                  vehicle[v]=simulation.model[m];  /* copy model structure */
                  vehicle[v].ident=v;            /* store vehicle number */
                  /* allocate individual memory: */
                  vehicle[v].z=vector(2*N+4);
                  vehicle[v].contact.force=vector(2*N);
                  vehicle[v].contact.previous=matrix(NofV,2);
                  vehicle[v].contact.multiplier=matrix(NofV,2);
                  /* initialize: */
                  for (i=1;i<=NofV;i++) for (j=1;j<=2;j++) {
                        vehicle[v].contact.multiplier[i][j]=0.0;
                        vehicle[v].contact.previous[i][j]=Pi/4.0;
                  }
            } else {
                  printf("\n\n Model nummer %d does not exist\n\n",m);
                  exit(EXIT_FAILURE);
            }
            find_token("INITIALLY_WITH",&ptr);

            /* Position of center of mass: */
            vehicle[v].z[1]=read_expr("X",&ptr);
            vehicle[v].z[2]=read_expr("Y",&ptr);
            vehicle[v].z[3]=simulation.model[m].init.r3;

            /* orientation means a rigid body rotation about the vertical axis:
*/
            theta=read_expr("ORIENTATION",&ptr);
            theta *=Pi/180;
            Q[1][1]=Q[2][2]=cos(theta);          /* rotation matrix */
            Q[2][1]=sin(theta);
            Q[1][2]= -Q[2][1];
            Q[1][3]=Q[3][1]=Q[2][3]=Q[3][2]=0.0;
            Q[3][3]=1.0;
            for (i=1;i<=3;i++) for (j=1;j<=3;j++) {   /* F=QF_equilibrium */
                  Fij=0.0;
                  for (k=1;k<=3;k++)
Fij+=Q[i][k]*simulation.model[m].init.F[k][j];
                  vehicle[v].z[3*j+i]=Fij;      /* copy F to the state vector */
            }
            /* Velocity of the center of mass: */
            vel=read_expr("SPEED",&ptr);
            vehicle[v].z[13]=vel*cos(theta);
            vehicle[v].z[14]=vel*sin(theta);
            for (j=15;j<=STATES;vehicle[v].z[j++]=0.0);
            /* tire speed */
            vehicle[v].tire.speed=read_expr("TIRE_SPEED",&ptr);
            vehicle[v].tire.driven= (fabs(vehicle[v].tire.speed)<=0.1) ? FALSE :
TRUE;
      /* Steer angle: */

      vehicle[v].suspension.steer_angle=read_expr("STEERING",&ptr)*Pi/180.;
      }
      free_matrix(Q);
      free(buffer);           /* free memory allocated by read_file */
}

/******************************************************************************/
/******                                                                  ******/
/******                              Supplements
       ******/
/******                                                                  ******/
/******************************************************************************/
char *read_file(char *filename)
/* Read the contents of the file 'filename' into a string whose adress is then
returned. Sequences of white space characters are reduced to a single space.
Everything between a '%' character and the following newline character is
considered a comment and ignored. The return string is allocated using malloc().
The corresponding memory must be freed by the calling function. The term 'white
space' is defined in any C manual. */
{
      FILE *file;
      int filelength;
      char *buffer, *ptr;           /* ptr points to character inside buffer[]
*/
      char c;                             /* Temporary storage place for
current character. */
      int space=FALSE;              /* Indicates contnuous white space to be
ignored. */
      int comment=FALSE;            /* Indicates a comment which is ignored. */

      /* Open file for reading: */
      if ((file=fopen(filename,"r"))== NULL) nrerror("cannot open model file");
      setvbuf(file,NULL,_IOFBF,BUFSIZ);

      /* determine length of file and create a buffer of that size: */
      for (filelength=0;fgetc(file)!=EOF;filelength++);
      ptr=buffer=(char *) malloc((size_t) ((filelength+3)*sizeof(char)));
      if (!buffer) nrerror("allocation failure in read_file()");

      rewind(file);           /* back to start of file */

      /* copy file to buffer and ignore comments, multiple spaces, etc.: */
      while ((c=fgetc(file))!=EOF) {      /* Read character from file. */
            if (c=='%') {      /* This indicates a comment. Ignore everything */
                  comment=TRUE;         /* until newline character is found. */
                  if (!space) {         /* A comment is trated as white space.
*/
                        space=TRUE;
                        *ptr++ =' ';
                  }
                  continue;
            }
            if (comment) {                      /* Ignore comments. */
                  if (c=='\n') comment=FALSE;  /* The end of the line ends a
comment. */
                  continue;
            }
            if (isspace(c)) {
                  if (!space) {
                        space=TRUE;             /* A sequence of white space
characters is */
                        *ptr=' ';               /* reduced to single space in
the buffer. */
                        ptr++;
                  }
                  continue;
            }
            if (!comment) *ptr++ =c;      /* copy character to buffer */
            space=FALSE;
      }
      *ptr++=' ';                               /* add a space as break for
other functions */
      *ptr='\0';                  /* end of string */
      fclose(file);
      return buffer;
}

void find_token(char token[], char **ptr)
/* This function reads a word from the position that *ptr points to and compares
it to token. If there is a match, then after execution, *ptr points to the
character following the word. If there is no match, the program is aborted and
an error message is generated. A space character leading the word is ignored.
The word ends with the occurrence of a space character. */
{
      char word[TOKEN_LENGTH+1];
      int i=0;

      if (**ptr==' ') *ptr+=1;  /* skip a leading space character */
      while (**ptr!=' ') {                /* Read word */
            word[i++]=toupper(**ptr);  /* Uppercase */
            *ptr+=1;                   /* Advance one character */
      }
      word[i]='\0';           /* terminate string */
      if (strcmp(token,word)) {
            printf("\n\n Unexpected token %s. %s expected.\n\n",word,token);
            exit(EXIT_FAILURE);
      }
}

double read_expr(char token[], char **ptr)
/* This function works as find_token() (see description there) but also returns
the number which follows the token. The number has to be in the that can be
converted by atof() (see C manual for reference). */
{
      int i=0;
      char word[31];
      void find_token(char token[], char **ptr);

      find_token(token,ptr);        /* get a match for the token */
      if (**ptr==' ') *ptr+=1;      /* skip a leading space character */

      while (**ptr!=' ') {                /* Read word */
            word[i++]=**ptr;
            *ptr+=1;                                  /* Advance one character
*/
      }
      word[i]='\0';                             /* terminate string */
      return atof(word);                  /* conversion */
}

/******************************************************************************/
/******                                                                 ******/
/******                       Initialize Road                           ******/
/******                                                                 ******/
/******************************************************************************/
void road_init(void)
/* Initialize the road section of the simulation structure. In particular,
   XC[NP+1][4], YC[NP+1][4] and ZC[NP+1][4] are the coefficients of cubic
   polynominals for NP+1 segments, where NP is the number of nodal points. */
{
      char dummy[200];
      Matrix XIN, *C;         /* Coefficients of cubic polynomials */
      int NP;                 /* number of nodal points */
      int i, j, k;
      double p;
      FILE *road_geo, *ofp;

      int seg_init(Vector);

      /* Read data from file road.dat */
      if((road_geo=fopen("road.dat","r"))==NULL) nrerror("Cannot open
road.dat");
      fscanf(road_geo,"%s",dummy);
      if (strlen(dummy) !=17) nrerror("Input data error in road.dat.");
      fscanf(road_geo,"%d",&NP);

      simulation.road.nodal_points=NP;
      XIN=simulation.road.XIN=matrix(NP+2,4); /* coordinates of NP nodal points
*/

      /* Allocate matrix array C, the coefficients of cubic polynomials where
```

```
            there are NP+1 segment numbers: */
      simulation.road.C=C=(Matrix *) malloc((size_t) (4*sizeof(Matrix)));
      if (!C) nrerror("allocation failure in road_init():C");
      for (i=1;i<=3;i++) C[i]=simulation.road.C[i]=matrix(NP+1,4);

      fscanf(road_geo,"%s",dummy);
      if (strlen(dummy) != 24) nrerror("Input data error in road.dat.");

      /* Read the 2nd to (NP+1)th points from road.dat. */
      /* The first and NP+2 points are virtual points. */
      /*************************************************************************
**/
      /*     1   2   3   4   5   6 .....      NP           original nodal points
*/
      /* 1---2---3---4---5---6---7 ..... ----NP+1----NP+2  modified nodal points
*/
      /*   1   2   3   4   5   6   .....  NP     NP+1      segment number
*/
      /*************************************************************************
**/
      for (i=2;i<=NP+1;i++)
            fscanf(road_geo,"%lf %lf %lf
%lf",XIN[i]+1,XIN[i]+2,XIN[i]+3,XIN[i]+4);

      fscanf(road_geo,"%s",dummy);
      if (strlen(dummy) != 11) nrerror("Input data error in road.dat.");
      fclose(road_geo);

      /* add virtual points at both ends: */
      XIN[1][1]=XIN[2][1]-10.*(XIN[3][1]-XIN[2][1]);
      XIN[1][2]=XIN[2][2]-10.*(XIN[3][2]-XIN[2][2]);
      XIN[1][3]= XIN[2][3]; /* same height as that of first nodal point. */
      XIN[1][4]=0.0;
      XIN[NP+2][1]=XIN[NP+1][1]+10.*(XIN[NP+1][1]-XIN[NP][1]);
      XIN[NP+2][2]=XIN[NP+1][2]+10.*(XIN[NP+1][2]-XIN[NP][2]);
      XIN[NP+2][3]= XIN[NP+1][3]; /* same height as that of last nodal point. */
      XIN[NP+2][4]=0.0;

      /* let the first segment be a straight line. */
      for (j=1;j<=3;j++) {
            C[j][1][4]=XIN[1][j];
            for (i=1;i<=3;i++) C[j][1][i]=0.0;
      }
      p= (XIN[1][1]+XIN[2][1])/3;
      C[1][1][3]=-11*XIN[1][1]/2+9*p-9*p+XIN[2][1];
      C[1][1][2]=9*XIN[1][1]-45*p/2+18*2*p-9*XIN[2][1]/2;
      C[1][1][1]=(-9*XIN[1][1]+27*p-27*2*p+9*XIN[2][1])/2;

      /****************************************************/
      /* Compute the cubic polynomial:                    */
      /* X(s)=XC[k][1]s^3+XC[k][2]s^2+XC[k][3]s+XC[k][4]=0 */
      /* by the fisrt four nodal points in road.dat, where */
      /* k is the segment number.                         */
      /****************************************************/
      for (i=1;i<=3;i++) {
            /* The coefficients of 2nd curve segment */
            C[i][2][4]=XIN[2][i];
            C[i][2][3]=-11*XIN[2][i]/2+9*XIN[3][i]-9*XIN[4][i]/2+XIN[5][i];
            C[i][2][2]=9*XIN[2][i]-45*XIN[3][i]/2+18*XIN[4][i]-9*XIN[5][i]/2;
            C[i][2][1]=(-9*XIN[2][i]+27*XIN[3][i]-27*XIN[4][i]+9*XIN[5][i])/2;
            /* Coefficients of 3rd and 4th segments are the same as for the 2nd:
*/
            for (j=1;j<=4;j++) C[i][3][j]=C[i][4][j]=C[i][2][j];
      }
      /*******************************************************/
      /* Compute remaining cubic polynomials by the continuty */
      /* of nodal points, first and second derivatives.       */
      /*******************************************************/
      for (j=1;j<=3;j++) for (i=5;i<=NP+1;i++) {
            C[j][i][4]= XIN[i][j];
            C[j][i][3]= C[j][i-1][3]+2*C[j][i-1][2]+3*C[j][i-1][1];
            C[j][i][2]= C[j][i-1][2]+3*C[j][i-1][1];
            C[j][i][1]= XIN[i+1][j]-XIN[i][j]-C[j][i-1][3]-3*C[j][i-1][2]-
6*C[j][i-1][1];
      }
      /***********************************************/
      /* Output the coefficients of polynomials       */
      /***********************************************/
      ofp= fopen("co","w");
      for (i=2;i<=NP+1;i++) {
            for (k=1;k<=3;k++) for (j=1;j<=4;j++)
fprintf(ofp,"%f\t",C[k][i][j]);
            fprintf(ofp,"\n\n");
      }
      fclose(ofp);
      /***********************************************/
      /* Determine the initial segment number and the */
      /* corresponding parameter of each vehicle      */
      /***********************************************/
      for (i=1;i<=simulation.NofV;i++)   /* initial guess of parameter s: */
            switch (simulation.vehicle[i].road.segment
                             =seg_init(simulation.vehicle[i].z)) {
            case 3: simulation.vehicle[i].road.parameter=1.0/3.0;
                          break;
            case 4: simulation.vehicle[i].road.parameter=2.0/3.0;
                          break;
            case 5: simulation.vehicle[i].road.parameter=1.0;
                          break;
            default: simulation.vehicle[i].road.parameter=0.0;
         }
      /* convert angle in road.dat from degree to radian: */
      for (i=2;i<=NP+1;i++) XIN[i][4] *= Pi/180.;
}

/*****************************************************************************/
/******                   Segment initialization                         *****/
/*****************************************************************************/
int seg_init(Vector r)
/* Returns the initial segment number of the vehicle. Vector r is the
      position of vehicle. */
{
      Matrix XIN=simulation.road.XIN;
      Matrix *C=simulation.road.C;
      int i, k;
```

```
	double dist_23, dist_34, rad1, rad2, rad3, rad4, rad5;
	Matrix cen=matrix(6,3);  /* coordinates of the centers of the spheres */

	/* Initially, a vehicle has to be within the first four segments. Thus,
		define the five spheres by the first five nodal points: */
	for (i=1;i<=3;i++) for (k=1;k<=5;k++) cen[k][i]=XIN[k][i];
	rad1=dt(cen[1],cen[2]);
	rad2= min(rad1, dist_23=dt(cen[2],cen[3]));
	rad3= min(dist_23, dist_34=dt(cen[3],cen[4]));
	rad4= min(dist_34, rad5=dt(cen[4],cen[5]));

	/* Is vehicle inside any of the spheres? */
	if (dt(r,cen[1]) <= rad1) k=1;
	else if (dt(r,cen[5]) <= rad5) k=4;
	else if (dt(r,cen[2]) <= rad2) k=(projection(C,cen[2],r,2) < 0.0) ? 1 : 2;
	else if (dt(r,cen[3]) <= rad3) k=(projection(C,cen[3],r,3) < 0.0) ? 2 : 3;
	else if (dt(r,cen[4]) <= rad4) k=(projection(C,cen[4],r,4) < 0.0) ? 3 : 4;
	else if ((dist_34>sqrt(rad3*rad3-LANEWIDTH*LANEWIDTH)+sqrt(rad4*rad4-
LANEWIDTH*LANEWIDTH))
				 || (min(rad3,rad4)<dist_34) ) {
		/* If the spheres 3 & 4 don't overlap over at least the lanewidth,
then a
			sixth sphere is needed: */
		for (i=1;i<=3;i++) cen[6][i]=(XIN[3][i]+XIN[4][i])/2;
		if (dt(r,cen[6])<=dt(cen[3],cen[4])/2) k=3;
		else nrerror("Vehicle position on road cannot be determined!");
	}
	else nrerror("The initial position of vehicle is not in the first four
segments!");
	free_matrix(cen);
	return k;
}
```

```
/*****************************************************************************/
/********                                                             *******/
/********                          vehicle.c   v2.0
*******/
/********                                                             *******/
/*****************************************************************************/
/* created by Peter Varadi, last modified June 30, 1999,  lines 377          */
/*****************************************************************************/
/* The code in this file calculates                                          */
/*    - the constraint forces acting on the individual vehicles when they    */
/*         are in contact with each other
*/
/*    - the equations of motion of a vehicle                                 */
/*    - the total energy of a vehicle                                        */
/*****************************************************************************/
/* Functions:                                                                */
/* - set_constraint_forces()     line 39                                     */
/* - equations_of_motion()       line 135                                    */
/* - tire_forces()               line 287                                    */
/* - energy()                    line 340                                    */
/*****************************************************************************/
#include <stdio.h>
#include <math.h>
#include "common.h"

extern simu_struct simulation;

/* These definitions reflect the 12 vector components and 24 states of the
position vector and the directors of the Cosserat point: */
#define N 12
#define twoN 24

/*****************************************************************************/
/*******                                                              ******/
/*******                  calculate contact forces                    ******/
/*******                                                              ******/
/*****************************************************************************/
#define CONSTRAINT_MULTIPLIER1 2e4 /* for updating the Lagrange multipliers */
#define CONSTRAINT_MULTIPLIER2 100

int set_constraint_forces(void)
/* This function repeatedly calls detect_contact() (defined in contact.c) to
   determine if any two vehicles are in contact. If so, the function calculates
   the constraint forces acting on these vehicles. The function returns TRUE if
   any vehicles are in contact, FALSE otherwise. */
{
      vehicle_struct *vehicle=simulation.vehicle;   /* substitutions */
      Vector z1, z2, c1, c2, gamma;
      Matrix F=matrix(3,3);
        double previous[5];               /* contact points from previous
time_step */
      double normal[4];                   /* surface normal vector */
      double rho1[4], rho2[4];      /* relative position vectors of contact
points */
      double X1[4], X2[4];                /* Cosserat labels of the contact
points */
      double phi1, phi2, n1, n2;
      int i,j, return_value=FALSE, cv, av;

      int detect_contact(Vector, vehicle_struct *,vehicle_struct *,
                                     Vector, Vector, Vector);

      for (cv=1;cv<=simulation.NofV;cv++) for (i=1;i<=twoN;i++)
            vehicle[cv].contact.force[i]=0.0;             /* reset contact
forces */

      for (cv=1;cv<=simulation.NofV;cv++) for (av=cv+1;av<=simulation.NofV;av++)
{

      /* substitutions: */
      z1=vehicle[cv].z;
      z2=vehicle[av].z;
      c1=vehicle[cv].contact.force;
      c2=vehicle[av].contact.force;
      gamma=vehicle[cv].contact.multiplier[av];

      previous[1]=vehicle[cv].contact.previous[av][1];
      previous[2]=vehicle[cv].contact.previous[av][2];
      previous[3]=vehicle[av].contact.previous[cv][1];
      previous[4]=vehicle[av].contact.previous[cv][2];

      /* Are the the two vehicles in contact? */
      if (detect_contact(previous,vehicle+cv,vehicle+av,normal,rho1,rho2)) {
            return_value=TRUE;

            /* position constraint: */
            phi1=(rho1[1]-rho2[1])*normal[1]+(rho1[2]-rho2[2])*normal[2]+
                    (rho1[3]-rho2[3])*normal[3];

            /* labels X^i for the contact point on each of the Cosserat points:
*/
            for (i=1;i<=3;i++) {
                  X1[i]=rho1[i];
                  X2[i]=z1[i]-z2[i]+rho2[i];
            }
            for (i=1;i<=3;i++) for (j=1;j<=3;j++) F[i][j]=z1[3*j+i];
            lin_solve(F,3,X1);
            for (i=1;i<=3;i++) for (j=1;j<=3;j++) F[i][j]=z2[3*j+i];
            lin_solve(F,3,X2);

            /* Calculate velocity constraint: */
            phi2=0.0;
            for (i=1;i<=3;i++)
              phi2+=(z1[12+i]+X1[1]*z1[15+i]+X1[2]*z1[18+i]+X1[3]*z1[21+i]-
                              z2[12+i]-X2[1]*z2[15+i]-X2[2]*z2[18+i]-
X2[3]*z2[21+i])*normal[i];

            gamma[1]-=CONSTRAINT_MULTIPLIER1*phi1;
            gamma[2]-=CONSTRAINT_MULTIPLIER2*phi2;

            /* Calculate contact forces: */
            for(i=1;i<=3;i++) {
                  n1=gamma[1]*normal[i];
                  n2=gamma[2]*normal[i];
```

```
                    c1[i]+=n1;
                    c2[i]-=n1;
                    c1[12+i]+=n2;
                    c2[12+i]-=n2;
                    for (j=1;j<=3;j++) {
                          c1[3*j+i]+=X1[j]*n1;
                          c2[3*j+i]-=X2[j]*n1;
                          c1[12+3*j+i]+=X1[j]*n2;
                          c2[12+3*j+i]-=X2[j]*n2;
                    }
              }
              } else gamma[1]=gamma[2]=0.0;  /* no contact */
              /* store for next time-step: */
              vehicle[av].contact.multiplier[cv][1]=gamma[1];
              vehicle[av].contact.multiplier[cv][2]=gamma[2];
              vehicle[cv].contact.previous[av][1]=previous[1];
              vehicle[cv].contact.previous[av][2]=previous[2];
              vehicle[av].contact.previous[cv][1]=previous[3];
              vehicle[av].contact.previous[cv][2]=previous[4];
        }
        free_matrix(F);
        return return_value;
}

/********************************************************************************/
/******                                                                   ******/
/******     calculate equations of motion of a single vehicle             ******/
/******                                                                   ******/
/********************************************************************************/
void equations_of_motion(Vector dzdt, vehicle_struct *vcl)
/* This function calculates the time derivative dzdt of a vehicle's state
      vector. Input is the pointer vcl to the vehicle data. */
{
      Vector r=vcl->z, v=vcl->z+12;
      Vector d1=vcl->z+3, d2=vcl->z+6, d3=vcl->z+9;
      Vector w1=vcl->z+15, w2=vcl->z+18, w3=vcl->z+21;

      double a, laged_alpha, S;       /* Slip and  laged slip for the tire */
      double vi[4];                               /* velocity of an assembly
point */
      double d11, d12, d13, d22, d23, d33, lam, tm; /* constitutive quantities
*/

      /* temporary variables for projection, steering, suspension and tires: */
      double sinphi=sin(vcl->suspension.steer_angle);
      double cosphi=cos(vcl->suspension.steer_angle);
      double dummy_vector[N+1], dumdum, diff[4], Fx, Fy, Fz;
      double e1[4], e2[4], e3[4], exp[4], eyp[4], ex[4], ey[4], ez[4], tp[4],
vt[4];

      double k[N+1]={0.0}, forces[N+1]={0.0};       /* intrinsic and applied
forces */

      /* placeholders for variables within the simulation structure: */
      Matrix M_inverse=vcl->Cosserat_point.I_inv;
      Matrix influence_matrix=vcl->suspension.infl;
      double *X1=vcl->suspension.X1, *X2=vcl->suspension.X2, *X3=vcl-
>suspension.X3;
      double *C=vcl->suspension.C, *D=vcl->suspension.D;
      double spring_ref=vcl->suspension.spring_ref;

      long double dummy;             /* Long double reduces numerical error. */
      int i,j;

      void road(Vector normal, Vector tangent, Vector tp, vehicle_struct *v);
      void tire_forces(double *, double *, double, double, double);

      road(e3,e1,tp,vcl);  /* road vector triad */
      vector_product(e2,e3,e1);

      dumdum=DOT3(d1,e3);          /* rear wheel heading vectors */
      for (i=1;i<=3;i++) {
            exp[i]=d1[i]-dumdum*e3[i];
            ez[i]= -e3[i];
      }
      dumdum=sqrt(DOT3(exp,exp));
      for (i=1;i<=3;exp[i++]/=dumdum);
      vector_product(eyp,ez,exp);

      /**********************************************/
      /* Tire and suspension forces for each wheel: */
      /**********************************************/
      for (j=1;j<=4;j++) {
        if (j>2) for (i=1;i<=3;i++) {           /* wheel basis vectors */
              ex[i]=exp[i];
              ey[i]=eyp[i];
        } else for (i=1;i<=3;i++) {
             ex[i]=cosphi*exp[i]-sinphi*eyp[i];
             ey[i]=sinphi*exp[i]+cosphi*eyp[i];
        }
        laged_alpha=vcl->z[24+j];     /* lagged slip angle of the current tire
*/

        /* Calculate vt (=\tilde{v} from the velocity vi[] of the assembly
point: */
        for (i=1;i<=3;i++) vi[i]=v[i]+X1[j]*w1[i]+X2[j]*w2[i]+X3[j]*w3[i];
        dumdum=DOT3(vi,e3);
        for (i=1;i<=3;i++) vt[i]=vi[i]-dumdum*e3[i];

        /* suspension force: */
        for (i=1;i<=3;i++) diff[i]=r[i]+X1[j]*d1[i]+X2[j]*d2[i]+X3[j]*d3[i]-
tp[i];
        Fz= -(C[j]*(DOT3(diff,e3)-spring_ref)+D[j]*DOT3(vi,e3)); /* normal force
*/
        if (Fz<0.0) printf("\n Tire %d lift off!",j);

        /* wheel forces: */
        if (Fz>0.0) {
             S= (vcl->tire.driven && j<3) ? 1.0-(vcl->tire.speed)/(DOT3(vt,ex))
: 0.0;
             a=atan2(DOT3(vt,ey),DOT3(vt,ex)); /* slip angle for next time step
*/
```

```
             dzdt[24+j]=vcl->tire.tau_inv*(a-laged_alpha); /* tire state vector
*/
             tire_forces(&Fx,&Fy,Fz,laged_alpha,S);
        } else Fx=Fy=Fz=0.0;

        for (i=1;i<=3;i++) forces[i+3*(j-1)]=Fx*ex[i]-Fy*ey[i]-Fz*ez[i];
      }

      /*************************/
      /* Constitutive Equations */
      /*************************/
      d11=DOT3(d1,d1)-1.0;
      d12=DOT3(d1,d2);
      d13=DOT3(d1,d3);
      d22=DOT3(d2,d2)-1.0;
      d23=DOT3(d2,d3);
      d33=DOT3(d3,d3)-1.0;
      tm=vcl->Cosserat_point.tm;
      lam=vcl->Cosserat_point.lam*(d11+d22+d33);
      for (i=1;i<=3;i++) {
            k[i]=0.0;
            k[3+i]=lam*d1[i]+tm*(d11*d1[i]+d12*d2[i]+d13*d3[i]);
            k[6+i]=lam*d2[i]+tm*(d12*d1[i]+d22*d2[i]+d23*d3[i]);
            k[9+i]=lam*d3[i]+tm*(d13*d1[i]+d23*d2[i]+d33*d3[i]);
      }

      /****************************/
      /* Build return vector dzdt: */
      /****************************/
      for (i=1;i<=N;i++) {
            dzdt[i]=vcl->z[N+i];
            dummy=0.0;
            for (j=1;j<=N;j++) dummy+=influence_matrix[i][j]*forces[j];
            dummy_vector[i]=dummy-k[i];
      }
      dummy_vector[3] -= 9.81*vcl->Cosserat_point.m;   /* gravity */

      for (i=1;i<=N;i++) {      /* multiply with inverse mass matrix */
            dummy=0.0;
            for (j=1;j<=N;j++) dummy+=M_inverse[i][j]*dummy_vector[j];
            dzdt[i+N]=dummy;
      }
      for(i=1;i<=twoN;i++) dzdt[i]+=vcl->contact.force[i];   /* contact force */
}

/****************************************************************************/
/******                        tire model                             ******/
/****************************************************************************/

/* Goodyear 185SR14 */
#define A0 7092.7808          /* 1583.21*4.48 */
#define A1 11.94
#define A2 13571.0848   /* 3029.26*4.48 */
#define A3 0.264        /* check dimensions! */
#define A4 -1765.46     /* check dimensions! */

#define B1 -0.0000254464285714286   /* -1.140E-04/4.48 */
```

```
#define B3 1.007
#define B4 -5.2913743622449e-11            /* -1.062E-09/4.48/4.48 */

#define K1 -2.595E-04    /* check dimensions! */
#define K2 2.198E-04     /* check dimensions! */
#define K3 0.073         /* check dimensions! */

#define SNT 85   /* test skid number */
#define SNP 85   /* pavement skid number */

#define CSFZ 18
#define Kmu 0.2

#define FZT 1160  /* tire design load at operating pressure (lbs) */
#define TW 5             /* tread width (inches) */
#define Tp 28            /* tire pressure (psi) */

/* bias ply tire: */
#define Ka 0.2
#define c1 0.535
#define c2 1.05
#define c3 1.15
#define c4 0.8

void tire_forces(double *Fx, double *Fy, double Fz,
                                 double slip_angle, double slip_ratio)
/* Calculates the tire longitudinal (Fx) and side (Fy) force using the modified
      STI/CalSpan tire model. The inputs are the slip_angle, longitudinal slip
      slip_ratio and the normal force Fz. */
{
      double s=slip_ratio, alpha=slip_angle;
      double sig, f, Kc_prime, mu, Ks, Kc, mu0, ap0;
      double sina, cosa, tana, S1, S2, S3;

      /* The formulas work only applicable for slip ratios between -1 and 1 and
for
      slip angles smaller than 90 degrees. We need to extrapolate: */
      if (fabs(s)>1) s=(-1.0);
      if (fabs(alpha) > .5*Pi) alpha=Pi-alpha;

      if (fabs(s)==0.0 && fabs(alpha)==0.0) *Fx=*Fy=0.0;
      else {
            tana=tan(alpha);
            sina=sin(alpha);
            cosa=cos(alpha);

            /* tire contact patch length: */
            ap0=0.0768*sqrt(Fz*FZT)/(TW*(Tp+5.0));

            /* lateral and longitudinal stiffness coefficients: */
            Ks=2.0/(ap0*ap0)*(A0+A1*Fz-A1/A2*Fz*Fz);
            Kc=2.0/(ap0*ap0)*Fz*CSFZ;

            /* peak tire/road coefficient of friction: */
            mu0=(B1*Fz+B3+B4*Fz*Fz)*SNP/SNT;

            /* slip to slide transition: */
            S3=sqrt(sina*sina+s*s*cosa*cosa);
            Kc_prime=Kc+(Ks-Kc)*S3;
            mu=mu0*(1.0-Kmu*S3);

            /* composite slip and force saturation function: */
            S1=Ks*Ks*tana*tana;
            sig=Pi*ap0*ap0/(8*mu0*Fz)*sqrt(S1+Kc*Kc*s*s/((1-s)*(1-s)));
            f=sig*(sig*(c1*sig+c2)+4.0/Pi)/(sig*(sig*(c1*sig+c3)+c4)+1.0);

            /* side and longitudinal force: */
            S2=mu*Fz*f/sqrt(S1+Kc_prime*Kc_prime*s*s);
            *Fy= S2*Ks*tana;
            *Fx= (-S2*Kc_prime*s);
      }
}

/*******************************************************************************/
/*******                                                                  ******/
/*******            total energy of the vehicle                           ******/
/*******                                                                  ******/
/*******************************************************************************/
double energy(vehicle_struct *vehicle)
/* The total energy of a vehicle is calculated. The vehicle parameters are in
      the structure vehicle. */
{
      Vector r=vehicle->z, d1=vehicle->z+3, d2=vehicle->z+6, d3=vehicle->z+9;
      double *C=vehicle->suspension.C;
      double *X1=vehicle->suspension.X1;
      double *X2=vehicle->suspension.X2;
      double *X3=vehicle->suspension.X3;
      double s_ref=vehicle->suspension.spring_ref;
      Matrix M=vehicle->Cosserat_point.I;
      long double dummy[N+1];
      double d11, d12, d13, d22, d23, d33, energy=0.0;
      int i,j;

      /* Calculate kinetic energy of Cosserat point: T=1/2 v.Mv */
      for (i=4;i<=N;i++) {
            dummy[i]=0.0;
            for (j=4;j<=N;j++) dummy[i]+=M[i][j]*vehicle->z[N+j];
      }
      for (i=4;i<=N;i++) energy+=0.5*dummy[i]*vehicle->z[i+N];

      /* Add stored energy of cosserat point: */
      d11 = DOT3(d1,d1)-1.0;
      d12 = DOT3(d1,d2);
      d13 = DOT3(d1,d3);
      d22 = DOT3(d2,d2)-1.0;
      d23 = DOT3(d2,d3);
      d33 = DOT3(d3,d3)-1.0;
      energy+=0.25*(vehicle->Cosserat_point.lam*(d11+d22+d33)*(d11+d22+d33)+
                                vehicle-
>Cosserat_point.tm*(d11*d11+d22*d22+d33*d33)+
                              2*vehicle-
>Cosserat_point.tm*(d12*d12+d13*d13+d23*d23));
      /* Add energy stored in the suspensions: */
      for (i=1;i<=4;i++)
```

```
            energy+=0.5*(C[i]*square(r[3]+X1[i]*d1[3]+X2[i]*d2[3]+X3[i]*d3[3]-
s_ref));
      /* Add potential energy of gravity: */
      return energy+vehicle->Cosserat_point.m*9.81*r[3];
}
```

```
/**************************************************************************/
/********                                                          *******/
/********                        road.c
*******/
/********                                                          *******/
/**************************************************************************/
/* created by Gwo-Jeng Lo and Peter Varadi, modified June 30, 1999, 290 lines */
/**************************************************************************/
/* The functions in this file  compute the road surface.                 */
/**************************************************************************/
/* Functions:                                                             */
/*   - road()                   line 27                                   */
/*   - range_error()            line 245                                  */
/*   - dist_road()               line 255                                  */
/*   - d_road()                  line 271                                  */
/**************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "common.h"
#include <string.h>

extern simu_struct simulation;

/**************************************************************************/
/******                  Orientation of the road                    *****/
/**************************************************************************/
void road(Vector normal, Vector tangent, Vector tp, vehicle_struct *v)
/* Calculates the road tangent and the road normal at the tracking point tp of
      the road spline. The input is the vehicle structure *v. */
{
  int NP=simulation.road.nodal_points;
  Matrix XIN=simulation.road.XIN;
  Matrix *C=simulation.road.C;
  Vector r=v->z;
  int i, j, k=v->road.segment, vn=v->ident;
  double s=v->road.parameter;

  int node=FALSE; /* Flag indicates if the vehicle is very close to a node. */

  double rad1, rad2, rad3, rad4, pro, angle;
  Matrix cen=matrix(5,3); /* coordinates of the centers of the spheres */
  double tan[4], q[4], si[14];
  double s_a, c_a, s_t, c_t, s_b, alpha, dist_23, c_s,s_s;
  double dummy;

  double dist_road(Vector);
  void d_road(Vector, Vector);
  void range_error(int, int);

  /************************************************************************/
  /*****   Determine in which road segment k the vehicle is           *****/
  /************************************************************************/
  /* From the previous segment number, say seg=P, we consider only nodes P-1,
  P, P+1 and P+2 since the position of vehicle assumed could only locate in
  current or neighboring segments for the next time step. Each of these four
  nodes is the center of a sphere. The radius of sphere P-1 is the distance
  between the nodes P-1 & P. The radius of sphere P+2 is the distance between
  the nodes P+1 & P+2. The radius of sphere P is the smaller of the two
  distances between the nodes P-1 & P and P & P+1, respectively. And similarly
  for the P+1 sphere. If the P and P+1 spheres don't overlap, we add a fifth
  sphere in between. */

  /*****************************************************/
  /*  Is vehicle in the first segment (a virtual one)? */
  /*****************************************************/
  if (k==1){
       for (i=1;i<=3;i++) cen[3][i]=XIN[k+1][i];
       pro=projection(C,cen[3],r,2);
       if (fabs(pro)<1.e-12) {
            node=TRUE;
            s=0.0;
       } else if (pro > 0.){
            for (i=1;i<=3;i++) cen[1][i]=XIN[k][i];
            if (dt(r,cen[1])> dt(cen[1],cen[3])) {
              /* real segment number is one less than the modified segment
number */
              printf("\nVehicle %d moves forward to segment %d.\n",vn,k);
              k++;
              s=0.0;
            } else s=1.0; /* k+=0; */
       } else s=1.0; /* k+=0; */
       /***********************************************/
       /* Is vehicle on the road defined by the user? */
       /***********************************************/
  } else {
       /* Define four spheres: */
       for (i=1;i<=3;i++) for (j=1;j<=4;j++) cen[j][i]=XIN[k-2+j][i];
       rad1= dt(cen[1],cen[2]);
       rad2= min(rad1,dist_23=dt(cen[2],cen[3]));
       rad3= min(dist_23,rad4=dt(cen[3],cen[4]));
       /**************************************/
       /* Is vehicle inside the first sphere? */
       /**************************************/
       if (dt(r,cen[1]) <= rad1) {
            k--;
            printf("\nVehicle %d moves backward to segment %d.\n",vn,k-1);
            if (k>4) s=1.0;

      /*********************************************************************/
            /* Is vehicle inside the fourth sphere and outside the second
sphere? */

      /*********************************************************************/
       } else if (dt(r,cen[4]) <= rad4 && dt(r,cen[2])>dist_23 ) {
            printf("\nVehicle %d moves forward to segment %d.\n",vn,k);
            k++;
            if (k>4) s=0.0;
            /***************************/
            /* If we have four spheres: */
            /***************************/
       } else if (!(dist_23>sqrt(rad2*rad2-LANEWIDTH*LANEWIDTH)+
             sqrt(rad3*rad3-LANEWIDTH*LANEWIDTH)) || (min(rad2,rad3)<dist_23)) {
            /*******************************************************/
            /*  Is vehicle in the region close to the nodal point k?  */
            /*********************************************************/
            if (dt(r,cen[2]) <= rad2 && dt(r,cen[3]) > rad3) {
              pro=projection(C,cen[2],r,k);
              /* Check if the tracking point is right on the nodal point k: */
              if (fabs(pro)<1.e-12) {
                   node=TRUE;
                   s=0.0;
              } else if (pro < 0.) {
                   k--;
                   printf("\nVehicle %d moves backward to segment %d.\n",vn,k-
1);
                   if (k>4) s=1.0;
              } /* else k+=0; */
              /*********************************************************/
              /*  Is vehicle in the region close to the nodal point k+1?  */
              /*********************************************************/
            } else if (dt(r,cen[3]) <= rad3 && dt(r,cen[4]) > rad4 ) {
              pro=projection(C,cen[3],r,k+1);
              /* Check if the tracking point is right on the nodal point k+1: */
              if (fabs(pro)<1.e-12) {
                   node=TRUE;
                   s=1.0;
              } else if (pro>0.0 && dt(r,cen[2])> dist_23){
                   printf("\nVehicle %d moves forward to segment %d.\n",vn,k);
                   k++;
                   if(k>4) s=0.0;
              } /* else k+=0; */

/*******************************************************************/
              /* If vehicle isn't in the region of segment k, print error
message*/

/*******************************************************************/
            } else if (!(dt(r,cen[2]) <= rad2 && dt(r,cen[3]) <= rad3 ))
                  range_error(1,vn);
            /*****************************/
            /* If we need a fifth sphere: */
            /*****************************/
       } else {
            for (i=1;i<=3;i++) cen[5][i]=(XIN[k][i]+XIN[k+1][i])/2;
            if (dt(r,cen[5]) > dt(cen[2],cen[3])/2) {
              /*********************************************************/
              /* Is vehicle in the region close to the nodal point k?  */
              /*********************************************************/
              if (dt(r,cen[2]) <= rad2) {
                   pro=projection(C,cen[2],r,k);
                   /* Check if the tracking point is right on the nodal point k:
*/
                   if (fabs(pro)<1.e-12) {
                        node=TRUE;
                        s=0.0;
                   } else if (pro < 0.0) {
                        k--;
                        printf("\nVehicle %d moves backward to segment
%d.\n",vn,k);
                        if(k>4) s=1.0;
                   } /* else k+=0; */
                   /*********************************************************/
                   /* Is vehicle in the region close to the nodal point k+1? */
                   /*********************************************************/
              } else if (dt(r,cen[3]) <= rad3) {
                   /* compute the tangent vector at nodal point k+1:*/
                   if (k!=NP-1) for (i=1;i<=3;i++) tan[i]=C[i][k+1][3];
                   else  for (i=1;i<=3;i++)
tan[i]=3*C[i][k][1]+2*C[i][k][2]+C[i][k][3];
                   for (i=1;i<=3;i++) q[i]=r[i]-cen[3][i];
                   pro=DOT3(q,tan);
                   /* Check if the tracking point is right on the nodal point
k+1: */
                   if (fabs(pro)<1.e-12) {
                        node=TRUE;
                        s=1.0;
                   } else if (pro > 0. && dt(r,cen[2])> dist_23){
                        printf("\nVehicle %d moves forward to segment
%d.\n",vn,k);
                        k++;
                        if(k>4) s=0.0;
                   } /* else k+=0; */
              } else range_error(2,vn);
            } else range_error(3,vn);
       }
  }
  if (k<1) range_error(4,vn);
  if (k>NP) {
       printf("\n\n r=%lf %lf %lf\n",r[1],r[2],r[3]);
       printf("The vehicle %d reaches the end of the road!\n",vn);
       exit(EXIT_FAILURE);
  }
  /******************************************************************/
  /*****   Determine the tracking point within the road segment:   *****/
  /******************************************************************/
  /* The tracking point on the road corresponds to the smallest distance from
  the vehicle to the road curve. */
  if (!node) {
       /* parameters for minimize() and  dist_road() and d_road() */
       si[1]=s;
       for (i=1;i<=4;i++) for (j=1;j<=3;j++) si[4*j-3+i]=C[j][k][i];
       si[5]-=r[1];
       si[9]-=r[2];
       si[13]-=r[3];
       minimize(si,1,dist_road,d_road,si);
       s=si[1];

       if (s<-0.0001 || s>1.0001){
            printf("k=%d s=%lf v=%d \n",k,s,vn);
            printf("\nWarning! Tracking of vehicle lost!\n");
            printf("Please modify the %dth data point in file: road.dat.\n",k);
            nrerror("\n");
       }
  }
  /* cubic interpolation: compute tracking point and tangent vector: */
  c_s= s*s*s; s_s=s*s;
  for (i=1;i<=3;i++) {
       tp[i]= C[i][k][1]*c_s+C[i][k][2]*s_s+C[i][k][3]*s+C[i][k][4];
       tangent[i]= 3*C[i][k][1]*s_s+2*C[i][k][2]*s+C[i][k][3];
  }
  tangent[1]= fabs(tangent[1]);
  /* and make it a unit tangent vector: */
  dummy=sqrt(DOT3(tangent,tangent));
  for (i=1;i<=3;tangent[i++]/=dummy);

  /*****************************************************/
  /* compute the unit normal vector of the road plan: */
  /*****************************************************/
  /* linear interpolation of banking angle: */
  /* positive: cw, negative: ccw viewed from the origin of tangent vector */
  if(k==1) angle= 0.0;
  if(k==2) angle= 3*(XIN[k+1][4]-XIN[k][4])*s+XIN[k][4];
  if(k==3) angle= 3*(XIN[k+1][4]-XIN[k][4])*s+2*XIN[k][4]-XIN[k+1][4];
  if(k==4) angle= 3*(XIN[k+1][4]-XIN[k][4])*s+3*XIN[k][4]-2*XIN[k+1][4];
  if(k>4) angle= (XIN[k+1][4]-XIN[k][4])*s+XIN[k][4];
  s_t=sin(angle); c_t=cos(angle);
  alpha=atan(tangent[2]/tangent[1]);
  s_a=sin(alpha); c_a=cos(alpha); s_b= tangent[3];
  normal[1]= -c_t*s_b*c_a+s_t*s_a;
  normal[2]= -c_t*s_b*s_a-s_t*c_a;
  normal[3]= c_t*sqrt(square(tangent[1])+square(tangent[2]));
  free_matrix(cen);

  v->road.segment=k;
  v->road.parameter=s;
}

void range_error(int k, int n)
{
  printf("\n\nErr %d: vehicle %d is out of road range!\n",k,n);
  printf("Now exiting to System!\n\n");
  exit(EXIT_FAILURE);
}

/*******************************************************************************/
/******                       Distance function                          ******/
/*******************************************************************************/
double dist_road(Vector x)
/* Input x[1] is the natural parameter of the tracking point on the road.
      dist_road() returns the distance between the mass center of vehicle and
the
      corresponding tracking point on the road. x[2]-x[16] are used to pass the
      values XC[1]-XC[4], YC[1]-YC[4] and ZC[1]-ZC[4] for the current segment.
*/
{
  double q=x[1], sq=q*q, cq=sq*q;

  return sqrt(square(x[2]*cq+x[3]*sq+x[4]*q+x[5])+
                  square(x[6]*cq+x[7]*sq+x[8]*q+x[9])+
                  square(x[10]*cq+x[11]*sq+x[12]*q+x[13]));
}

/*******************************************************************************/
/******                       Gradient function                          ******/
```

```
/*****************************************************************************/
void d_road(Vector g, Vector x)
{
  /* Input x[1] is the natural parameter of the tacking point on the road.
        Output g is the gradient at the tracking point x[1].  x[2]-x[13] are
used
        to pass the values XC[1]-XC[4], YC[1]-YC[4] and ZC[1]-ZC[4] for the
current
        segment. */

  int i;
  double q=x[1], sq=q*q, cq=sq*q;
  double dummy[4], s=0.0;

  for (i=1;i<=3;i++) dummy[i] = cq*x[4*i-2]+sq*x[4*i-1]+q*x[4*i]+x[4*i+1];
  for (i=1;i<=3;i++) s += dummy[i]*dummy[i];

  g[1]=(3*sq*x[2]+2*q*x[3]+x[4])*dummy[1]+
       (3*sq*x[6]+2*q*x[7]+x[8])*dummy[2]+
       (3*sq*x[10]+2*q*x[11]+x[12])*dummy[3];

  g[1] /= sqrt(s);
}
```

```
/*****************************************************************************/
/********                                                            *******/
/********                      Contact Model                         *******/
/********                                                            *******/
/*****************************************************************************/
/* created by Gwo-Jeng Lo, last modified on June 30 1999, 1123 lines          */
/*****************************************************************************/
/* This file contains 24 functions which do two major jobs:                  */
/*   (I)  detecting the contact situations of two vehicles,                  */
/*   (II) computing the positions of contact points on two vehicles and the   */
/*        direction of contact force acting on the vehicle 2 by vehicle 1.    */
/*****************************************************************************/
/* Functions:                                                                 */
/* - detect_contact()      line 66                                            */
/*   - info()              line 255                                           */
/*   - eig()               line 282                                           */
/*   - jacobi()            line 296                                           */
/*   - eigsrt()            line 380                                           */
/*   - enorm()             line 407                                           */
/*   - pos()               line 555                                           */
/*   - angle()             line 519                                           */
/*   - norm_angle()        line 541                                           */
/*   - dist()              line 595                                           */
/*   - d_dist1()           line 621                                           */
/*   - d_dist2()           line 674                                           */
/*   - d_dist()            line 725                                           */
/*   - pert()              line 802                                           */
/*   - func()              line 935                                           */
/*   - piksrt()            line 959                                           */
/*   - opp()               line 995                                           */
/*   - search()            line 1052                                          */
/*****************************************************************************/
#include <stdio.h>
#include <math.h>
#include "common.h"

#define EP 1.e-4
#define EP1 5.e-7
#define MAXSTEP 20
#define SHIFT1 5.e-2
#define SHIFT2 1.e-6
#define DEV 2e-4
#define DEV1 1.5e-4
#define VARY_X1X2 1
#define VARY_X3X4 2
#define VARY_X 3
#define EXP 0.4 /* Cannot be assigned as an integer. */
#define STEP1MAX 300
#define STEP2MAX 300
#define FREE_MATRIX free_matrix(KHAT1); free_matrix(KHAT2); free_matrix(v1);
free_matrix(v2); free_matrix(F1); free_matrix(F2);

extern simu_struct simulation;
/*****************************************************************************/
/* Global Variables                                                           */
/*****************************************************************************/
static double xbar[4]={0.0};            /* relative position of the vehicles  */
static double A[4], B[4], null[4];
static double d1[4], d2[4], len1[4], len2[4];/* eigenvalues of the ellipsoids */
static Matrix KHAT1, KHAT2, v1, v2;

/*******************************************************************************/
/********                                                                *******/
/********                   main detection procedure                     *******/
/********                                                                *******/
/*******************************************************************************/
int detect_contact(Vector state, vehicle_struct *car1, vehicle_struct *car2,
                                     Vector n1, Vector r1, Vector r2)
/* Detect contact point between two cars. Inputs are the state={u1,v1,u2,v2} of
the previously found contact point and vehicles car1 and car2 point to the
vehicle geometries. Outputs are position vectors r1, r2 of the contact points
and the normal vector n at that point. Returns TRUE when contact happens, else
FALSE. */
{
  int i,j,step1,step2;
  double delt=10.0*EP;
  double rold1[4],rold2[4],deep[4],en1[4],en2[4],temp[4];
  double check,pt;

  Matrix F1=matrix(3,3),F2=matrix(3,3); /* deformation gradients */

  void pos(Vector, Vector, Vector);
  void enorm(Vector, Vector, Vector);
  void eig(Matrix, int, Vector, Matrix);
  void info(Vector, Matrix, Matrix);
  int pert(Vector, double);
  double dist(Vector);
  void d_dist(Vector,Vector);
  void d_dist1(Vector,Vector);
  void d_dist2(Vector,Vector);
  void norm_angle(Vector,int);

  /* n1 is defined in vehicle.c */
  KHAT1=matrix(3,3); KHAT2=matrix(3,3); v1=matrix(3,3); v2=matrix(3,3);

  /* Read the deformation gradients of the two vehicles: */
  F1[1][1]=car1->z[4] , F1[1][2]=car1->z[7] , F1[1][3]=car1->z[10];
  F1[2][1]=car1->z[5] , F1[2][2]=car1->z[8] , F1[2][3]=car1->z[11];
  F1[3][1]=car1->z[6] , F1[3][2]=car1->z[9] , F1[3][3]=car1->z[12];

  F2[1][1]=car2->z[4] , F2[1][2]=car2->z[7] , F2[1][3]=car2->z[10];
  F2[2][1]=car2->z[5] , F2[2][2]=car2->z[8] , F2[2][3]=car2->z[11];
  F2[3][1]=car2->z[6] , F2[3][2]=car2->z[9] , F2[3][3]=car2->z[12];

  for(i=1;i<=3;i++) xbar[i]=car2->z[i]-car1->z[i];

  A[1]=car1->contact.dimension_box[1]/2;
  A[2]=car1->contact.dimension_box[2]/2;
  A[3]=car1->contact.dimension_box[3]/2;
  B[1]=car2->contact.dimension_box[1]/2;
  B[2]=car2->contact.dimension_box[2]/2;
  B[3]=car2->contact.dimension_box[3]/2;

  /* Check the potential contact of two vehicles: */
  if (dt(car2->z,car1->z) <
(max(A[1],max(A[2],A[3]))+max(B[1],max(B[2],B[3]))+1.e-8)){
       info(A, F1, KHAT1);
       info(B, F2, KHAT2);

       eig(KHAT1, 3, d1, v1); /* d1[i]=1/(semi-axes)^2 */
       eig(KHAT2, 3, d2, v2);

       for (j=1;j<=3;j++) {
            temp[j]=v1[j][1]; v1[j][1]=v1[j][3]; v1[j][3]=temp[j];
       }
    for (j=1;j<=3;j++) {
            temp[j]=v2[j][1]; v2[j][1]=v2[j][3]; v2[j][3]=temp[j];
       }
       len1[1]= 1/sqrt(d1[3]); len1[2]= 1/sqrt(d1[2]); len1[3]= 1/sqrt(d1[1]);
    len2[1]= 1/sqrt(d2[3]); len2[2]= 1/sqrt(d2[2]); len2[3]= 1/sqrt(d2[1]);

       info(A, F1, KHAT1);
       info(B, F2, KHAT2);
       /* This minimization procedure finds the minimum distance from a given
point
       on the surface of an ellipsoid to the surface of the other ellipsoid.
Notice
       that the following procedure is independent of the positions of starting
       points: */
       minimize(state,2,dist,d_dist1,state);
       norm_angle(state,2);
       pos(state,r1,r2);
       /* Minimization of distance by Variable Metric Method w/two variables */
       /* Minimization Phase I: SHIFT1 is provided to perturbate a sequence
             of points computed by minimize() */
       i=1;
       do {
            for (j=1;j<=3;j++) {
              rold1[j]=r1[j];
              rold2[j]=r2[j];
            }
            state[3] += SHIFT1;
            state[4] += SHIFT1;
            minimize(state+2,2,dist,d_dist2,state);
            norm_angle(state+2,2);
            state[1] += SHIFT1;
            state[2] += SHIFT1;
            minimize(state,2,dist,d_dist1,state);
            norm_angle(state,2);
            pos(state,r1,r2);
            delt=max(dt(r1,rold1),dt(r2,rold2));
       } while ((delt > EP) && (i++ <= MAXSTEP));
       /* Minimization Phase II: SHIFT 2 is provided at this step */
       if (delt > EP) {
            step1=1;
            for (; delt > EP && step1 < STEP1MAX ;){
              for (j=1;j<=3;j++) {
                   rold1[j]=r1[j];
                   rold2[j]=r2[j];
              }
              state[3] += SHIFT2;
```

```
                state[4] += SHIFT2;
                minimize(state+2,2,dist,d_dist2,state);
                norm_angle(state+2,2);
                state[1] += SHIFT2;
                state[2] += SHIFT2;
                minimize(state,2,dist,d_dist1,state);
                norm_angle(state,2);
                pos(state,r1,r2);
                delt=max(dt(r1,rold1),dt(r2,rold2));
                step1++;
              }
         }
         if(dt(r1,r2) < min(A[1],min(A[2],min(A[3],min(B[1],min(B[2],B[3])))))){
              /* Minimizations by Variable Metric Method w/ four variables */
              minimize(state,4,dist,d_dist,state);
              norm_angle(state,4);
              pos(state,r1,r2);
              enorm(state,en1,en2);
              /* Check the contact occurs: */
              check=(r2[1]-r1[1])*en1[1]+(r2[2]-r1[2])*en1[2]+(r2[3]-
r1[3])*en1[3];
              pt= dot(en1,en2,3);
              if(check > 0. && pt<= -1+DEV) {
                FREE_MATRIX;
                return FALSE; /* no real contact */
              } else {
                /* Minimization Phase III: EP1 which is smaller than EP to obtain
the
                converged points with better accuracy*/
                step2=1;
                for (;delt > EP1 && step2 < STEP2MAX;) {
                     for (j=1;j<=3;j++) {
                          rold1[j]=r1[j];
                          rold2[j]=r2[j];
                     }
                     state[3] += SHIFT2;
                     state[4] += SHIFT2;
                     minimize(state+2,2,dist,d_dist2,state);
                     norm_angle(state+2,2);
                     state[1] += SHIFT2;
                     state[2] += SHIFT2;
                     minimize(state,2,dist,d_dist1,state);
                     norm_angle(state,2);
                     pos(state,r1,r2);
                     delt=max(dt(r1,rold1),dt(r2,rold2));
                     step2++;
                }
                minimize(state,4,dist,d_dist,state);
                norm_angle(state,4);
                pos(state,r1,r2);
                enorm(state,n1,en2);
                /* Check the contact occurs: */
                check=(r2[1]-r1[1])*n1[1]+(r2[2]-r1[2])*n1[2]+(r2[3]-r1[3])*n1[3];
                pt= DOT3(n1,en2);
                if(check > 0. && pt<= -1+DEV1) {
                     FREE_MATRIX;
                     return FALSE; /* no real contact */
                } else {
                     /* Searching for the unique contact point*/
                     if (pert(state,1.e-7)) { /* pert() returns TRUE or FALSE */
                          for (i=1;i<=4;i++) deep[i]=state[i];
                          pos(deep,r1,r2);
                          enorm(state,n1,en2);  /* for superqudratic model */
                          FREE_MATRIX;
                          return TRUE; /* contact happens */
                     } else {
                          FREE_MATRIX;
                          return FALSE;
                     }
                } /* end perturbation procedure */
              }   /* end Phase III */
         } else {
              FREE_MATRIX;
              return FALSE; /* no real contact */
         }
    } else{ /* end potential contact detect */
         FREE_MATRIX;
         return FALSE; /* no potential contact */
    }
  }

  /******************************************************************************/
  /********                                                               *******/
  /********                        supplements                            *******/
  /********                                                               *******/
  /******************************************************************************/
  /******************************************************************************/
  /*   Compute the matrix KHAT=F^(-T)KF^(-1)                                    */
  /******************************************************************************/
  void info(Vector SA, Matrix F, Matrix KHAT)
  {
    Matrix FINV, FT, K, C;
    FINV=matrix(3,3); FT=matrix(3,3); K=matrix(3,3); C=matrix(3,3);

    matrix_inverse(F, 3, FINV);
    matrix_transpose(FINV,FT,3,3);

    K[1][1]=1/(SA[1]*SA[1]); K[1][2]=0.; K[1][3]=0.;
    K[2][1]=0.; K[2][2]=1/(SA[2]*SA[2]); K[2][3]=0.;
    K[3][1]=0.; K[3][2]=0.; K[3][3]=1/(SA[3]*SA[3]);

    matrix_times_matrix(C,FT,K,3,3,3);
    matrix_times_matrix(KHAT,C,FINV,3,3,3);  /* KHAT=F^(-T)KF^(-1) */

    free_matrix(C);
    free_matrix(FT);
    free_matrix(K);
    free_matrix(FINV);
  }

  /******************************************************************************/
  /*  Compute and sort the eigenvalues and eigenvectors                         */
  /******************************************************************************/
  void eig(Matrix a, int n, Vector d, Matrix v)
```

```
{
  int nrot;

  void jacobi(Matrix a, int n, Vector d, Matrix v, int *nrot);
  void eigsrt(Vector d, Matrix v, int n);

  jacobi(a, n, d, v, &nrot);
  eigsrt(d, v, n);
}

/********************************************************************************/
/*  Compute the principal directions and principal values of the ellipsoids   */
/*  in the current configuration                                                 */
/********************************************************************************/
#define ROTATE(a,i,j,k,l) g=a[i][j]; h=a[k][l]; a[i][j]=g-s*(h+g*tau);
a[k][l]=h+s*(g-h*tau);

void jacobi(Matrix a, int n, Vector d, Matrix v, int *nrot)
/* Compute all eigenvalues and eigenvectors of a real symmetric matrix
      a[1..n][1..n]. On output, the elements of above the diagonal are
destroyed.
      d[1..n] returns the eigenvalues of a. v[1..n][1..n] is a matrix whose
columns
      contain, on output, the normalized eigenvectors of a. nrot returns the
number
      jacobi rotates that were required. See Numerical Recipes on C pp.467-468.
*/
{
  int j, iq, ip, i;
  double tresh, theta, tau, t, sm, s, h, g, c;
  Vector b, z;

  b=vector(n); z=vector(n);

  for(ip=1;ip<=n;ip++){
       for (iq=1;iq<=n;iq++) v[ip][iq]=0.0;
       v[ip][ip]=1.0;
  }
  for (ip=1;ip<=n;ip++){
       b[ip]=d[ip]=a[ip][ip];
       z[ip]=0.0;
  }
  *nrot=0;
  for (i=1;i<=50;i++) {
       sm=0.0;
       for (ip=1;ip<=n-1;ip++) for (iq=ip+1;iq<=n;iq++) sm += fabs(a[ip][iq]);
       if(sm==0.0){
            free_vector(z);
            free_vector(b);
            return;
       }
       if (i<4) tresh=0.2*sm/(n*n);
       else tresh=0.0;
       for (ip=1;ip<=n-1;ip++) {
            for (iq=ip+1;iq<=n;iq++) {
              g=100.*fabs(a[ip][iq]);
              if (i>4 && (double)(fabs(d[ip])+g)==(double)fabs(d[ip])
                        && (double)(fabs(d[iq])+g)==(double)fabs(d[ip]))
                   a[ip][iq]=0.0;
              else if (fabs(a[ip][iq])>tresh) {
                   h=d[iq]-d[ip];
                   if ((double)(fabs(h)+g)==(double)fabs(h)) t=(a[ip][iq])/h;
                   else {
                        theta=0.5*h/(a[ip][iq]);
                        t=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
                        if (theta<0.0) t= -t;
                   }
                   c=1.0/sqrt(1+t*t);
                   s=t*c;
                   tau=s/(1.0+c);
                   h=t*a[ip][iq];
                   z[ip] -= h;
                   z[iq] += h;
                   d[ip] -= h;
                   d[iq] += h;
                   a[ip][iq]=0.0;

                   for (j=1;j<=ip-1;j++) {
                        ROTATE(a,j,ip,j,iq)
                   }
                   for (j=ip+1;j<=iq-1;j++) {
                        ROTATE(a,ip,j,j,iq)
                   }
                   for (j=iq+1;j<=n;j++) {
                        ROTATE(a,ip,j,iq,j)
                   }
                   for (j=1;j<=n;j++) {
                        ROTATE(v,j,ip,j,iq)
                   }
                   ++(*nrot);
              }
            }
       }
       for (ip=1;ip<=n;ip++){
            b[ip] += z[ip];
            d[ip]=b[ip];
            z[ip]=0.0;
       }
  }
  nrerror("Too many iterations in routine jacobi()");
}

/*******************************************************************************/
/*  Sorting the eigenvalues and eigenvectors                                   */
/*******************************************************************************/
void eigsrt(Vector d, Matrix v, int n)
/* Given the eigenvalues d[1..n] and the eigenvector in matrix v[1..n][1..n]
   as the output from Jacobi, this routine sorts the eigenvalues into decending
   order, adn rearranges the eigenvectors correspondingly. See Numerical Recipes
   in C on p.468. */
{
  int k, j, i;
  double p;
  for (i=1;i<n;i++) {
    p=d[k=i];
       for (j=i+1;j<=n;j++) if (d[j] >= p) p=d[k=j];
       if (k != i) {
            d[k]=d[i];
            d[i]=p;
            for (j=1;j<=n;j++) {
              p=v[j][i];
              v[j][i]=v[j][k];
              v[j][k]=p;
            }
       }
  }
}

/*************************************************************************/
/*  Compute the outward unit normal on the surface of the ellipsoids          */
/*************************************************************************/
void enorm(Vector x, Vector nvec1, Vector nvec2)
        /* x[1]=u1, x[2]=v1, x[3]=u2, x[4]=v2 */
{
  int i,j,k1,k2,s[5],c[5],k[5];
  double test1, test2, tang1[4], tang2[4], cp1[4], tang3[4], tang4[4], cp2[4];
  double norm1, norm2, L1[4], L2[3], L3[4],L4[3],temp[5];
  double s_x1,s_x2,c_x1,c_x2,s_x3,s_x4,c_x3,c_x4;
  double s_x1n,c_x1n,s_x2n,c_x2n,s_x3n,c_x3n,s_x4n,c_x4n;
  double s_x2p,c_x1p,c_x2p,s_x4p,c_x3p,c_x4p;

  for (i=1;i<=4;i++) temp[i]=x[i];
  for (i=1;i<=4;i++) {
       k[i]=(int)(x[i]/(2*Pi));
       if (x[i]>=0) x[i]=x[i]-2*k[i]*Pi;
       else x[i]=x[i]-2*(k[i]-1)*Pi;
  }

  for (i=1;i<=4;i++) {
       if (x[i]>=0 && x[i]<=Pi/2) {
            s[i]=1; c[i]=1;
       } else if (x[i]>Pi/2 && x[i]<=Pi) {
            x[i]= Pi-x[i];  s[i]=1; c[i]= -1;
       } else if (x[i]>Pi && x[i]<=3*Pi/2) {
            x[i]=x[i]-Pi; s[i]= -1; c[i]= -1;
       } else {
            x[i]=2*Pi-x[i]; s[i]= -1; c[i]=1;
       }
  }
  if (EXP==1) {
       s_x1n=c_x1n=s_x2n=c_x2n=s_x3n=c_x3n=c_x4n=s_x4n=1.0;
  } else {
       if (x[1]<1.e-16) s_x1n=1.e16;
       else  s_x1n= pow(sin(x[1]),EXP-1);
       if (x[1]> (Pi/2-1.e-16)) c_x1n=1.e16;
       else c_x1n= pow(cos(x[1]),EXP-1);
       if (x[2]<1.e-16) s_x2n=1.e16;
       else s_x2n= pow(sin(x[2]),EXP-1);
       if (x[2]> (Pi/2-1.e-16)) c_x2n=1.e16;
       else c_x2n= pow(cos(x[2]),EXP-1);
```

```
     if (x[3]<1.e-16) s_x3n=1.e16;
     else s_x3n= pow(sin(x[3]),EXP-1);
     if (x[3]> (Pi/2-1.e-16)) c_x3n=1.e16;
     else c_x3n= pow(cos(x[3]),EXP-1);
     if (x[4]<1.e-16) s_x4n=1.e16;
     else s_x4n= pow(sin(x[4]),EXP-1);
     if (x[4]> (Pi/2-1.e-16)) c_x4n=1.e16;
     else c_x4n= pow(cos(x[4]),EXP-1);
}
s_x2p= s[2]*pow(sin(x[2]),EXP);
c_x1p= c[1]*pow(cos(x[1]),EXP);
c_x2p= c[2]*pow(cos(x[2]),EXP);
s_x4p= s[4]*pow(sin(x[4]),EXP);
c_x3p= c[3]*pow(cos(x[3]),EXP);
c_x4p= c[4]*pow(cos(x[4]),EXP);
s_x1= s[1]*sin(x[1]);
s_x2= s[2]*sin(x[2]);
c_x1= c[1]*cos(x[1]);
c_x2= c[2]*cos(x[2]);
s_x3= s[3]*sin(x[3]);
s_x4= s[4]*sin(x[4]);
c_x3= c[3]*cos(x[3]);
c_x4= c[4]*cos(x[4]);

L1[1]= -len1[1]*c_x1n*s_x1*c_x2p;
L1[2]= -len1[2]*c_x1n*s_x1*s_x2p;
L1[3]=  len1[3]*s_x1n*c_x1;
L2[1]= -len1[1]*c_x1p*s_x2*c_x2n;
L2[2]=  len1[2]*c_x1p*c_x2*s_x2n;

L3[1]= -len2[1]*c_x3n*s_x3*c_x4p;
L3[2]= -len2[2]*c_x3n*s_x3*s_x4p;
L3[3]=  len2[3]*s_x3n*c_x3;
L4[1]= -len2[1]*c_x3p*s_x4*c_x4n;
L4[2]=  len2[2]*c_x3p*c_x4*s_x4n;
L2[3]=L4[3]=0.;

for (i=1;i<=3;i++) tang1[i]=tang2[i]=tang3[i]=tang4[i]=0.0;
for (i=1;i<=3;i++) {
     for (j=1;j<=3;j++) {
          tang1[i]  += EXP*L1[j]*v1[i][j];
          tang2[i]  += EXP*L2[j]*v1[i][j];
          tang3[i]  += EXP*L3[j]*v2[i][j];
          tang4[i]  += EXP*L4[j]*v2[i][j];
     }
}
/* Cross product of the two tangent vectors: */
cp1[1]=tang2[2]*tang1[3]-tang1[2]*tang2[3];
cp1[2]=tang2[3]*tang1[1]-tang2[1]*tang1[3];
cp1[3]=tang2[1]*tang1[2]-tang2[2]*tang1[1];
cp2[1]=tang4[2]*tang3[3]-tang3[2]*tang4[3];
cp2[2]=tang4[3]*tang3[1]-tang4[1]*tang3[3];
cp2[3]=tang4[1]*tang3[2]-tang4[2]*tang3[1];

/* Compute unit normal vectors */
norm1=sqrt(square(cp1[1])+square(cp1[2])+square(cp1[3]));
norm2=sqrt(square(cp2[1])+square(cp2[2])+square(cp2[3]));
```

```
  k1=(int)(temp[1]/(2*Pi));
  test1=fabs(temp[1]-2*k1*Pi);

  k2=(int)(temp[3]/(2*Pi));
  test2=fabs(temp[3]-2*k2*Pi);

  if (test1<=Pi/2 || test1>=3*Pi/2) for (i=1;i<=3;i++) nvec1[i]=cp1[i]/norm1;
  else for (i=1;i<=3;i++) nvec1[i]= -cp1[i]/norm1;

  if (test2<=Pi/2 || test2>=3*Pi/2) for (i=1;i<=3;i++) nvec2[i]= cp2[i]/norm2;
  else for (i=1;i<=3;i++) nvec2[i]= -cp2[i]/norm2;

  for (i=1;i<=4;i++) x[i]=temp[i];
}

void angle(Vector x, int s[], int c[])
{
  int i, k[5];

  for (i=1;i<=4;i++) {
       k[i]=(int)(x[i]/(2*Pi));
       if (x[i]>= 0) x[i]=x[i]-2*k[i]*Pi;
       else x[i]=x[i]-2*(k[i]-1)*Pi;
       if (x[i]>=0 && x[i]<=Pi/2) {
            s[i]=1; c[i]=1;
       } else if (x[i]>Pi/2 && x[i]<=Pi) {
            x[i]= Pi-x[i];  s[i]=1; c[i]= -1;
       } else if (x[i]>Pi && x[i]<=3*Pi/2) {
            x[i]=x[i]-Pi; s[i]= -1; c[i]= -1;
       } else {
            x[i]=2*Pi-x[i]; s[i]= -1; c[i]=1;
       }
  }
}
/*******************************************************************************/
/*   Angle normalization                                                       */
/*******************************************************************************/
void norm_angle(Vector x, int n)
        /* normalize the state vector to [0, 2*Pi] */
{
  int i, k[5];

  for (i=1;i<=n;i++) {
       k[i]=(int)(x[i]/(2*Pi));
       if (x[i]>= 0.0) x[i]=x[i]-2*k[i]*Pi;
       else x[i]=x[i]-2*(k[i]-1)*Pi;
  }
}
/*******************************************************************************/
/*   Compute the  position vectors of the contact points                       */
/*******************************************************************************/
void pos(Vector x, Vector r1, Vector r2)
        /* x={u1,v1,u2,v2}, ri is the position vector on the surface of body i
*/
{
  int i,j;
  int s[5],c[5];
  double L1[4],L2[4],temp[5];
  double s_x1, c_x12, c_x34, s_x3, cs_x12, cs_x34, c_x1, c_x3;
  void angle(Vector, int s[], int c[]);

  for (i=1;i<=4;i++) temp[i]=x[i];
  angle(x,s,c);

  s_x1= s[1]*pow(sin(x[1]),EXP);
  s_x3= s[3]*pow(sin(x[3]),EXP);
  c_x1= c[1]*pow(cos(x[1]),EXP);
  c_x3= c[3]*pow(cos(x[3]),EXP);
  c_x12= c_x1*c[2]*pow(cos(x[2]),EXP);
  c_x34= c_x3*c[4]*pow(cos(x[4]),EXP);
  cs_x12= c_x1*s[2]*pow(sin(x[2]),EXP);
  cs_x34= c_x3*s[4]*pow(sin(x[4]),EXP);

  L1[1]=len1[1]*c_x12; L1[2]=len1[2]*cs_x12; L1[3]=len1[3]*s_x1;
  L2[1]=len2[1]*c_x34; L2[2]=len2[2]*cs_x34; L2[3]=len2[3]*s_x3;

  for (i=1;i<=3;i++) r1[i]=r2[i]=0.0;

  for (i=1;i<=3;i++) {
       for (j=1;j<=3;j++) {
            r1[i] += L1[j]*v1[i][j];
            r2[i] += L2[j]*v2[i][j];
       }
       r2[i] += xbar[i];
  }
  for (i=1;i<=4;i++) x[i]=temp[i];
}

/*****************************************************************************/
/*  Calculate the distance S(u1,v1,u2,v2) at x[1..4]:                       */
/*  x[1]=u1, x[2]=v1, x[3]=u2, x[4]=v2                                       */
/*****************************************************************************/
double dist(Vector x)
{
  int i, s[5],c[5];
  double dummy=0.0, st[5], ct[5], t[5];   /* substitution variables */
  void angle(Vector, int s[], int c[]);

  for (i=1;i<=4;i++) t[i]=x[i];
  angle(t,s,c);           /* note: angle() modifies t! */
  for (i=1;i<=4;i++) {
       ct[i]=c[i]*pow(cos(t[i]),EXP);
       st[i]=s[i]*pow(sin(t[i]),EXP);
  }

  for (i=1;i<=3;i++){
       dummy += square(-xbar[i]+v1[i][1]*ct[1]*ct[2]*len1[1]-
                   v2[i][1]*ct[3]*ct[4]*len2[1]+v1[i][3]*len1[3]*st[1]+
                   v1[i][2]*ct[1]*len1[2]*st[2]-v2[i][3]*len2[3]*st[3]-
                   v2[i][2]*ct[3]*len2[2]*st[4]);
  }
  return sqrt(dummy);
}
```

```
/****************************************************************************/
/*  Calculate the gradient g=(dS/du1,dS/dv1) at the point p[1,2]=(u1,v1)      */
/*  given q[1,2]=(u2,v2).                                                     */
/****************************************************************************/
void d_dist1(Vector g, Vector x)
{
  int i, s[5], c[5];
  double st[5], ct[5], t[5], sine[5], cose[5];
  double cn1, cn2, sn1, sn2;
  double m, ds=0.0, sqds, dummy1=0.0, dummy2=0.0;
  void angle(Vector, int s[],  int c[]);

  for (i=1;i<=4;i++) t[i]=x[i];
  angle(t,s,c);            /* note: angle() modifies t! */
  for (i=1;i<=4;i++) {
       sine[i]=sin(t[i]);
       cose[i]=cos(t[i]);
       ct[i]=c[i]*pow(cose[i],EXP);
       st[i]=s[i]*pow(sine[i],EXP);
  }
  if (EXP==1) {
       cn1=cn2=sn1=sn2=1.0;
  } else {
       if (t[1]> (Pi/2-1.e-14)) cn1=1.e14;
       else cn1=pow(cose[1],EXP-1);
       if (t[2]> (Pi/2-1.e-14)) cn2=1.e14;
       else cn2=pow(cose[2],EXP-1);
       if (t[1]<1.e-14) sn1=1.e14;
       else sn1=pow(sine[1],EXP-1);
       if (t[2]<1.e-14) sn2=1.e14;
       else sn2=pow(sine[2],EXP-1);
  }

  for (i=1;i<=3;i++) {
       m= -xbar[i]+v1[i][1]*ct[1]*ct[2]*len1[1]-
            v2[i][1]*ct[3]*ct[4]*len2[1]+v1[i][3]*len1[3]*st[1]+
            v1[i][2]*ct[1]*len1[2]*st[2]-v2[i][3]*len2[3]*st[3]-
            v2[i][2]*ct[3]*len2[2]*st[4];

       ds += square(m);
       dummy1+= (-v1[i][1]*cn1*len1[1]*s[1]*sine[1]*ct[2]-
                            v1[i][2]*cn1*len1[2]*s[1]*sine[1]*st[2]+
                            v1[i][3]*sn1*len1[3]*c[1]*cose[1])*EXP*m;

       dummy2+= (-v1[i][1]*ct[1]*cn2*len1[1]*s[2]*sine[2]+
                    v1[i][2]*ct[1]*sn2*len1[2]*c[2]*cose[2])*EXP*m;
  }
  /* dS/su1=(1/2S)*dS^2/du1, since there is a 2 in dummy, so dummy is divided by
S only. */
  sqds=sqrt(ds);
  g[1]= dummy1/sqds;
  g[2]= dummy2/sqds;
}

/*************************************************************************/
/*  Calculate the gradient g=(dS/du2, dS/dv2) at the point p[1,2]=(u2,v2)  */
/*  given q[1,2]=(u1,v1).                                                  */
/**************************************************************************/
void d_dist2(Vector g, Vector x)
{
  int i, s[5], c[5];
  double st[5], ct[5], t[5], sine[5], cose[5];
  double cn3, cn4, sn3, sn4;
  double m, ds=0.0, sqds, dummy3=0.0, dummy4=0.0;
  void angle(Vector, int s[],  int c[]);

  for (i=1;i<=4;i++) t[i]=x[i];
  angle(t,s,c);           /* note: angle() modifies t! */
  for (i=1;i<=4;i++) {
       sine[i]=sin(t[i]);
       cose[i]=cos(t[i]);
       ct[i]=c[i]*pow(cose[i],EXP);
       st[i]=s[i]*pow(sine[i],EXP);
  }
  if (EXP==1) {
       cn3=cn4=sn3=sn4=1.0;
  } else {
       if (t[3]> (Pi/2-1.e-14)) cn3=1.e14;
       else cn3=pow(cose[3],EXP-1);
       if (t[4]> (Pi/2-1.e-14)) cn4=1.e14;
       else cn4=pow(cose[4],EXP-1);
       if (t[3]<1.e-14) sn3=1.e14;
       else sn3=pow(sine[3],EXP-1);
       if (t[4]<1.e-14) sn4=1.e14;
       else sn4=pow(sine[4],EXP-1);
  }
  for (i=1;i<=3;i++) {
       m= -xbar[i]+v1[i][1]*ct[1]*ct[2]*len1[1]-
            v2[i][1]*ct[3]*ct[4]*len2[1]+v1[i][3]*len1[3]*st[1]+
            v1[i][2]*ct[1]*len1[2]*st[2]-v2[i][3]*len2[3]*st[3]-
            v2[i][2]*ct[3]*len2[2]*st[4];

       ds += square(m);
       dummy3 += (v2[i][1]*cn3*len2[1]*s[3]*sine[3]*ct[4]+
                          v2[i][2]*cn3*len2[2]*s[3]*sine[3]*st[4]-
                          v2[i][3]*sn3*len2[3]*c[3]*cose[3])*EXP*m;

       dummy4 += (v2[i][1]*ct[3]*cn4*len2[1]*s[4]*sine[4]-
                   v2[i][2]*ct[3]*sn4*len2[2]*c[4]*cose[4])*EXP*m;
  }
  sqds=sqrt(ds);
  g[1]= dummy3/sqds;
  g[2]= dummy4/sqds;
}

/**************************************************************************/
/*  Calculate the gradient g=(dS/du1, dS/dv1, dS/du2, dS/dv2) at          */
/*  x[1..4]=(u1,v1,u2,v2)                                                  */
/**************************************************************************/
void d_dist(Vector g, Vector x)
{
  int i, s[5], c[5];
  double st[5], ct[5], t[5], sine[5], cose[5];
  double sn[5], cn[5];
  double m, ds=0.0, sqds, dummy[5]={0.0, 0.0, 0.0, 0.0, 0.0};
  void angle(Vector, int s[],  int c[]);

  for (i=1;i<=4;i++) t[i]=x[i];
  angle(t,s,c);            /* note: angle() modifies t! */
  for (i=1;i<=4;i++) {
       sine[i]=sin(t[i]);
       cose[i]=cos(t[i]);
       ct[i]=c[i]*pow(cose[i],EXP);
       st[i]=s[i]*pow(sine[i],EXP);
  }
  if (EXP==1) {
       for (i=1;i<=4;i++) cn[i]=sn[i]=1.0;
  }else {
       if (t[1]> (Pi/2-1.e-14)) cn[1]=1.e14;
       else cn[1]=pow(cos(t[1]),EXP-1);
       if (t[2]> (Pi/2-1.e-14)) cn[2]=1.e14;
       else cn[2]=pow(cos(t[2]),EXP-1);
       if (t[1]<1.e-14) sn[1]=1.e14;
       else sn[1]=pow(sin(t[1]),EXP-1);
       if (t[2]<1.e-14) sn[2]=1.e14;
       else sn[2]=pow(sin(t[2]),EXP-1);
       if (t[3]> (Pi/2-1.e-14)) cn[3]=1.e14;
       else cn[3]=pow(cos(t[3]),EXP-1);
       if (t[4]> (Pi/2-1.e-14)) cn[4]=1.e14;
       else cn[4]=pow(cos(t[4]),EXP-1);
       if (t[3]<1.e-14) sn[3]=1.e14;
       else sn[3]=pow(sin(t[3]),EXP-1);
       if (t[4]<1.e-14) sn[4]=1.e14;
       else sn[4]=pow(sin(t[4]),EXP-1);
  }
  for (i=1;i<=3;i++) {
       m= -xbar[i]+v1[i][1]*ct[1]*ct[2]*len1[1]-
            v2[i][1]*ct[3]*ct[4]*len2[1]+v1[i][3]*len1[3]*st[1]+
            v1[i][2]*ct[1]*len1[2]*st[2]-v2[i][3]*len2[3]*st[3]-
            v2[i][2]*ct[3]*len2[2]*st[4];

       ds += square(m);
       dummy[1] += (-v1[i][1]*cn[1]*len1[1]*s[1]*sine[1]*ct[2]-
                                 v1[i][2]*cn[1]*len1[2]*s[1]*sine[1]*st[2]+
                                 v1[i][3]*sn[1]*len1[3]*c[1]*cose[1])*EXP*m;

       dummy[2] += (-v1[i][1]*ct[1]*cn[2]*len1[1]*s[2]*sine[2]+
                          v1[i][2]*ct[1]*sn[2]*len1[2]*c[2]*cose[2])*EXP*m;

       dummy[3] += (v2[i][1]*cn[3]*len2[1]*s[3]*sine[3]*ct[4]+
                              v2[i][2]*cn[3]*len2[2]*s[3]*sine[3]*st[4]-
                              v2[i][3]*sn[3]*len2[3]*c[3]*cose[3])*EXP*m;

       dummy[4] += (v2[i][1]*ct[3]*cn[4]*len2[1]*s[4]*sine[4]-
                         v2[i][2]*ct[3]*sn[4]*len2[2]*c[4]*cose[4])*EXP*m;
  }
  sqds=sqrt(ds);
  for (i=1;i<=4;i++) g[i]= dummy[i]/sqds;
}
```

```
/*****************************************************************************/
/********                                                            *******/
/********                Unique Contact Point Detection              *******/
/********                                                            *******/
/*****************************************************************************/
/*****************************************************************************/
/* Input a[1..4] is the vector from detect_contact and a[1..4] is also the    */
/* output vector as which corresponds to unique contact points on both bodies.*/
/*****************************************************************************/
#define EPS1 1.e-15
#define EPS2 1.e-15
#define NT 32
#define FAC1 0.7
#define FAC2 2
#define MAX 300

int pert(Vector a, double SHIFT)
{
  int i,j,lp1,lp2;
  double func(Vector, Vector, Vector, Matrix);
  void pos(Vector state, Vector r1, Vector r2);
  void piksrt(int, int , Vector a, Matrix atri);
  void opp(Vector a, Vector b);
  void search(Vector, Vector, double, double, double);

  double diff1, diff2, fc1, fc2, va1, va2;
  double ftemp1, ftemp2;
  double r1[4], r2[4], b[5], c[5], fa[NT+1], fb[NT+1];

  Matrix atri=matrix(NT,4);
  Matrix btri=matrix(5,4);

  fb[5]=1.0; fa[5]=1.0; va1= 1.0; va2= 1.0;
  for (;fb[5]>0.0 || fa[5]>0.0 ;) {
       for (i=1;i<=NT;i++) {        /* perturbation along NT directions */
            atri[i][1]=a[1]+cos((i-1)*2*Pi/NT)*SHIFT; /* SHIFT : radian */
            atri[i][2]=a[2]+sin((i-1)*2*Pi/NT)*SHIFT;
            atri[i][3]=a[3]+cos((i-1)*2*Pi/NT)*SHIFT;
            atri[i][4]=a[4]+sin((i-1)*2*Pi/NT)*SHIFT;
            for (j=1;j<=4;j++) b[j]=atri[i][j];
            pos(b,r1,r2); fb[i]=func(r1,xbar,len2,v2);
fa[i]=func(r2,null,len1,v1);
       }
       /* Sorting (atri[i][1],atri[i][2]) which are on the body 1 by index
fb[1..n]*/
       /* Sorting (atri[i][3],atri[i][4]) which are on the body 2 by index
fa[1..n]*/
       piksrt(1,NT,fb,atri);
       piksrt(3,NT,fa,atri);

       if (fb[5]>0.0 || fa[5]>0.0 ) {
            if (fa[1] > va1 || fb[1] > va2) SHIFT *= FAC1;
            else SHIFT *= FAC2;
            if (SHIFT > 0.1) {
              free_matrix(atri); free_matrix(btri);
              return FALSE;
            }
            if (SHIFT < 1.e-9) {
              free_matrix(atri); free_matrix(btri);
              return FALSE;
            }
            va1=fa[1]; va2=fb[1];
        }
    }
    /* perturbating along the five directions with smallest function values on
          body 1 & 2 */
    for (j=1;j<=5;j++){
        for (i=1;i<=4;i++) b[i]=atri[j][i];
        opp(a,b);
        for (i=1;i<=4;i++) btri[j][i]=b[i];
        /* btri[][] are the oppsite points to a[], the start points. */
    }
    free_matrix(atri);

    /* Take five middle points on body 1  & 2 */
    for (j=1;j<=5;j++){
        for (i=1;i<=4;i++) btri[j][i]=(a[i]+btri[j][i])/2;
        /* btri[][] are the middle points now. */
        for (i=1;i<=4;i++) b[i]=btri[j][i];
        pos(b,r1,r2); fb[j]=func(r1,xbar,len2,v2); fa[j]=func(r2,null,len1,v1);
    }
    piksrt(1,5,fb,btri);
    piksrt(3,5,fa,btri);

    /* Using the smallest three middle points to perturbate along a_c! */
    for (j=1;j<=3;j++){
        for (i=1;i<=4;i++) c[i]= btri[j][i];
        fc1= fb[j]; fc2= fa[j];
        search(a,c,fc1,fc2,EPS2);
        for (i=1;i<=4;i++) btri[j][i]= c[i];
        /* btri[][] are the points with smallest function value along curves
  formed
             by start point and middle points. */
    }
    /* Perturbate along the previous three points */
    for (i=1;i<=4;i++) b[i]=btri[1][i]+0.01*(btri[2][i]-btri[1][i]);
    pos(b,r1,r2); ftemp1=func(r1,xbar,len2,v2); ftemp2=func(r2,null,len1,v1);
    if (ftemp1 < fb[1]) for (i=1;i<=2;i++) c[i]=(btri[1][i]+btri[2][i])/2;
    else for (i=1;i<=2;i++) c[i]=(btri[1][i]+btri[3][i])/2;
    if (ftemp2 < fa[1]) for (i=3;i<=4;i++) c[i]=(btri[1][i]+btri[2][i])/2;
    else for (i=3;i<=4;i++) c[i]=(btri[1][i]+btri[3][i])/2;
    pos(c,r1,r2); fc1=func(r1,xbar,len2,v2); fc2=func(r2,null,len1,v1);
    diff1=fabs(fabs(fb[1])-fabs(fc1));

    for (i=1;i<=4;i++) b[i]=btri[1][i];
    /* Loop to search for the direction corresponding to the min function value */
    /* points on body 1 */
    lp1=1;
    for (;diff1 > EPS1 && lp1 < MAX;){
        if (fb[1]<fc1) {
            /* update vector c */
            for (i=1;i<=2;i++) c[i]=(c[i]+b[i])/2;
            pos(c,r1,r2); fc1=func(r1,xbar,len2,v2);
        } else {
            /* update vector b */
            for (i=1;i<=2;i++) b[i]=(c[i]+b[i])/2;
            pos(b,r1,r2); fb[1]=func(r1,xbar,len2,v2);
       }
       diff1=fabs(fabs(fb[1])-fabs(fc1));
       lp1++;
  }
  /* points on body 2 */
  diff2=fabs(fabs(fa[1])-fabs(fc2));
  lp2=1;
  for (;diff2 > EPS1 && lp2 < MAX;) {
       if (fa[1]<fc2) {
            /* update vector c */
            for (i=3;i<=4;i++) c[i]=(c[i]+b[i])/2;
            pos(c,r1,r2); fc2=func(r2,null,len1,v1);
       } else {
            /* update vector b */
            for (i=3;i<=4;i++) b[i]=(c[i]+b[i])/2;
            pos(b,r1,r2); fa[1]=func(r2,null,len1,v1);
       }
       diff2=fabs(fabs(fa[1])-fabs(fc2));
       lp2++;
  }
  /* Final perturbation along the a_c direction */
  search(a,c,fc1,fc2,EPS2);
  free_matrix(btri);
  return TRUE;
}

/*****************************************************************************/
/*   Find the corresponding function value of the point x in the current      */
/*   configuration.                                                           */
/*****************************************************************************/
/* x-xbar=F(X-XBAR) Inputs are x:current position,                            */
/* z=xbar: current mass center, and F: deormation gradient.                   */
/* Ellipsoidal function(reference): (X-Xbar).K(X-Xbar)=1.                    */
/* Ellipsoidal function(current): (x-xbar).F^(-T)KF^(-1)(x-xbar)=1.          */
/* K=diag(1/A^2,1/B^2,1/C^2).                                                 */
/*****************************************************************************/
double func(Vector x, Vector z, Vector L, Matrix v)
{
  int i,j;
  double y[4], proj[4], p[4], dummy, inx;
  double dot(Vector, Vector, int);

  for (i=1;i<=3;i++) {
       for (j=1;j<=3;j++) {
            p[j]=v[j][i];
            y[j]= x[j]-z[j];
       }
       proj[i]= fabs(dot(y,p,3));
  }
  dummy=0.0;
  inx= 2./EXP;

  for (i=1;i<=3;i++) {
       dummy+= pow((proj[i]/L[i]),inx);
  }
  return dummy-1;
}
/******************************************************************************/
/*   Sorting                                                                  */
/******************************************************************************/
void piksrt(int k, int n, double arr[], Matrix brr)
/*Sorting an array[1..n] into ascending numerical order, by
straight insertion, while making the corresponding rearrangement
of the Matrix brr.*/
{
  int i,j;
  double a,b,c;

  for (j=2;j<=n;j++) {
       a=arr[j];
       b=brr[j][k];
       c=brr[j][k+1];
       i=j-1;
       while (i>0 && arr[i] > a){
            arr[i+1]=arr[i];
            brr[i+1][k]=brr[i][k];
            brr[i+1][k+1]=brr[i][k+1];
            i--;
       }
       arr[i+1]=a;
       brr[i+1][k]=b;
       brr[i+1][k+1]=c;
  }
}
/******************************************************************************/
/*   Find the oppsite points on the intersection curve                        */
/******************************************************************************/
/* Input:(a[1],a[2]): original point on body 1 */
/*      (a[3],a[4]): original point on body 2 */
/* Input:(b[1],b[2]): perturbation point on body 1 */
/*      (b[3],b[4]): perturbation point on body 2 */
/* output:(b[1],b[2]): the opposite point on body 1 */
/*       (b[3],b[4]): the opposite point on body 2 */
#define FVMIN 1.e-15
#define ITERMAX 200

void opp(Vector a, Vector b)
{
  double func(Vector , Vector ,Vector , Matrix);
  void pos(Vector state, Vector r1, Vector r2);

  int i, ite1,ite2;
  double sht[5], r1[4], r2[4], shtmin[3], ENLARGE[3], fv1, fv2;

  for (i=1;i<=4;i++) sht[i]=b[i]-a[i];
  shtmin[1]=min(sht[1],sht[2]);
  shtmin[2]=min(sht[3],sht[4]);

  for (i=1;i<=2;i++) {
       if (fabs(shtmin[i])<1.e-7) ENLARGE[i]=1.e7;
       else ENLARGE[i]=fabs(1/(shtmin[i]));
  }
  for (i=1;i<=2;i++) b[i]=a[i]+ENLARGE[1]*sht[i];
  for (i=3;i<=4;i++) b[i]=a[i]+ENLARGE[2]*sht[i];
  pos(b,r1,r2);  fv1=func(r1,xbar,len2,v2); fv2=func(r2,null,len1,v1);
  for (;fv1<0. ;) {
       ENLARGE[1] *= 2;
       for (i=1;i<=2;i++) b[i]=a[i]+ENLARGE[1]*sht[i];
       pos(b,r1,r2);  fv1=func(r1,xbar,len2,v2);
  }
  for (;fv2<0. ;) {
       ENLARGE[2] *= 2;
       for (i=3;i<=4;i++) b[i]=a[i]+ENLARGE[2]*sht[i];
       pos(b,r1,r2); fv2=func(r2,null,len1,v1);
  }
  ite1=1;
  for (;(fabs(fv1))> FVMIN && ite1 < ITERMAX;){
            ENLARGE[1]=ENLARGE[1]/2;
       if (fv1>0){
            for (i=1;i<=2;i++) b[i] -= ENLARGE[1]*sht[i];
       } else {
            for (i=1;i<=2;i++) b[i] += ENLARGE[1]*sht[i];
       }
       pos(b,r1,r2);  fv1=func(r1,xbar,len2,v2);
       ite1++;
  }
  ite2=1;
  for (;(fabs(fv2))> FVMIN && ite2 < ITERMAX;){
            ENLARGE[2]=ENLARGE[2]/2;
       if (fv2>0){
            for (i=3;i<=4;i++) b[i] -= ENLARGE[2]*sht[i];
       } else {
            for (i=3;i<=4;i++) b[i] += ENLARGE[2]*sht[i];
       }
       pos(b,r1,r2);  fv2=func(r2,null,len1,v1);
       ite2++;
  }
}

/**************************************************************************/
/*  Searching along the current and starting points located on the surface of */
/*  the ellipsoid                                                         */
/**************************************************************************/
void search(Vector a, Vector c, double fc1, double fc2, double g)
{
  int i,lp1,lp2;
  double r1[4], r2[4], anew[5], fanew1, fanew2, fbnew1, fbnew2, factor;
  double func(Vector , Vector , Vector, Matrix);
  void pos(Vector state, Vector r1, Vector r2);

  factor= 0.1;
  for (i=1;i<=4;i++) anew[i]=c[i]+1.e-12*(c[i]-a[i]);
  pos(anew,r1,r2); fanew1=func(r1,xbar,len2,v2); fanew2=func(r2,null,len1,v1);
  for (i=1;i<=4;i++) anew[i]=c[i]-1.e-12*(c[i]-a[i]);
  pos(anew,r1,r2); fbnew1=func(r1,xbar,len2,v2); fbnew2=func(r2,null,len1,v1);
  if (fc1<fanew1 && fc1<fbnew1){
       a[1]=c[1]; a[2]=c[2]; /* The final point on body 1 */
  } else if (fanew1<fc1){
       for (i=1;i<=2;i++) anew[i]=c[i]+factor*(c[i]-a[i]);
       pos(anew,r1,r2); fanew1=func(r1,xbar,len2,v2);
       lp1=1;
       for (;fabs(fanew1-fc1)> g && lp1 < MAX;){
            if (fanew1<fc1){
              c[1]=anew[1]; c[2]=anew[2]; fc1=fanew1;
            } else factor=factor/2;
            for (i=1;i<=2;i++) anew[i]=c[i]+factor*(c[i]-a[i]);
            pos(anew,r1,r2); fanew1=func(r1,xbar,len2,v2);
            lp1++;
       } /* end of for loop */
       a[1]=c[1]; a[2]=c[2]; /* The final point on body 1 */
  } else {
       for (i=1;i<=2;i++) anew[i]=c[i]-factor*(c[i]-a[i]);
       pos(anew,r1,r2); fanew1=func(r1,xbar,len2,v2);
       lp1=1;
       for (;fabs(fanew1-fc1)> g && lp1 < MAX;){
            if (fanew1<fc1){
              c[1]=anew[1]; c[2]=anew[2]; fc1=fanew1;
            } else factor=factor/2;
            for (i=1;i<=2;i++) anew[i]=c[i]-factor*(c[i]-a[i]);
            pos(anew,r1,r2); fanew1=func(r1,xbar,len2,v2);
            lp1++;
       } /* end of for loop */
       a[1]=c[1]; a[2]=c[2]; /* The final point on body 1 */
  }
  factor= 0.1;
  if (fc2<fanew2 && fc2<fbnew2){
       a[3]=c[3]; a[4]=c[4]; /* The final point on body 2 */
  } else if (fanew2<fc2) {
       for (i=3;i<=4;i++) anew[i]=c[i]+factor*(c[i]-a[i]);
       pos(anew,r1,r2); fanew2=func(r2,null,len1,v1);
       lp2=1;
       for (;fabs(fanew2-fc2)> g && lp2 < MAX;){
            if (fanew2<fc2) {
              c[3]=anew[3]; c[4]=anew[4]; fc2=fanew2;
            } else factor=factor/2;
            for (i=3;i<=4;i++) anew[i]=c[i]+factor*(c[i]-a[i]);
            pos(anew,r1,r2); fanew2=func(r2,null,len1,v1);
            lp2++;
       } /* end of for loop */
       a[3]=c[3]; a[4]=c[4]; /* The final point on body 2 */
  } else {
       for (i=3;i<=4;i++) anew[i]=c[i]-factor*(c[i]-a[i]);
       pos(anew,r1,r2); fanew2=func(r2,null,len1,v1);
       lp2=1;
       for (;fabs(fanew2-fc2)> g && lp2 < MAX;) {
            if (fanew2<fc2) {
              c[3]=anew[3]; c[4]=anew[4]; fc2=fanew2;
            } else factor=factor/2;
            for (i=3;i<=4;i++) anew[i]=c[i]-factor*(c[i]-a[i]);
            pos(anew,r1,r2); fanew2=func(r2,null,len1,v1);
            lp2++;
       } /* end of for loop */
       a[3]=c[3]; a[4]=c[4]; /* The final point on body 2 */
  }
}
```

```
/*****************************************************************************/
/******                                                                 ******/
/******                         common.c                                ******/
/******                                                                 ******/
/******                                                                 ******/
/*****************************************************************************/
/* created by Peter Varadi and Gwo-Jeng Lo, modified June 30, 1998, 479 lines */
/*****************************************************************************/
/* common.c contains some programing tools and vector and matrix related      */
/* material. It is adapted from:                                              */
/*    W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery,      */
/*    Numerical Recipes in C: the Art of Scientific Computing. 2nd ed.,       */
/*    Cambridge University Press, 1992.                                       */
/*****************************************************************************/
/* Functions:                                                                 */
/*   - max()                  line 49                                         */
/*   - min()                  line 50                                         */
/*   - square()               line 51                                         */
/*   - dt()                   line 52                                         */
/*   - nrerror()              line 55                                         */
/*   - vector()               line 71                                         */
/*   - ivector()              line 80                                         */
/*   - matrix                 line 89                                         */
/*   - free_vector()          line 108                                        */
/*   - free_ivector()         line 110                                        */
/*   - free_matrix()          line 112                                        */
/*   - dot()                  line 124                                        */
/*   - vector_product()       line 134                                        */
/*   - matrix_times_vector()  line 142                                        */
/*   - matrix_times_matrix()  line 156                                        */
/*   - matrix_transpose()     line 170                                        */
/*   - matrix_inverse()       line 177                                        */
/*   - lin_solve()            line 202                                        */
/*   - ludcmp()               line 222                                        */
/*   - lubksb()               line 281                                        */
/*   - minimize()             line 330                                        */
/*   - linsearch()            line 412                                        */
/*   - projection()           line 467                                        */
/*****************************************************************************/
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include "common.h"

void ludcmp(Matrix a, int n, int *indx, Vector d);
void lubksb(Matrix a, int n, int *indx, double b[]);

double max(double x, double y) { return (x>y) ? x : y; }
double min(double x, double y) { return (x<y) ? x : y; }
double square(double x) { return x*x; }
double dt(double x[], double y[])
{ return sqrt(square(x[1]-y[1])+square(x[2]-y[2])+square(x[3]-y[3])); }

void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
      fprintf(stderr,"Run-time error...\n");
      fprintf(stderr,"%s\n",error_text);
      fprintf(stderr,"...now exiting to system\n");
      exit(EXIT_FAILURE);
}

/******************************************************************************/
/******                                                                  ******/
/******                    memory management                             ******/
/******                                                                  ******/
/******************************************************************************/
#define FREE_ARG char*

Vector vector(int n)
/* allocate a double vector with subscript range v[1..n] */
{
      Vector v;
      v=(double *) malloc((size_t) (n*sizeof(double)));
      if (!v) nrerror("allocation failure in vector()");
      return v-1;
}

int *ivector(int n)
/* allocate an int vector with range v[1..n] */
{
      int *v;
      v=(int *) malloc((size_t) (n*sizeof(int)));
      if (!v) nrerror("allocation failure in ivector()");
      return v-1;
}

Matrix matrix(int nrow, int ncol)
/* allocate a double matrix with range m[1..nrow][1..ncol] */
{
      int i;
      Matrix m;

      /* allocate pointers to rows */
      m=(double **) malloc((size_t)(nrow*sizeof(double*)));
      if (!m) nrerror("allocation failure 1 in matrix()");
      m-=1;
      /* allocate rows and set pointers to them */
      m[1]=(double *) malloc((size_t)((nrow*ncol)*sizeof(double)));
      if (!m[1]) nrerror("allocation failure 2 in matrix()");
      m[1]-=1;
      for(i=2;i<=nrow;i++) m[i]=m[i-1]+ncol;
      /* return pointer to array of pointers to rows */
      return m;
}

void free_vector(Vector v) { free((FREE_ARG) (v+1)); }
/* free a double vector allocated with vector() */
void free_ivector(int *v) { free((FREE_ARG) (v+1)); }
/* free an int vector allocated with ivector() */
void free_matrix(Matrix m)
/* free a double matrix allocated with matrix() */
{
      free((FREE_ARG) (m[1]+1));
      free((FREE_ARG) (m+1));
}

/*******************************************************************************/
/******                                                                 ******/
/******                      matrix operations                          ******/
/******                                                                 ******/
/*******************************************************************************/
double dot(Vector x, Vector y, int n)
/* calculates scalar product <x,v> of two n vectors */
{
  int i;
  double dummy=0.0;

  for (i=1;i<=n;i++) dummy += x[i]*y[i];
  return dummy;
}

void vector_product(Vector a, Vector b, Vector c)
/* calculates a=b x c */
{
  a[1]=b[2]*c[3]-c[2]*b[3];
  a[2]=b[3]*c[1]-c[3]*b[1];
  a[3]=b[1]*c[2]-c[1]*b[2];
}

void matrix_times_vector(Vector y, Matrix A, Vector x, int nrow, int ncol)
/* calculates y=Ax. A and x are input, y is output. A is a nrow by ncol matrix.
      x is a ncol vector, y is a nrow vector. */
{
      long double dummy; /* This reduces numerical error. */
      int i,j;

      for (i=1;i<=nrow;i++) {
            dummy=0.0;
            for (j=1;j<=ncol;j++) dummy+=A[i][j]*x[j];
            y[i]=dummy;
      }
}

void matrix_times_matrix(Matrix Y, Matrix A, Matrix B, int M, int N, int P)
/* calculates Y=AB. A is a M by N matrix, B is a N by P matrix and Y is a M by
      P matrix. */
{
  int m,n,p;
  long double dummy;

  for (m=1;m<=M;m++) for (p=1;p<=P;p++) {
        dummy=0.0;
        for (n=1;n<=N;n++) dummy += A[m][n]*B[n][p];
        Y[m][p]=dummy;
  }
}

void matrix_transpose(Matrix Y, Matrix A, int nrow, int ncol)
/* A=Y^T , where Y is a nrow*ncol matrix, A is a ncol*nrow matrix */
```

```
{
  int i,j;
  for (i=1;i<=ncol;i++) for (j=1;j<=nrow;j++) A[i][j]=Y[j][i];
}

void matrix_inverse(Matrix a, int n, Matrix y)
/* Calculates the inverse y of a square matrix a. n is the size of the matrix.
      This function is explained in the Numerical Recipes in C on page 48. */
{
      double d;
      int i, j, *indx;
      Vector col;
      Matrix b;
      col=vector(n);
      indx=ivector(n);
      b=matrix(n,n);                             /* create a working copy of a in
b */

      for (i=1;i<=n;i++) for (j=1;j<=n;j++) b[i][j]=a[i][j];
      ludcmp(b,n,indx,&d);
      for (j=1;j<=n;j++) {
            for (i=1;i<=n;i++) col[i]=0.0;
            col[j]=1.0;
            lubksb(b,n,indx,col);
            for (i=1;i<=n;i++) y[i][j]=col[i];
      }
      free_matrix(b);
      free_ivector(indx);
      free_vector(col);
}

void lin_solve(Matrix a, int n, Vector b)
/* This function solves ax=b for x. a is an quadratic matrix of size n that
      is subsequently changed. The vector b is replaced by x. This function is
      explained on page 48 of the Numerical Recipes in C. */
{
      double d;
      int *indx;
      indx=ivector(n);
      ludcmp(a,n,indx,&d);
      lubksb(a,n,indx,b);
      free_ivector(indx);
}

/*****************************************************************************/
/******                                                                ******/
/******                      Utility Routines                          ******/
/******                                                                ******/
/*****************************************************************************/
#define TINY 1.0e-20;

void ludcmp(Matrix a, int n, int *indx, Vector d)
/* Numerical Recipes in C, page 46/47:
      Given a matrix a[1..n][1..n], this routine replaces it by the LU
      decomposition of a rowwise permutation of itself. a and n are input. a is
      output, arranged as in equation (2.3.14); indx[1..n] is an output vector
      that records the row permutation effected by the partial  pivoting; d is
      output as +/-1 depending on whether the number of row interchanges was
even
      or odd, respectively. This routine is used  in combination with lubksb()
to
      solve linear equations or invert a matrix. */
{
      int i, imax, j, k;
      double big, dum, sum, temp;
      Vector vv;

      vv=vector(n);
      *d=1.0;
      for (i=1;i<=n;i++) {
            big=0.0;
            for (j=1;j<=n;j++)
                  if ((temp=fabs(a[i][j])) > big) big=temp;
            if (big == 0.0) nrerror("Singular matrix in routine ludcmp");
            vv[i]=1.0/big;
      }
      for (j=1;j<=n;j++) {
            for (i=1;i<j;i++) {
                  sum=a[i][j];
                  for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                  a[i][j]=sum;
            }
            big=0.0;
            for (i=j;i<=n;i++) {
                  sum=a[i][j];
                  for (k=1;k<j;k++)
                        sum -= a[i][k]*a[k][j];
                  a[i][j]=sum;
                  if ( (dum=vv[i]*fabs(sum)) >= big) {
                        big=dum;
                        imax=i;
                  }
            }
            if (j != imax) {
                  for (k=1;k<=n;k++) {
                        dum=a[imax][k];
                        a[imax][k]=a[j][k];
                        a[j][k]=dum;
                  }
                  *d = -(*d);
                  vv[imax]=vv[j];
            }
            indx[j]=imax;
            if (a[j][j] == 0.0) a[j][j]=TINY;
            if (j!=n) {
                  dum=1.0/(a[j][j]);
                  for (i=j+1;i<=n;i++) a[i][j] *= dum;
            }
      }
      free_vector(vv);
}

void lubksb(Matrix a, int n, int *indx, double b[])
/* Numerical Recipes in C, page 47:
      Solves the set of n linear equations AX=B. Here a[1..n][1..n] is input,
not
      as the matrix A but rather as its LU decomposition, determined by the
      routine ludcmp(). indx[1..n] is input as the permutation vector returned
by
      ludcmp(). b[1..n] is input as the right-hand side vector B, and returns
with
      the solution vector X. a,n and indx are not modified by this routine and
can
      be left in place for successive calls with different right-hand sides b.
      This routine takes into account the possibility that b will begin with
many
      zero elements, so it is efficient for use in matrix inversion. */
{
      int i, ii=0,ip, j;
      double sum;

      for (i=1;i<=n;i++) {
            ip=indx[i];
            sum=b[ip];
            b[ip]=b[i];
            if (ii) for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
            else if (sum) ii=i;
            b[i]=sum;
      }
      for (i=n;i>=1;i--) {
            sum=b[i];
            for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
            b[i]=sum/a[i][i];
      }
}

/********************************************************************************/
/********                                                               *******/
/********                   Variable Metric Method                      *******/
/********                                                               *******/
/********************************************************************************/

/********************************************************************************/
/* Global Variables                                                             */
/********************************************************************************/
static double dsqrarg;
static double maxarg1, maxarg2;

#define DSQR(a) ((dsqrarg=(a)) == 0.0 ? 0.0 : dsqrarg*dsqrarg)
#define FMAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ? (maxarg1) :
(maxarg2))
#define ITMAX  500
#define EPS    1.e-15
#define TOLX   (4*EPS)
#define STPMX  200.
#define GTOL 1.e-12

void minimize(Vector x, int n, double (*f)(Vector),
                          void (*df)(Vector, Vector), Vector parameter)
/* Uses the Variable-Metric method to solve for the local minimum of the
```

```
      function f(x) where x is a vector of length n<=4. df() is the gradient of
f()
      and parameter is a vector containing constant parameters for f() and df().
*/
{
      void linsearch(Vector, int, double, Vector, Vector,double *, double,
                                    double (*f)(Vector), Vector);
      int i, its, j;
      double fret, den, fac, fad, fae, fp, stpmax, sum=0.0, sumdg, sumxi, temp,
test;
      double dg[5], g[5], hdg[5], xold[5], xi[5];
      Matrix hessin=matrix(4,4);

      fp= (*f)(parameter);
      (*df)(g,parameter);

      for (i=1;i<=n;i++) {                /* dfpmin begins here */
            for (j=1;j<=n;j++) hessin[i][j]=0.0;
            hessin[i][i]= 1.0;
            xi[i]= -g[i];
            sum += x[i]*x[i];
      }
      stpmax=STPMX*FMAX(sqrt(sum),(double)n);
      for (its=1;its<=ITMAX;its++) {
            for (i=1;i<=n;i++) xold[i]=x[i];
            linsearch(x, n, fp, g, xi, &fret, stpmax, f, parameter);
            fp= fret;
            for (i=1;i<=n;i++) xi[i]=x[i]-xold[i];
            test=0.0;
            for (i=1;i<=n;i++) {
              temp=fabs(xi[i])/FMAX(fabs(x[i]),1.0);
              if (temp > test) test=temp;
            }
            if (its >ITMAX-1 && test>1.e-5) test=0.;
            if (test < TOLX) {
                  free_matrix(hessin);
                  return;
            }
            for (i=1;i<=n;i++) dg[i]=g[i];
            (*df)(g, parameter);
            test=0.0;
            den=FMAX(fret,1.0);
            for (i=1;i<=n;i++) {
                  temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
                  if (temp > test) test=temp;
            }
            if (its >ITMAX-1 && test>1.e-5) test=0.;
            if (test < GTOL){
                  free_matrix(hessin);
                  return;
            }
            for (i=1;i<=n;i++) dg[i]=g[i]-dg[i];
            for (i=1;i<=n;i++) {
                  hdg[i]=0.0;
                  for (j=1;j<=n;j++) hdg[i] += hessin[i][j]*dg[j];
            }
            fac=fae=sumdg=sumxi=0.0;
            for (i=1;i<=n;i++){
                  fac += dg[i]*xi[i];
                  fae += dg[i]*hdg[i];
                  sumdg += DSQR(dg[i]);
                  sumxi += DSQR(xi[i]);
            }
            if (fac*fac > EPS*sumdg*sumxi){
                  fac=1.0/fac;
                  fad=1.0/fae;
                  for (i=1;i<=n;i++) dg[i]=fac*xi[i]-fad*hdg[i];
                  for (i=1;i<=n;i++) for (j=1;j<=n;j++)
                        hessin[i][j] += fac*xi[i]*xi[j]-
fad*hdg[i]*hdg[j]+fae*dg[i]*dg[j];
            }
            for (i=1;i<=n;i++) {
                  xi[i]=0.0;
                  for (j=1;j<=n;j++) xi[i] -= hessin[i][j]*g[j];
            }
      }
      nrerror("too many iterations in minimize");
      free_matrix(hessin);
}
#undef TOLX

#define ALF 1.e-4
#define TOLX 1.e-9
void linsearch(Vector x, int n, double fold, Vector g, Vector p, double *f,
             double stpmax, double (*funct)(Vector), Vector parameter)
/* Given the vector x of length n<=4, the value of the function fold and
gradient g there, and a direction p[1..2 or 4], linsearch finds a new point
state along the direction p from the original state where the function func()
has decreased "sufficiently". The new function value is returned in f. stpmax is
an input quantity that limits the length of the steps. p is usually the Newton
direction. Parameter is the vector for funct(). */
{
      int i;
      double a, alam, alam2, alamin, b, disc, f2, rhs1, rhs2, slope, sum;
      double temp, test, tmplam, xold[5];

      for (i=1;i<=n;i++) xold[i]=x[i];
      for (sum=0.0, i=1;i<=n;i++) sum += p[i]*p[i];
      sum=sqrt(sum);

      if(sum > stpmax) for (i=1; i<=n; i++) p[i] *= stpmax/sum;
      for (slope=0.0,i=1; i<=n; i++) slope += g[i]*p[i];
      test=0.0;
      for (i=1; i<=n; i++){
            temp=fabs(p[i])/FMAX(xold[i],1.0);
            if (temp > test) test=temp;
      }
      if (test == 0.0) alamin= 1.e+12;
      else alamin=TOLX/test;
      alam=1.0;
      for (;;) {
            for (i=1; i<=n; i++) x[i]=xold[i]+alam*p[i];
            *f= (*funct)(parameter);
            if(alam < alamin) {
                  for (i=1; i<=n; i++) x[i]=xold[i];
                  return;
            } else if (*f <= fold+ALF*alam*slope) return;
            else {
                  if (alam == 1.0) tmplam = -slope/(2.0*(*f-fold-slope));
                  else {
                        rhs1= *f-fold-alam*slope;
                        rhs2= f2-fold-alam2*slope;
                        a= (rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2);
                        b= (-
alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2);
                        if (a==0.0) tmplam= -slope/(2.0*b);
                        else {
                              disc= b*b-3.*a*slope;
                              tmplam= (-b+sqrt(fabs(disc)))/(3.0*a);
                        }
                        if (tmplam > 0.5*alam) tmplam= 0.5*alam;
                  }
            }
            alam2=alam;
            f2= *f;
            alam=FMAX(tmplam, 0.1*alam);
      }
}

double projection(Matrix *C, Vector cen, Vector r, int k)
/* This function returns the projection of a vector q[] along tangent direction
of nodal point k. */
{
      int i;
      double tan[4],q[4];

      tan[1]=fabs(C[1][k][3]);  /* tangent vector at nodal point k */
      tan[2]=C[2][k][3];
      tan[3]=C[3][k][3];
      for (i=1;i<=3;i++) q[i]=r[i]-cen[i];
      return DOT3(q,tan);
}
```

```
MEDUSA = MOU309_main.c MOU309_common.o MOU309_init.o MOU309_contact.o
MOU309_vehicle.o MOU309_road.o
medusa: $(MEDUSA)
      cc -g -std1 $(MEDUSA) -lm -o medusa

common.o:
      cc -g -c MOU309_common.c -o MOU309_common.o
init.o:
      cc -g -c MOU309_init.c -o MOU309_init.o
contact.o:
      cc -g -c MOU309_contact.c -o MOU309_contact.o
vehicle.o:
      cc -g -c MOU309_vehicle.c -o MOU309_vehicle.o
road.o:
      cc -g -c MOU309_road.c -o MOU309_road.o
```

```
% physical parameters for the vehicle models
%
% created by Peter Varadi, last modified May 6, 1997

NUMBER_OF_MODELS 2

% description of Model 1 starts here
MODEL 1
 COSSERAT_POINT
  MASS 1573.0  % mass of car [kg]
      Ix    479.6  % moments of inertia along principal axes [kg m^2]
      Iy   2594.6
      Iz   2782.0
      E   600.0e8  % Young's modulus [N/m^2]
      nu  0.30     % Poisson's ratio
      volume 0.42  % assumed volume of the Chassis [m^3]
 SUSPENSION
      L1 1.0     % distance from cg to front axle [m]
      L2 1.6     % distance from cg to rear axle [m]
      B  1.2     % track of axle [m]
      H1 0.0       % vertical distance from cg to front assembly pts.
      H2 0.0       % and to rear assembly points [m] (assumed)
      spring_ref 0.15  % reference length of spring [m]
      C1 40000.0   % spring constant for front wheel suspension [N/m]
      C2 40000.0   % spring constant for rear wheel suspension [N/m]
      D1 1500.0    % damping coeff. for front wheel suspension [Ns/m]
      D2 1200.0    % damping coeff. for rear wheel suspension [Ns/m]
 TIRE 0.0016    % lag parameter for tire model [s]
 CONTACT
      A1 4.0       % dimensions of a vehicle: length [m]
      A2 1.60       % width [m]
      A3 1.30       % height [m] (Don't make two of them equal!)
 EQUILIBRIUM
      R3  5.039617e-02   % vertical position of vehicle's center of mass [m]
        D11  1.0   D12  0.0           D13 0.0 % director 1 [.]
        D21  0.0            D22  1.000000e+00  D23  0.0          % director 2
[.]
        D31  0.0   D32  0.0           D33 1.0 % director 3 [.]
% description of model 1 ends here

% description of Model 2 starts here
MODEL 2
 COSSERAT_POINT
  MASS 1573.0  % mass of car [kg]
      Ix    479.6  % moments of inertia along principal axes [kg m^2]
      Iy   2594.6
      Iz   2782.0
      E   600.0e6  % Young's modulus [N/m^2]
      nu  0.30     % Poisson's ratio
      volume 0.42  % assumed volume of the Chassis [m^3]
 SUSPENSION
      L1 1.0     % distance from cg to front axle [m]
      L2 1.6     % distance from cg to rear axle [m]
      B  1.2     % track of axle [m]
      H1 0.0       % vertical distance from cg to front assembly pts.
      H2 0.0       % and to rear assembly points [m] (assumed)
      spring_ref 0.15  % reference length of spring [m]
```

```
     C1 40000.0   % spring constant for front wheel suspension [N/m]
     C2 40000.0   % spring constant for rear wheel suspension [N/m]
     D1 15000.0    % damping coeff. for front wheel suspension [Ns/m]
     D2 12000.0    % damping coeff. for rear wheel suspension [Ns/m]
TIRE 0.0016    % lag parameter for tire model [s]
CONTACT
     A1 4.0       % dimensions of a vehicle: length [m]
     A2 1.60       % width [m]
     A3 1.30       % height [m] (Don't make two of them equal!)
EQUILIBRIUM
     R3  5.039617e-02   % vertical position of vehicle's center of mass [m]
       D11  -0.001   D12 1.0           D13 0.0 % director 1 [.]
       D21  -1.0            D22  -0.001  D23  0.0          % director 2 [.]
       D31  0.0   D32  0.0           D33 1.0 % director 3 [.]
% description of model 2 ends here
```

```
%
% Initialization of Platoon
%
% created by Peter Varadi, last modified june 26, 1997

NUMBER_OF_VEHICLES 1

% Vehicle 1
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 0.0                  % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [.] (cw:"-", ccw:"+")
  SPEED  50.0            % forward speed [m/s]
  TIRE_SPEED 50.0         % in [m/s], if 0.0, tires roll freely
  STEERING 0.0           % steer angle [rad]

% Vehicle 2
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 6.0                  % X coordinate of center of mass [m]
  Y 0.4                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [.] (cw:"-", ccw:"+")
  SPEED  20.0            % forward speed [m/s]
  TIRE_SPEED 0.0         % in [m/s], if 0.0, tires roll freely
  STEERING 0.0           % steer angle [rad]

% Vehicle 3
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 12.0                 % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [.] (cw:"-", ccw:"+")
  SPEED  22.0            % forward speed [m/s]
  TIRE_SPEED 0.0         % in [m/s], if 0.0, tires roll freely
  STEERING 0.0           % steer angle [rad]

% Vehicle 4
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 18.0                 % X coordinate of center of mass [m]
  Y 0.0                  % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [.] (cw:"-", ccw:"+")
  SPEED  22.0            % forward speed [m/s]
  TIRE_SPEED 0.0         % in [m/s], if 0.0, tires roll freely
  STEERING 0.0           % steer angle [rad]

% Vehicle 5
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 6.0                  % X coordinate of center of mass [m]
  Y -4.0                 % Y coordinate of center of mass [m]
  ORIENTATION 0.0        % heading angle [.] (cw:"-", ccw:"+")
  SPEED  25.0            % forward speed [m/s]
  TIRE_SPEED 24.1        % in [m/s], if 0.0, tires roll freely
  STEERING 8.0           % steer angle [rad]
```

```
NUMBER_OF_POINTS:
7
XYZ_COORDINATES_&_ANGLE:
0.      0.0   0.0   0.0
20.     0.0   0.0   0.0
40.     0.0   0.0   0.0
60      0.0   0.0   0.0
120     0.0   0.0   0.0
280     0.0   0.0   0.0
700     0.0   0.0   0.0
END_OF_FILE
```