Title

Models Of Vehicular Collision: Development And Simulation With Emphasis On Safety III: Computer Code Programmer's Guide And User Manual For Medusa

Authors

O'reilly, O. M.
Papadopoulos, P.
Lo, G.-j.
et al.

Publication Date

1998

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

Models of Vehicular Collision: Development and Simulation with Emphasis on Safety III: Computer Code, Programmer's Guide and User Manual for MEDUSA

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

California PATH Research Report
UCB-ITS-PRR-98-10

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 232

February 1998

ISSN 1055-1425

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

REPORT – September 1997

Submitted to: PATH (MOU 232)

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

Department of Mechanical Engineering
University of California, Berkeley

Abstract

In this report, which also constitutes a final report for MOU232, a User's Manual and a Programmer's Guide for the program MEDUSA is presented. This program is capable of simulating the impacts of several vehicles. In particular, it assumes that the collisions are elastic, and is consequently applicable for low relative velocity impacts. The vehicular model and collision detection algorithm used in the program are based on previously published reports by the authors. Several of these developments are summarized, and improved features of the collision detection algorithm are presented in a theoretical section at the beginning of this report.

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

Executive Summary

This report, which constitutes a final report for the contract MOU232, is a User's Manual and a Programmer's Guide for the program MEDUSA. In addition, a section outlining the theoretical developments implemented in this program is presented and the source code is provided.

The program MEDUSA is an *ANSI-C* based simulation package which is designed to simulate the impact of an arbitrary number of vehicles which are moving on a horizontal surface. The vehicular models are partially based on the theory of a Cosserat point. This theory was developed by M. B. Rubin in the mid 1980's and subsequently extended in the early 1990's by A. E. Green and P. M. Naghdi. This theory models the (elastic) deformation of each vehicle during a motion and possible impacts with other vehicles. Additional features include the suspension and tyre models, and a collision detection algorithm. The algorithm models the lateral surfaces of a vehicle using a deformable ellipsoid. The development of the vehicle model was outlined in two earlier reports by the authors. However, in the first section of this report, additional features of the collision detection algorithm are presented.

This report also presents a User's Manual for the program MEDUSA. It is intended to give the reader the requisite background on using the program to simulate the motion of any number of vehicles which may collide with one another. It also contains sample input and output files in the appendices.

Presently, MEDUSA is suitable for modeling and simulating low relative velocity impacts of vehicles. In these cases, no permanent damage to the vehicles arises, and the program then provides data on the post-collision behavior of the vehicles. It should be noted that the model only allows for limited modes of deformation. This feature enchances the program's computational efficiency and makes it ideally suited to large scale simulations. In order to provide the user with an avenue for improving the model, a Programmer's Guide is presented here. It can be used to modify the code so as to incorporate other tyre models or an increased number of directors in the Cosserat point model, for instance.

Both the code for MEDUSA and the vehicle models will be updated and improved in the future. In so doing, the contents of this report will also be updated. The reader interested in obtaining these updated versions should contact either Prof. O. M. O'Reilly (oreilly@me.berkeley.edu) or Prof. P. Papadopoulos (panos@me.berkeley.edu).

Contents

Chapter 0

Conventions

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

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

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

Theoretical Background

1.1 Introduction

In this section, we recall some of the background for the vehicle model and the contact detection algorithm used by MEDUSA. These developments were discussed in earlier reports, however some of the material presented here updates these reports. Furthermore, we outline the numerical time-integration routines used in the simulations.

1.2 Review of the Vehicle Model

The vehicle model that MEDUSA uses is discussed in O'Reilly, Papadopoulos, Lo and Varadi [9, 10] where the references for the different parts of the model are listed. In this section, we briefly review the equations of the model which will be referred to later when we discuss the program code of MEDUSA in Chapter 3.

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 the 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 used (cf. equation (0.1)).

The chassis of the vehicle is modeled using the theory of a Cosserat point which was introduced by Rubin [13], and subsequently developed by Green and Naghdi [4]. In this

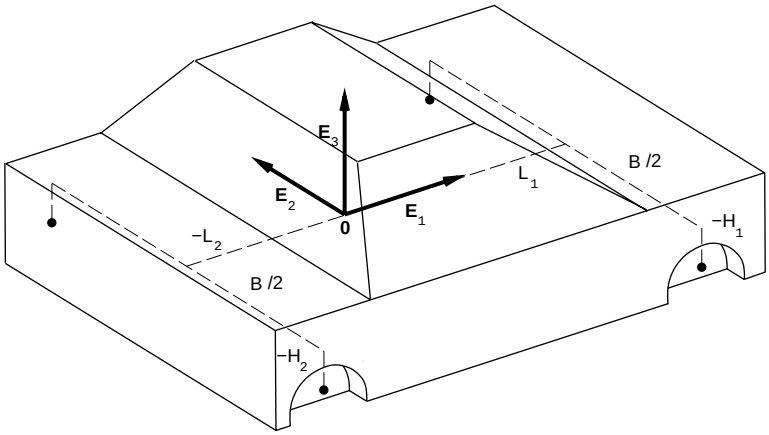

Figure 1.1: *Schematic depiction of the reference configuration of the chassis. The coordinates of the suspension assembly points are also shown.*

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 [1] 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 [15, Sections 42-46]. We will use this property later to determine if two vehicles are in contact. To that end, 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_σ^j 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 the 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 $\mathbf{J}_0 = J_0^{ij}\mathbf{E}_i \otimes \mathbf{E}_j$ as follows:

$$my^{ij} = -J_0^{ij} = 0 \quad , \quad i \neq j \quad , \tag{1.8}$$

$$\begin{pmatrix} J_0^{11} \\ J_0^{22} \\ J_0^{33} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} my^{11} \\ my^{22} \\ my^{33} \end{pmatrix} \quad . \tag{1.9}$$

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.

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

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

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

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

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

[1] It is usual 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.

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

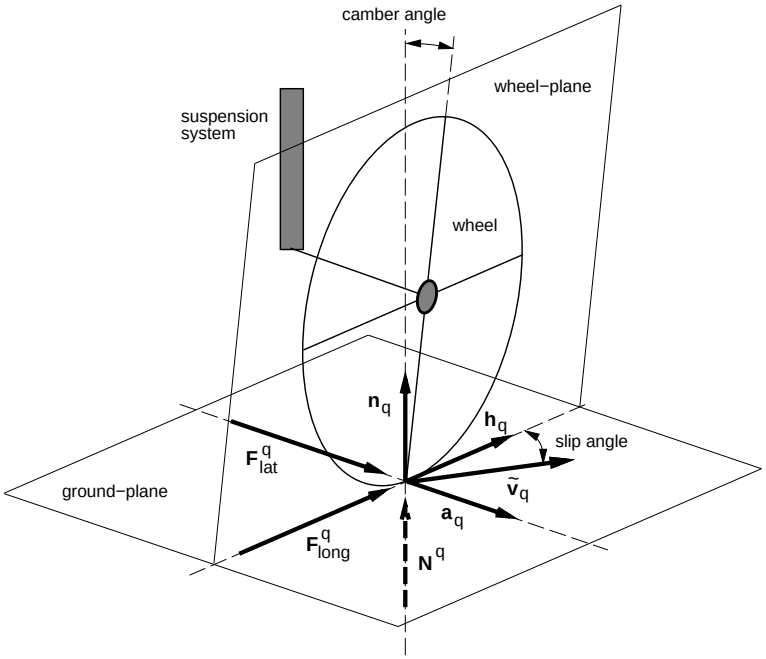

Figure 1.2: *Schematic depiction of one tyre illustrating the forces and kinematic quantities defined in the text. Here,* $\mathbf{n}_q$ *is the surface normal, such that* $\{\mathbf{a}_q, \mathbf{h}_q, \mathbf{n}_q\}$ *form a local orthonormal basis. The force* $\mathbf{N}^q = -F^q_{susp}\mathbf{E}_3$ *is the normal contact force. The camber angle is assumed to be negligible.*

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

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

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 [14]):

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

1.2.2 Suspension and Tyre Forces

In the current implementation of MEDUSA, the road is assumed to be a horizontal plane. This simplifies the equations and the program code considerably.

It is assumed that the unit heading vectors $\mathbf{h}_3 = \mathbf{h}_4$ of the rear wheels are parallel to the projection of $\mathbf{d}_1$ onto the road plane (see Figure 1.2), while the unit orientation vectors

$\mathbf{a}_3 = \mathbf{a}_4$ of the rear wheels are perpendicular to $\mathbf{h}_3$ and $\mathbf{h}_4$, respectively, i.e.,

$$\mathbf{h}_3 = \mathbf{h}_4 = \frac{(\mathbf{d}_1 \cdot \mathbf{E}_1)\,\mathbf{E}_1 + (\mathbf{d}_1 \cdot \mathbf{E}_2)\,\mathbf{E}_2}{||(\mathbf{d}_1 \cdot \mathbf{E}_1)\,\mathbf{E}_1 + (\mathbf{d}_1 \cdot \mathbf{E}_2)\,\mathbf{E}_2||} \quad ,$$

$$\mathbf{a}_3 = \mathbf{a}_4 = \frac{-(\mathbf{d}_1 \cdot \mathbf{E}_2)\,\mathbf{E}_1 + (\mathbf{d}_1 \cdot \mathbf{E}_1)\,\mathbf{E}_2}{||-(\mathbf{d}_1 \cdot \mathbf{E}_2)\,\mathbf{E}_1 + (\mathbf{d}_1 \cdot \mathbf{E}_1)\,\mathbf{E}_2||} \quad . \tag{1.14}$$

Similarly, if we define a steering angle Θ about $\mathbf{E}_3$, the unit heading vectors $\mathbf{h}_1 = \mathbf{h}_2$ and the unit orientation vectors $\mathbf{a}_1 = \mathbf{a}_2$ of the (steered) front wheels are given by

$$\mathbf{h}_1 = \mathbf{h}_2 = \cos\Theta\,\mathbf{h}_3 + \sin\Theta\,\mathbf{a}_3 \quad , \quad \mathbf{a}_1 = \mathbf{a}_2 = -\sin\Theta\,\mathbf{h}_3 + \cos\Theta\,\mathbf{a}_3 \quad . \tag{1.15}$$

We assume that the wheels are massless. In the simplified suspension and tyre models discussed in O'Reilly, Papadopoulos, Lo and Varadi [9], the applied forces $\mathbf{f}^q$ in equations (1.7) are calculated using

$$\mathbf{f}^q = -F^q_{susp}\,\mathbf{E}_3 + F^q_{lat}\,\mathbf{a}_q \quad , \tag{1.16}$$

where F^q_{susp} are the magnitudes of the suspension forces:

$$F^q_{susp} = C^q(\mathbf{r} + X^i_q\,\mathbf{d}_i)\cdot\mathbf{E}_3 - \Delta s + D^q(\mathbf{v} + X^i_q\,\mathbf{w}_i)\cdot\mathbf{E}_3 \ , \ C^q, D^q = \begin{cases} C_1, D_1 & : \ q = 1, 2. \\ C_2, D_2 & : \ q = 3, 4. \end{cases} \tag{1.17}$$

It is important to note that the tyre model we use does not accommodate braking or accelerating of the vehicle. The parameters $\{C_1, D_1\}$ and $\{C_2, D_2\}$ are the linear spring coefficients and the linear damping coefficients of the front and rear suspensions, respectively. The unstretched spring length Δs is assumed to be the same for all four suspensions.

Equation (1.16) expresses the assumption that the wheels roll without resistance. One has therefore only the wheel force F^q_{lat} acting laterally on each wheel. To calculate this force, we first need to calculate the (side) slip angle α_q which is defined as the angle between the wheel heading $\mathbf{h}_q$ and the direction of wheel travel $\tilde{\mathbf{v}}_q$, where

$$\tilde{\mathbf{v}}_q = (\mathbf{v}_q \cdot \mathbf{E}_1)\,\mathbf{E}_1 + (\mathbf{v}_q \cdot \mathbf{E}_2)\,\mathbf{E}_2 \quad , \quad \mathbf{v}_q = \mathbf{v} + X^i_q\,\mathbf{w}_i \quad . \tag{1.18}$$

Using equations (1.15) and (1.14), the slip angles are

$$\alpha_q = \arctan\left(\frac{(\mathbf{h}_q \times \tilde{\mathbf{v}}_q)\cdot\mathbf{E}_3}{\mathbf{h}_q \cdot \tilde{\mathbf{v}}_q}\right) \quad \text{(no sum on } q\text{)}. \tag{1.19}$$

We assume that the force response of the tyres is delayed by an amount which is described by a first order differential equation with a parameter $\tau^{-1}\ [rad/s]$ from which the four delayed or lagged slip angles $\alpha_{lag,q}$ are calculated:

$$\tau\,\dot{\alpha}_{lag,q} + \alpha_{lag,q} = \alpha_q \quad . \tag{1.20}$$

For each wheel q, the lateral tyre forces F_{lat}^q are now calculated using the following identities recorded in Kortüm and Sharp [6] for a Calspan tyre model. For readability, we drop the wheel index q:

$$F_{lat} = \mu_y F_{susp}\, g(\bar{\alpha}) \quad , \quad \mu_y = (-B_1 F_{susp} + B_3 + B_4 F_{susp}^2) SN \quad . \tag{1.21}$$

In this equation, g is called the side force shaping function given by

$$\begin{aligned} g(\bar{\alpha}) &= \bar{\alpha} - \frac{1}{3}\bar{\alpha}|\bar{\alpha}| + \frac{1}{27}\bar{\alpha}^3 \quad \text{for } |\bar{\alpha}| < 3 \quad , \\ g(\bar{\alpha}) &= \frac{\bar{\alpha}}{|\bar{\alpha}|} \quad \text{for } |\bar{\alpha}| \leq 3 \quad , \end{aligned} \tag{1.22}$$

where the dimensionless sideslip angle $\bar{\alpha}$ is calculated using

$$\begin{aligned} \bar{\alpha} &= -\left(\frac{A_1 F_{susp}(F_{susp} + A_2) - A_0 A_2}{A_2 \mu_y F_{susp}} \right) \alpha_{lag} \quad \text{if } (-F_{susp}) \leq A_2 \quad , \\ \bar{\alpha} &= \left(\frac{A_0}{\mu_y F_{susp}} \right) \alpha_{lag} \quad \text{if } (-F_{susp}) > A_2 \quad . \end{aligned} \tag{1.23}$$

The suspension force F_{susp} is calculated using equation (1.17). Note that F_{lat} is positive for a positive (lagged) slip angle. The numerical values for this Calspan tyre model are

$$A_0 = 2625\ [N] \quad , \quad A_1 = 14.47\ [.] \quad , \quad A_2 = 12930\ [N] \quad , \quad SN = 1.0274\ [.] \quad ,$$

$$B_1 = -0.464 \cdot 10^{-4}\ [N^{-1}] \quad , \quad B_3 = 1.216\ [.] \quad , \quad B_4 = 0.218 \cdot 10^{-10}\ [N^{-2}] \quad . \tag{1.24}$$

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:

$$r_j = \mathbf{r} \cdot \mathbf{E}_j \quad , \quad d_{ij} = \mathbf{d}_i \cdot \mathbf{E}_j \quad , \quad k_j^i = \mathbf{k}^i \cdot \mathbf{E}_j \quad , \quad f_j^q = \mathbf{f}^q \cdot \mathbf{E}_j \quad , \tag{1.25}$$

$$\begin{aligned} (\mathbf{z}_1) &= (r_1\,,\, r_2\,,\, r_3\,,\, d_{11}\,,\, d_{12}\,,\, d_{13}\,,\, d_{21}\,,\, d_{22}\,,\, d_{23}\,,\, d_{31}\,,\, d_{32}\,,\, d_{33})^T \ , \\ (\mathbf{z}_2) &= (v_1\,,\, v_2\,,\, v_3\,,\, w_{11}\,,\, w_{12}\,,\, w_{13}\,,\, w_{21}\,,\, w_{22}\,,\, w_{23}\,,\, w_{31}\,,\, w_{32}\,,\, w_{33})^T \ , \\ (\boldsymbol{\alpha}) &= (\alpha_1\,,\, \alpha_2\,,\, \alpha_3\,,\, \alpha_4)^T \ , \\ (\mathbf{k}) &= \left(0\,,\, 0\,,\, 0\,,\, k_1^1\,,\, k_2^1\,,\, k_3^1\,,\, k_1^2\,,\, k_2^2\,,\, k_3^2\,,\, k_1^3\,,\, k_2^3\,,\, k_3^3\right)^T \ , \\ (\mathbf{f}) &= \left(f_1^1\,,\, f_2^1\,,\, f_3^1\,,\, f_1^2\,,\, f_2^2\,,\, f_3^2\,,\, f_1^3\,,\, f_2^3\,,\, f_3^3\,,\, f_1^4\,,\, f_2^4\,,\, f_3^4\right)^T \ , \\ (\mathbf{u}) &= (0\,,\, 0\,,\, -mg\,,\, 0\,,\, 0\,,\, 0\,,\, 0\,,\, 0\,,\, 0\,,\, 0\,,\, 0\,,\, 0)^T \ . \end{aligned} \tag{1.26}$$

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

$$\begin{aligned} \dot{\mathbf{z}}_1 &= \mathbf{z}_2 \quad , \\ \dot{\mathbf{z}}_2 &= \mathbf{M}^{-1}\left[-\mathbf{k}(\mathbf{z}_1) + \mathbf{A}\mathbf{f}(\mathbf{z}_1, \mathbf{z}_2, \boldsymbol{\alpha}_{lag}, t) + \mathbf{u}\right] \equiv \mathbf{g}(\mathbf{z}_1, \mathbf{z}_2, \boldsymbol{\alpha}_{lag}, \mathbf{u}, t) \quad , \\ \dot{\boldsymbol{\alpha}}_{lag} &= \tau^{-1}\boldsymbol{\alpha}_{lag} + \boldsymbol{\alpha} \quad , \end{aligned} \tag{1.27}$$

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

$$(\mathbf{M}) = \begin{pmatrix} m\,\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & my^{11}\,\mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & my^{22}\,\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & my^{33}\,\mathbf{I} \end{pmatrix} \quad , \quad (\mathbf{A}) = \begin{pmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{I} \\ X_1^1\,\mathbf{I} & X_2^1\,\mathbf{I} & X_3^1\,\mathbf{I} & X_4^1\,\mathbf{I} \\ X_1^2\,\mathbf{I} & X_2^2\,\mathbf{I} & X_3^2\,\mathbf{I} & X_4^2\,\mathbf{I} \\ X_1^3\,\mathbf{I} & X_2^3\,\mathbf{I} & X_3^3\,\mathbf{I} & X_4^3\,\mathbf{I} \end{pmatrix} \quad , \tag{1.28}$$

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_q^i are defined in equation (1.11) and Figure 1.1.

The program code of MEDUSA makes repeated use of the vector

$$(\mathbf{z}) = \left(\mathbf{z}_1^T\,,\,\mathbf{z}_2^T\,,\,\boldsymbol{\alpha}_{lag}^T\right)^T \tag{1.29}$$

which has 28 components. This vector will be referred to as the state vector of the system (1.27).

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:

$$E = T + m\psi + \sum_{q=1}^{4} V^q + U \quad . \tag{1.30}$$

Using the notation from equations (1.17), (1.26) and (1.27), the energy terms are defined as follows:

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

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

$$V^q = \frac{1}{2}C^q\left((r_3 + X_q^i\,d_{i3} - \Delta s\right)^2 \quad , \quad C^q = \begin{cases} C_1 & : \; q = 1,2 \\ C_2 & : \; q = 3,4 \end{cases} \quad . \tag{1.31}$$

Note that the expression for $m\psi$ is consistent with equation (1.12) (see O'Reilly, Papadopoulos, Lo and Varadi [9]).

1.3 Contact Constraints and Contact Forces

When two vehicles come into contact, contact forces associated with the constraint of impenetrability (see O'Reilly and Varadi [11]) 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 1.3. 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

$$\begin{aligned} \phi_1 &= (\mathbf{r}^{\sigma}_{(2)} - \mathbf{r}^{\sigma}_{(1)}) \cdot \mathbf{n}_{(1)} \\ &= \left(\mathbf{r}_{(2)} + X^i_{\sigma(2)}\, \mathbf{d}_{(2),i} - \mathbf{r}_{(1)} - X^i_{\sigma(1)}\, \mathbf{d}_{(1),i}\right) \cdot \mathbf{n}_{(1)} = \hat{\phi}_1(\mathbf{z}_{(1),1}, \mathbf{z}_{(2),1}) \end{aligned} \tag{1.32}$$

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:

$$\begin{aligned} \phi_2 &= (\overset{*}{\mathbf{r}}{}^{\sigma}_{(2)} - \dot{\mathbf{r}}^{\sigma}_{(1)}) \cdot \mathbf{n}_{(1)} \\ &= \hat{\phi}_2(\mathbf{z}_{(1),1}, \mathbf{z}_{(1),2}, \mathbf{z}_{(2),1}, \mathbf{z}_{(2),2}) \ . \end{aligned} \tag{1.33}$$

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 (1.27) of the two vehicles $\beta = 1, 2$ as follows:

$$\begin{aligned} \dot{\mathbf{z}}_{(\beta),1} &= \mathbf{z}_{(\beta),2} + \mathbf{c}_{(\beta),1} \ , \\ \dot{\mathbf{z}}_{(\beta),2} &= \mathbf{g}_{(\beta)} + \mathbf{c}_{(\beta),2} \ , \\ \dot{\boldsymbol{\alpha}}_{(\beta),lag} &= \tau^{-1}_{(\beta)}\, \boldsymbol{\alpha}_{(\beta),lag} + \boldsymbol{\alpha}_{(\beta)} \ . \end{aligned} \tag{1.34}$$

We will refer to $\mathbf{c}_{(\beta),1}$ and $\mathbf{c}_{(\beta),2}$ as the (generalized) contact forces. They are calculated from the constraints (1.32) and (1.33) using a normality assumption:

$$\mathbf{c}_{(\beta),1} = \gamma_1 \frac{\partial \hat{\phi}_1}{\partial \mathbf{z}_{(\beta),1}} \ , \qquad \mathbf{c}_{(\beta),2} = \gamma_2 \frac{\partial \hat{\phi}_2}{\partial \mathbf{z}_{(\beta),2}} \ . \tag{1.35}$$

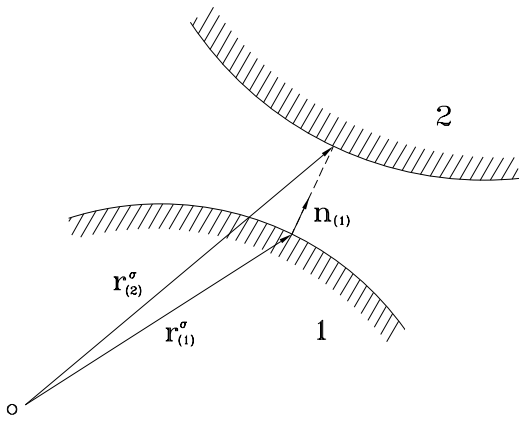

Figure 1.3: *Schematic depiction of the kinematical quantities involved in describing the contact between two bodies.*

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 1.5. In the interest of brevity, we do not write the forces (1.35) in a component form similar to (1.26). 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.

1.4 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 ellipsoidal surfaces.

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

$$||\mathbf{r}_{(2)} - \mathbf{r}_{(1)}|| \leq \rho_{(1)} + \rho_{(2)} \tag{1.36}$$

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].

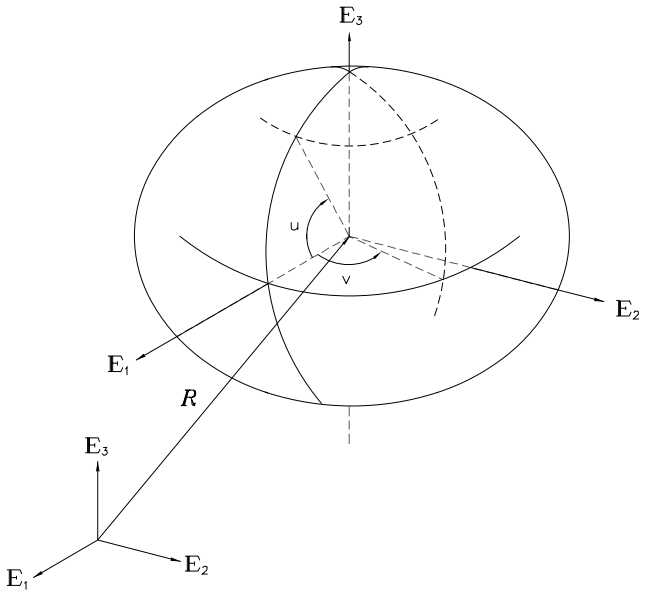

Figure 1.4: *Parametric representation of the vehicle ellipsoid in the reference configuration.*

1.4.1 The Vehicle Ellipsoid

For convenience, we approximate the outer surface σ of each vehicle in the reference configuration by an ellipsoid whose major axes are parallel to $\mathbf{E}_i$ and whose geometric center is identical to the center of mass of the vehicle, i.e.,[4]

$$\left(\frac{X_\sigma^1}{A}\right)^2 + \left(\frac{X_\sigma^2}{B}\right)^2 + \left(\frac{X_\sigma^3}{C}\right)^2 = 1 \quad , \tag{1.37}$$

where A, B and C are the lengths of the semi-axes of the ellipsoid. This surface has the parametric representation

$$X_\sigma^1 = A\cos(u)\cos(v) \quad , \quad X_\sigma^2 = B\cos(u)\sin(v) \quad , \quad X_\sigma^3 = C\sin(u) \quad , \tag{1.38}$$

[3]In MEDUSA, this idea has been implemented with $\rho = \max(A, B, C)$, where A, B and C are defined in equation (1.37).

[4]It has come to our attention that these assumptions may be too restrictive. We intend to relax them in future versions of MEDUSA.

where $u \in [-\pi/2, \pi/2]$, $v \in [0, 2\pi)$ are the curvilinear surface coordinates of the ellipsoid (see also Figure 1.4).

Under the action of the deformation gradient $\mathbf{F}(t)$ defined in equation (1.3), the material surface σ defined in equation (1.37) subsequently deforms into a surface which is described by equation (1.4). As mentioned in the lines following that equation, that surface is also ellipsoidal. In particular, the surface will remain convex.

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

$$\mathbf{a}_u = \frac{\partial \mathbf{r}^\sigma}{\partial u} \quad , \quad \mathbf{a}_v = \frac{\partial \mathbf{r}^\sigma}{\partial v} \quad , \quad \mathbf{n} = \frac{\mathbf{a}_v \times \mathbf{a}_u}{||\mathbf{a}_v \times \mathbf{a}_u||} \quad . \tag{1.39}$$

1.4.2 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 ellipsoids. Consider the distance function

$$f = ||\mathbf{r}^\sigma_{(1)} - \mathbf{r}^\sigma_{(2)}|| = f(\mathbf{x}) \quad , \quad \mathbf{x} = (u_{(1)}, v_{(1)}, u_{(2)}, v_{(2)}) \quad . \tag{1.40}$$

We would like to define our contact points from Section 1.3 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 this minimum, we should consult Figure 1.5. It is intuitively clear that we will not able to find unique points K and L when the vehicles are interpenetrating, in which case $\inf(f) = 0$ on the intersection curve. This case needs some additional work which is outlined in Section 1.4.3.

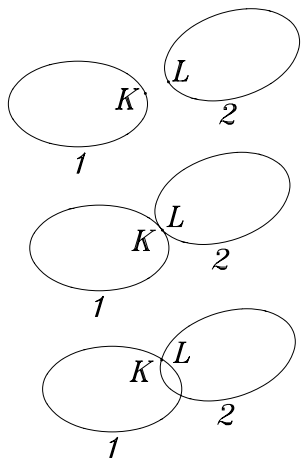

Figure 1.5: *The contact situations between two bodies*

In order to find the points K and L on the respective ellipsoids that minimize the distance function f, we start with two *arbitrary* points K_1 and L_1 on the respective surfaces (see also Figure 1.6). 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, ... which converges to two points K and L (which will not be unique if there is a curve of intersection).

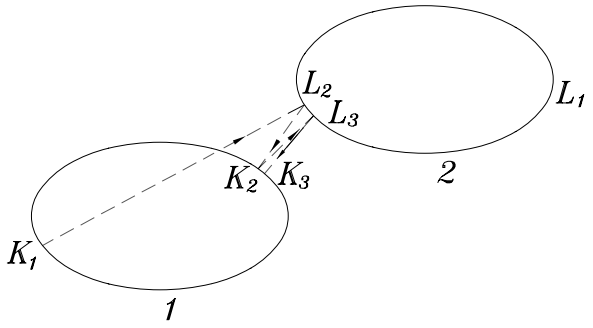

Figure 1.6: *From initial guesses K_1 and L_1 a sequence of points L_2, K_2, L_3, ... is calculated which minimizes the distance function f defined in the text.*

The algorithm employed in MEDUSA to perform the series of local minimizations of the distance function f is called the *variable metric* method whose details are discussed in Appendix A. Note that every one of these minimization steps will find a unique (local) minimum of f. This is due to the fact that, in each step, we are minimizing the distance from a point to a convex surface (for further details, see Kowalik [8]).

Now consider Figure 1.5 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 (1.39) satisfy[5] $\mathbf{n}_{(1)} \approx -\mathbf{n}_{(2)}$ only if the ellipsoids are not penetrating. We can therefore specify the following no-contact condition:

$$(\mathbf{r}^{\sigma}_{(2)} - \mathbf{r}^{\sigma}_{(1)}) \cdot \mathbf{n}_{(1)} > 0 \quad , \quad \mathbf{n}_{(1)} \approx -\mathbf{n}_{(2)} \quad . \tag{1.41}$$

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 ellipsoids. We now turn to this matter.

1.4.3 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 ellipsoids. 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.

[6] Clearly, this is one possibility to define unique contact points. There are others, however, the detection of the maximum penetration points is considered to be the easiest method.

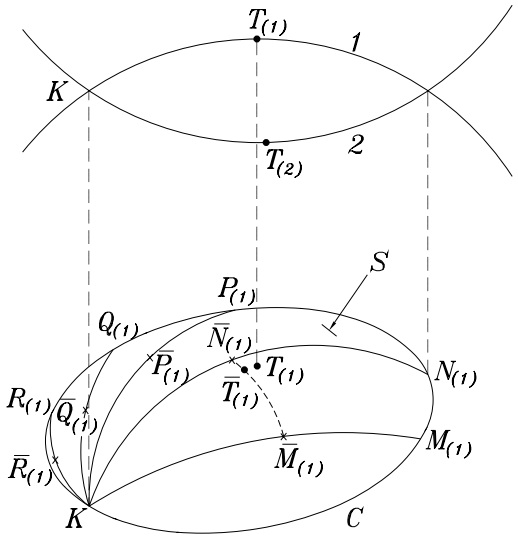

Figure 1.7: *Unique contact point detection scheme*

Recall that the equation of the ellipsoid for each vehicle in its reference configurations has the form (1.37) which can be rewritten as

$$(\mathbf{R}^{\sigma}_{(\beta)} - \mathbf{R}_{(\beta)}) \cdot \mathbf{K}_{(\beta)} (\mathbf{R}^{\sigma}_{(\beta)} - \mathbf{R}_{(\beta)}) = 1 \quad , \quad \beta = 1, 2 \quad , \tag{1.42}$$

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 ellipsoid in the current configuration can be expressed as

$$(\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) \cdot \hat{\mathbf{K}}_{(\beta)} (\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) = 1 \quad , \quad \beta = 1, 2 \quad , \tag{1.43}$$

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

$$\hat{f}_{(\beta)} = (\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) \cdot \hat{\mathbf{K}}_{(\beta)} (\mathbf{r}^{\sigma}_{(\beta)} - \mathbf{r}_{(\beta)}) - 1 \quad , \quad \beta = 1, 2 \quad . \tag{1.44}$$

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

Consider the curve $\mathcal{C}$ which is the intersection of two ellipsoids in three-dimensional Euclidean space and the point K computed using the previous iteration. The point corresponding to the maximum penetration of ellipsoid 1 into the ellipsoid 2, denoted as $T_{(1)}$,

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

is the point on the surface of ellipsoid 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 ellipsoid 2 into ellipsoid 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 ellipsoids.

The procedure for unique contact point detection is commenced by circling around point K using a tiny perturbation on the surface of ellipsoid 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 ellipsoid 2 into ellipsoid 1 can be obtained using a similar procedure.

1.5 Time Integration

Classical explicit time integration methods have proven to be unsatisfactory when solving the equations of motion (1.27) and (1.34) 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 (1.34) when two vehicles are in contact. The corresponding schemes for equation (1.27), and for additional vehicles, are easily inferred. To simplify the notation, we supress the vehicle index β here:

$$
\begin{aligned}
\tilde{\mathbf{z}}_{1,k+1} &= \mathbf{z}_{1,k} + \Delta t\, \mathbf{z}_{2,k} \ , \\
\gamma_{1,k+1} &= \gamma_{1,k} + p_1\, \phi_1(\tilde{\mathbf{z}}_{1,k+1}) \ , \\
\gamma_{2,k+1} &= \gamma_{2,k} + p_2\, \phi_2(\tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k}) \ , \\
\mathbf{z}_{1,k+1} &= \mathbf{z}_{1,k} + \Delta t \left[\mathbf{z}_{2,k} + \mathbf{c}_1(\gamma_{1,k+1}, \tilde{\mathbf{z}}_{1,k+1})\right] \ , \\
\mathbf{z}_{2,k+1} &= \mathbf{z}_{2,k} + \Delta t \left[\mathbf{g}(\tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k}, \boldsymbol{\alpha}_{lag,k}, t) + \mathbf{c}_2(\gamma_{2,k+1}, \tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k})\right] \ , \\
\boldsymbol{\alpha}_{lag,k+1} &= \boldsymbol{\alpha}_{lag,k} + \Delta t \left[\tau^{-1}\, \boldsymbol{\alpha}_{lag,k} + \boldsymbol{\alpha}(\tilde{\mathbf{z}}_{1,k+1}, \mathbf{z}_{2,k})\right] \ ,
\end{aligned}
\tag{1.45}
$$

where quantities of the form x_k are approximations of $x(t_k)$. In these incremental equations, Δt is the step size. Equation $(1.45)_1$ 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 (1.45) use this prediction. Equations $(1.45)_{4,5,6}$ are the forward Cauchy-Euler integrations steps of equation

(1.34). The use of the predictor $\tilde{\mathbf{z}}_{1,k+1}$ mainly affects the function $\mathbf{g}$ in $(1.45)_5$ since it contains the stiffest part of the equations of motion (i.e., the intrinsic director forces).

Equations $(1.45)_{2,3}$ compute approximations to the Lagrange multipliers of equation (1.35) 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 $(1.45)_{2,3}$ whenever contact is initiated.

We close this chapter with a remark concerning Δt and the transitions to and from equations (1.27) and equations (1.34) 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 to a smaller value in order to keep the penetration of the vehicles small[8]. 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 time interval (say $100\Delta t$), the step size is increased again back to its original value.

[8]See the Programmer's Guide for further details.

Chapter 2

User's Manual

2.1 Introduction

MEDUSA simulates arbitrarily many vehicles driving on a horizontal road without obstacles. The program allows vehicular collisions with moderately high relative velocities. MEDUSA can be used to qualitatively investigate collision scenarios that occur within platoons of vehicles. In this chapter, we explain with examples how to use MEDUSA to simulate vehicle platoons. 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 3. There, possible modifications of the mathematical models of the vehicles and the road are discussed.

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 mathematical model. Thus, if we say that two vehicles have different models, then we mean that their model parameters are different. Their mathematical model is however the same since it is a standard part of the program code.

The usage of MEDUSA is quite simple. The user provides the vehicle models and the initial positions, orientations and velocities of the vehicles in two separate files. These two input files are written in a plain text format whose contents are explained in Sections 2.2 and 2.3, respectively. 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 like `-t1.0` in the previous example. These options are explained in Section 2.4. MEDUSA writes the simulation data into a plain text file that can be read and graphically presented by other programs such as MATLAB, MATHEMATICA or SMARTPATH. We discuss this in Section 2.5. Finally, in Section 2.6, we outline a procedure to obtain the equilibrium state of a vehicle in which all of the vibrations of the suspensions and chassis have damped out: we will explain in Section 2.2 why this is important.

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

2.2 The File *model.dat*

The user must edit the file *model.dat* which is in plain text format. This file provides MEDUSA with a database of vehicle models. The models in this file are listed sequentially and are numbered starting from one. In the next section, when we define a platoon of vehicles, we will use these numbers to define a model for each vehicle. The advantage of having such a database is that the model of a vehicle can be changed simply by using a different model number from the database. Also, if all vehicles in a platoon are identical, then the file *model.dat* need only contain one single set of model parameters.

The structure of the file *model.dat* is rigorously defined as a sequence of mandatory keywords, or tokens, some of which are followed by a number. If keywords are missing or misspelled, a run-time error message is generated. The keywords are listed in their proper order below. A sample model file containing two vehicle models is printed in Appendix C.1 for reference. Comments can be put 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 denote a keyword and small letters to denote a number that has to be substituted for it.

`NUMBER_OF_MODELS m` This keyword appears only once at the top of the file. MEDUSA will subsequently try to read `m` vehicle models from this file. The sample in Appendix C.1 contains two models. If we set `m` equal to `3` there, an error message is generated.

`MODEL m` Every model starts with this keyword. The symbol `m` stands for the model number. The first model in the file has number `1`, the second `2` and so on. This improves the readability of the file, but an error occurs if this sequence is not met.

For each model, the data is grouped for the Cosserat point, the suspension, the tyre 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 `x` $[kg]$ of the vehicle.

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

`E e NU u VOLUME v` The material properties are defined by Young's modulus `e` $[N/m^2]$, Poisson's ratio `u` and the material volume `v` 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` The coordinates `u`, `v`, `x`, `y` and `z` in $[m]$ are defined in Figure 1.1.

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

`C1 x C2 y` The spring constants for the front and rear wheel suspensions in $[N/m]$.

`D1 x D2 y` These are the corresponding viscous damping coefficients in $[Ns/m]$.

`TYRE x` This is presently the only parameter of the tyre model that can be changed. x $[s]$ is the tyre lag parameter and corresponds to the parameter τ in equation (1.20).

`CONTACT A1 x A2 y A3 z` This data group lists the three semi-axes `x`, `y` and `z` $[m]$ of the ellipsoid that is used to approximate the vehicle's outer geometry. This data is used to determine if two vehicles are in contact with each other.

`EQUILIBRIUM` This data group defines a state of equilibrium of the vehicle. The initial conditions of the vehicle will later be defined relative to this equilibrium.

`R3 r` The height `r` of the vehicle's center of mass above ground in $[m]$.

`D11 x D12 y D13 z` The components of director 1 at the equilibrium [.].

`D21 x D22 y D23 z` The components of director 2 at the equilibrium [.].

`D31 x D32 y D33 z` The components of director 3 at the equilibrium [.].

Additional models are added the same way starting with the keyword `MODEL` and then following the above sequence of keywords. If the equilibrium state is not known, it can be obtained using the procedure outlined in Section 2.6.

2.3 The Platoon Description File

Apart from the file *model.dat*, the user has to provide a second file containing a description of the vehicle platoon. We assume that this file has the default name *platoon.dat*. A different filename may be specified within the restrictions of the operating system when running the program. This is explained in Section 2.4.

The structure of the file *platoon.dat* is similar to that of *model.dat*. There is a mandatory sequence of keywords and, as before, comments may be added starting with the symbol **%**. The keywords may be typed in upper or lower case. A sample of *platoon.dat* can be found in Appendix C.2.

`NUMBER_OF_VEHICLES m` This keyword appears only once at the top of the file. MEDUSA will subsequently try to read the definitions for `m` vehicles from this file. The sample in Appendix C.2 defines two vehicles. If we set `m` to `3` there without adding an additional definition block, a run-time error message is generated.

`VEHICLE_HAS_MODEL n` This keyword starts the definition of a vehicle. `n` is the number of a vehicle model in the database *model.dat* that MEDUSA will use to simulate the vehicle. Note that unlike the models in *model.dat*, the vehicles here are not explicitly numbered. MEDUSA will however assign the first vehicle in this file the number 1 and so forth. This is the same numbering that is used in the output file. For details, see Section 2.5.

`INITIALLY_WITH` This keyword starts the block of initial conditions for this vehicle:

`X x Y y` The initial position of the vehicle's center of mass on the plane in $[m]$.

`ORIENTATION u` This defines in $[rad]$ the initial direction in which the vehicle is heading. The angle is measured about $\mathbf{E}_3$ counter-clockwise from $\mathbf{E}_1$. The value $\frac{\pi}{2}$ would therefore correspond to a heading in the $\mathbf{E}_2$ direction.

`SPEED v` This is the initial speed of the vehicle in $[m/s]$.

`STEERING x` The present version of MEDUSA has a trivial steering model. The steering angle `x` [rad] is held fixed for the whole simulation. The value `0` makes the vehicle drive straight. A positive value makes the vehicle turn to the left, as viewed from the driver's perspective.

2.4 Running the Program

After one has written the model database file *model.dat* and the platoon description file (e.g., *platoon.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:
  -dx.xxx  set fixed stepsize to x.xxx [s] (default: 0.5e-4)
  -e  include total energy in output
  -ffile  parameter file (default: platoon.dat)
  -Ffile  output file (default: data.asc)
  -h  prints this list
  -sx.xx  save data point every x.xx [s] (default: 0.01)
  -tx.xx  simulation ends at x.xx [s] (mandatory)
  -v  include velocities in output
```

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

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 is however order-insensitive. Numbers are indicated with a `x.xx` above. They can be written in the normal floating point format, e.g., `0.015`, `.15e-1` or `1.5E-2` are all valid entries.

`-dx` In the present version, MEDUSA uses a time integrator whose constant stepsize is specified by `x`. Without the `-d` option, the value of the stepsize is the default indicated

by the `-h` option. Note however that at times when vehicles are in contact with each other, the stepsize is reduced to a value that has been pre-programed into MEDUSA.

`-e` This option makes MEDUSA store the total energy of the individual vehicles in the output file. In collision dynamics, it is often a variable of interest.

`-fname` The platoon description is read from *platoon.dat* by default. However, when this option is used, it is read from the file `name`.

`-Fname` By default, the simulation data is written to the file *data.asc*. This option redirects the output to the file `name`. If this file already exists, its contents will be erased.

`-h` This prints the help screen above.

`-sx` Data points are stored in time intervals `x`. If this option is missing, a default value will be used. This value is displayed by the `-h` option.

`-tx` The simulation starts at time zero and ends after `x` seconds. MEDUSA will not run without this parameter

`-v` This option makes MEDUSA write the velocities and director velocities of the individual vehicles to the output file.

We now look at a few examples. Assume that we have written the files *model.dat* and *platoon.dat*. One would like to simulate the vehicles for ten seconds with a fixed stepsize of 10^{-5} seconds. One also wishes that the positions and the directors of the vehicles are written to the file *test.out* every 0.2 seconds. This is the command that does this:

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

Assume now that one has named the platoon description file *crash_it*, one wishes to simulate the vehicles for one second with the default integration stepsize. In addition to the positions and directors, one wants the velocities and the total energies to be written to the default output file:

```
>>medusa -fcrash_it -e -t1 -v
```

Finally, suppose that one would like to run the same simulation with different vehicle models. For this, one has written the file *new_model.dat* according to the guidelines of Section 2.2. How does one run the simulation now? MEDUSA expects the model data in the file *model.dat*. In order to use the new models, one has to change the filename *new_model.dat* to *model.dat*. It is however more efficient to append the new models to the existing ones in *model.dat*. In so doing, remember to properly modify the numbers following the keywords `NUMBER_OF_MODELS` and `MODEL`.

At execution time, MEDUSA displays some information on the screen. If one runs the program using the file *platoon.dat* from Appendix C.2, the screen will look something like this:

```
>>medusa -e -v -t10

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

The columns of the output file data.asc contain the following:
   col. 1: Time
   cols. 2-4: Components of position vector of vehicle 1
   cols. 5-13: Components of the three directors of vehicle 1
   cols. 14-16: Components of position vector of vehicle 2
   cols. 17-25: Components of the three directors of vehicle 2
   cols. 26-28: Components of velocity of vehicle 1
   cols. 29-37: Components of the three director velocities of vehicle 1
   cols. 38-40: Components of velocity of vehicle 2
   cols. 41-49: Components of the three director velocities of vehicle 2
   cols. 50-51: The total energies of each of the 2 vehicles

         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 file in further detail in the next section.

2.5 The Simulation Output File

The simulation output is by default written to the file *data.asc*. A different filename may be specified with the command line option `-F` (see previous section). If the output file is viewed with a screen editor, it will look like this:

```
0.000000e+00  0.000000e+00  0.000000e+00  -3.730000e-02   ...
1.000000e-02  2.444440e-01  0.000000e+00  -3.691894e-02   ...
2.000000e-02  4.888880e-01  0.000000e+00  -3.579733e-02   ...
3.000000e-02  7.333320e-01  0.000000e+00  -3.398085e-02   ...
4.000000e-02  9.777760e-01  0.000000e+00  -3.152424e-02   ...
5.000000e-02  1.222220e+00  0.000000e+00  -2.849029e-02   ...
    ...           ...           ...            ...        ...
```

The ellipses ... indicate that we have truncated the output in the interest of brevity. We see that the file contains a matrix. The numbers are in the standard floating point format. Other software packages, such as MATLAB or MATHEMATICA, can easily read the data from this file.

Every line in the file contains one stored integration step. The first entry on every line (i.e., the first column) contains time. The next three columns contain the three components $\mathbf{r} \cdot \mathbf{E}_i$ $(i = 1, 2, 3)$ of the position vector $\mathbf{r}$ of vehicle 1. This is followed by the three directors of vehicle 1, each with three components (cf. equation (1.25)). If there is a second vehicle in the platoon, the components of its position vector can be found in columns $14 - 16$. This is followed by a total of 9 components for its three directors. This is repeated in this order for every vehicle in the platoon. Also, while executing, MEDUSA displays this information on the screen.

If the `-v` option was specified on the command line, MEDUSA appends additional columns to the ones just described starting with the column number $2 + 12n$, where n is the number of vehicles in the platoon. The first three additional columns contain the components of the velocity of the first vehicle. The next 9 columns contain the components of its three director velocities. Again, this is repeated for every vehicle in the platoon.

If the `-e` option was specified on the command line, MEDUSA appends for each vehicle one column at the end of the line. The first of these columns contains the total energy of vehicle 1, and so forth.

2.6 Adding a New Vehicle Model

A new model is added to the database *model.dat* by adding a data group `MODEL` to the end of the file according to the guidelines in Section 2.2. The `NUMBER_OF_VEHICLES` must be increased by one. This new number also becomes the number of the new model (i.e., the number following the `MODEL` keyword).

In this section, we outline a procedure to obtain the `EQUILIBRIUM` data group of the new appended vehicle model. We will simulate a single vehicle driving by itself in a straight line long enough so that it can reach a relative equilibrium where the oscillations of the suspensions and the chassis have ceased.

First, we write the `EQUILIBRIUM` data group of the new model in *model.dat* as

```
EQUILIBRIUM
  R3 0.0
  D11 1.0  D12 0.0  D13 0.0
  D21 0.0  D22 1.0  D23 0.0
  D31 0.0  D32 0.0  D33 1.0
```

We then create a new file (e.g., *test.dat*) with the following contents:

```
NUMBER_OF_VEHICLES 1

VEHICLE_HAS_MODEL m INITIALLY_WITH  X 0.0  Y 0.0
  ORIENTATION 0.0  SPEED 20  STEERING 0.0
```

The letter `m` above must be replaced by the number of the new model in *model.dat*. The value of `SPEED` is not particularly important for this simulation and can be chosen arbitrarily[2]. MEDUSA is now executed with the command

```
>> medusa -t10 -ftest.dat
```

The simulation time (here ten seconds) should be chosen long enough to allow the vehicle to reach an equilibrium.

The data for `EQUILIBRIUM` can now be read from the last line of the output file *data.asc*: `R3` from column 4 and the directors `D11-D33` from column 5 through 13. Since in this simulation the vehicle was driving in a straight line, one obtains

```
EQUILIBRIUM
  R3 x.x
  D11 x.x  D12 0.0  D13 x.x
  D21 0.0  D22 x.x  D23 0.0
  D31 x.x  D32 0.0  D33 x.x
```

where `x.x` denotes the new equilibrium values.

[2]Note however that the tyre model that is presently used in MEDUSA is not valid for low speeds.

Chapter 3

Programmer's Guide

3.1 Introduction

MEDUSA is a program for the simulation of platoons of vehicles that incorporates moderate vehicle collisions. We briefly reviewed the theory behind MEDUSA in Chapter 1 and explained the use of MEDUSA in Chapter 2. In this chapter, we describe the program code itself[1]. For this we presume that the reader is familiar with the material of the previous chapters. In addition a familiarity with the *ANSI-C* programing language in which MEDUSA is programmed in presumed. The complete source code is listed in Appendix F.

The program consists of several functional blocks which we refer to as modules. These five modules are

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

The modules are, as far as possible, independent of each other. They are contained in separate files. Some general programing ideas are however common to all modules. These ideas are discussed in Section 3.2. In addition, Section 3.3 presents a toolbox of definitions and functions which is accessible from all modules.

As in every *C* program, execution starts with the function `main()`. It is explained in Section 3.4. From there, the initialization module is called which also serves as the input interface to the user and interprets the guidelines of Chapter 2. This module is explained in Section 3.5. Next, the time integration module is executed. We discuss it in Section 3.6.

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

It integrates the equations of motion of the vehicle models in the platoon under consideration. These models are collected in a single module which is explained in Section 3.7. The module that detects collisions is explained in Section 3.8. During integration, intermediate integration steps are written to an output data file. This is explained in Section 3.9.

For details on the program, which are not presented in this chapter, we refer to the source code in Appendix F. Section 3.10 is intended to give additional information for the programmer who wishes to make substantial changes and additions to the code.

3.2 Generalities

We highlight in this section a few ideas and techniques that went into the making of MEDUSA. They are meant both to enhance the readability of the source code and improve its structure. The mathematical models that are presently used for the vehicles are quite crude in places (such as the tyre model). Nevertheless, we attempted to make the code sufficiently general so that later modifications and improvements are possible.

Compatibility

MEDUSA is programed in *ANSI-C* (see Kernighan and Ritchie [7] for a complete reference). It therefore runs on any computer platform supporting this programing language. The only incompatibilities that are known so far are the following:

- MEDUSA The largest memory block that is allocated dynamically is the output buffer (see Section 3.9). If it is chosen too big, it can cause memory allocation errors on platforms with segmented memory, in particular on IBM-compatible platforms.

- The carriage return character `\r` is used in several `printf()` statements in *main.c*. This can have undesired effects on the screen of some platforms where `\r` needs to be replaced by `\n`.

Modularity and Global Variables

Each **.c* file of the program is considered to be a module of the code. This means that when global variables are used, they are only known to functions within that file. These variables are never used for the exchange of data between the modules. This is always done through function arguments which often are pointers to variables. In order to ensure that global variables remain encapsulated in a file, the storage class specifier `static` is always used when declaring them.

Macros

The `#define` statements that define a macro can be found either at the beginning of a source code listing, just in front of the function that uses the macro or in separate header files with

extension **.h*. The macro names are mostly written in capital letters to distinguish them visually from variables and functions. A typical set of definitions found in the listings is

```
#define N 12
#define twoN 24
#define STATES 28
```

Here, `N` stands for a total of 12 components of the position vector and the three directors of a Cosserat point. Also, `twoN` includes the components of the velocities and director velocities. `STATES` is the length of the state vector of a vehicle. In this case, it is the `twoN` plus four states that are used for the tyre models (see Section 3.7). These definitions can simplify later modifications when more complex Cosserat points and additional states are being simulated.

Data Types for Numerics

Almost all floating point variables in this program are in double precision. However, in connection with the adding of forces that act on a vehicle, we prefer to use the data type `long double` in a few instances. The reason for this is the following: if the forces algebraically add up to zero, the numerical sum may be of the order of the machine precision. This error is negligible for all practical purposes. Nevertheless, in code segments where this is an issue, the data type `long double` is used to eliminate this problem. Many platforms do not support this data type. Their compilers generate warning messages (which may be ignored) and use double precision for all variables instead.

For the storage of vectors and matrices and for use in vector and matrix operations, we use the data types `Vector` and `Matrix`. They are based on ideas presented in the *Numerical Recipes in C* [12]. We explain these data types in detail in Section 3.3.2.

Structures and Pointers

Structures and especially pointers to structures are an easy 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*. For reference, they are also listed in Appendix D. Here, we give a brief outline on the use of these structures:

`vehicle_struct` Several of these structures are used to store the parameters of the vehicle models that are defined in the file *model.dat* (cf. Section 2.2). Their contents are initialized at the beginning of a simulation run (cf. Section 3.5) and remain unchanged thereafter. For each vehicle in a platoon, MEDUSA uses a pointer which points to a structure of type `vehicle_struct`. It is very convenient to use these pointers to pass the parameters of a particular vehicle between functions.

Currently, `vehicle_struct` contains substructures, scalar values, vectors and matrices (cf. the note on numerics above). In future versions of MEDUSA, it is planned to include pointers to functions as well. With this, one will be able to vary not only the parameters of a mathematical model, but one will also be able to choose from a variety

of pre-programed, say, tyre models whose functions may look quite different from each other.

`simu_struct` This structure exists only once and subsumes fixed simulation parameters such as the integration step size and the duration of the simulation. In addition, this structure contains vectors, matrices and also arrays of these. They are used to store variable simulation data such as the state vectors of the vehicles and data pertaining to the contact between the vehicles.

We discuss the contents of these structures in detail in Section 3.3.3. Another small structure called `model_struct` is defined in *init.h* and explained in Section 3.5. It is only used during the initialization of the above 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 file dependencies and the function dependencies.

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

The header file *common.h* provides a body of definitions and function prototypes which can be used as tools in all other parts of the MEDUSA program code. All the source code files contain the line

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

to make these definitions available to them. The functions corresponding to the prototypes are coded and commented in the file *common.c* which is listed in Appendix F. Since these functions are adapted versions of similar ones in the *Numerical Recipes in C* [12], we refer to that book for a more detailed discussion of them.

When we explain the source code of MEDUSA in the subsequent sections, we will assume familiarity with the contents of the file *common.h* which we will soon present.

3.3.1 Some Useful Programing Tools

Macros

- These standard definitions are used for flag variables and tests:

```
#define FALSE 0
#define TRUE 1
```

- The number π:

```
#define Pi 3.141592654
```

- This macro calculates the scalar product of two 3-vectors:

```
#define DOT3(x,y)  x[1]*y[1]+x[2]*y[2]+x[3]*y[3]
```

Function Prototypes

- These functions calculate the maximum of two numbers, the minimum of two numbers and the square of a number, respectively:

```
double max(double x, double y);
double min(double x, double y);
double square(double x);
```

- Print an error message `error_text` to the screen and abort the program:

```
void nrerror(char error_text[]);
```

  This is a generic error handler that is taken from *Numerical Recipes in C* [12].

3.3.2 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 for debugging the program code.

In the *ANSI-C* programming language, pointers and one-dimensional arrays are essentially the same (see Kernighan and Ritchie [7]). 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* [12] 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 again. 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 quite probably lead to a runtime error.

Finally, as for double precision vectors, we define similar function prototypes in the file *common.h* for vectors of integers:

```
int *ivector(int n);
void free_ivector(int *);
```

Their use parallels that of `vector()` and `free_vector()`. However, we do not define an extra data type for integer vectors (such as `Vector` above).

Matrices

We adapted the ideas proposed in the *Numerical Recipes in C* [12] to treat matrices. To denote a matrix, we have created the new data type `Matrix`:[2]

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

```
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);
```

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 used in MEDUSA from time to time:

```
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
```

Matrix Operations

The functions that are presented here accept vector and matrices as arguments. One should remember to allocate memory for these variables using the functions `vector()` and `matrix()` discussed above. Note also that as with any dynamically allocated memory, one has to be careful when using the `Vector` and `Matrix` data type. In particular, one should free the allocated memory when these variables are not used anymore. On the other hand, one should obviously check that these variables are not used once the memory that they occupied has been freed.

- Calculate the product $y = Ax$, where A is a n by m matrix, x is a m vector and the result y is a n vector:

```
void matrix_times_vector(Vector y, Matrix A, Vector x, int n, int m);
```

- Calculate the product $Y = AB$, where A is a m by n matrix, B is a n by p matrix and the result Y is a m by p matrix:

  ```
  void matrix_times_matrix(Matrix Y, Matrix A, Matrix B, int m, int n, int p);
  ```

- Solve the system of linear equations $Ax = b$ where the n by n square matrix A and the n-vector b are given:

  ```
  void lin_solve(Matrix A, int n, Vector b);
  ```

  After the function has been executed, the vector `b` will contain the result x. Note that the function will change the contents of `A`!

- The following function calculates the inverse A^{-1} of a non-singular n by n matrix A:

  ```
  void matrix_inverse(Matrix A, int n, Matrix Ainv);
  ```

3.3.3 The Globally Used Data Structure Types

The Data Type `vehicle_struct`

The `vehicle_struct` data structure helps to conveniently manage the model parameters of a vehicle. It is listed in Appendix D.1 for reference. MEDUSA makes extensive use of pointers to structures of this type. As an example, let us define a pointer `*model` as

```
vehicle_struct *model;
```

The expression `model->#` denotes then an element of the structure `vehicle_struct`. The following is a complete list of these elements:

`model->Cosserat_point.#` This substructure contains the parameters of the Cosserat point that is used to model the chassis of the vehicle. The wildcard `#` stands for one of the following (e.g., `model->Cosserat_point.m`):

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

`Matrix I`: The inertia matrix M in equation (1.28) of data type `Matrix`.

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

`double lambda, two_mu`: The Lamé constants λ and 2μ of equation (1.12).

`double half_volume`: Half the volume of the Chassis; $\frac{V}{2}$ in equation (1.12).

`model->suspension.#` This substructure contains the parameters of the suspension model. The wildcard `#` stands for one of the following (e.g., `model->suspension.L1`):

`double L1, L2, half_B, H1, H2`: These are the coordinates $L1$, $L2$, $\frac{B}{2}$, $H1$ and $H2$ which are defined in equation (1.11) and Figure 1.1.

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

`double C1, C2`: The linear stiffnesses of the front and rear suspension springs, respectively, as defined in equation (1.17).

`double D1, D2`: The linear damping coefficients of the front and rear suspensions, respectively, as defined in equation (1.17).

`Matrix infl`: The influence matrix $\mathbf{A}$ in equation (1.28).

`model->tyre.tau_inv`: This is the tyre force lag parameter τ^{-1} in equation (1.20).

`model->contact.semi_axes[i]`: The three semi-axes of the ellipsoidal which approximates the vehicle geometry. The index `i` can be `1`, `2` or `3`.

The Data Type `simu_struct`

The `simu_struct` data structure helps to conveniently manage the parameters and the variable data which are used during the execution of MEDUSA. It is listed in Appendix D.2 for reference. MEDUSA makes extensive use of pointers to structures of these type. As an example, let us define a pointer `*simulation` as

```
simu_struct *simulation;
```

An element `#` of the structure `simu_struct` may now be accessed using `simulation->#`. The following is a complete list of these elements:

`simulation->in_file`: The name of the platoon description file is stored in this character string. By default it is called *platoon.dat*. This default name is preset in the file *init.h* by the macro `DEFAULT_INPUT_FILE` . The name is changed during run-time by the option `-f` (see Section 2.4).

`simulation->out_file`: The name of the output file is stored in this character string. By default it is called *data.asc*. This default name is preset by the `DEFAULT_OUTPUT_FILE` macro in the file *init.h*. The name is changed during run-time by the option `-F` (see Section 2.4).

`simulation->flags.#` These integer flag variables specify what kinds of data will be included in the output file whose name is stored in `simulation->out_file` (see also Section 2.4). The wildcard `#` stands for one of the following:

`int velocity`: Corresponds to the run-time option `-v` (see Section 2.4).

`int energy`: Corresponds to the run-time option `-e` (see Section 2.4).

The corresponding data will be included in the output if a flag is `TRUE`. The default value is `FALSE`.

`simulation->n` The number of vehicles in the simulation. MEDUSA finds this number in the platoon description file (whose name is stored in `simulation->in_file`) after the keyword `NUMBER_OF_VEHICLES` (see also Section 2.3).

`simulation->integrate.#` These parameters control the time integration. Here, the wildcard `#` stands for one of the following (e.g., `simulation->integrate.end_time`):

`double end_time`: The simulation ends after a simulated time of `end_time` seconds. This corresponds to the run-time option `-t` (see Section 2.4).

`double delta_t`: The fixed stepsize of the integrator. The default value is preset by the macro `DELTA_T` in the file *init.h*. The value is changed by the option `-d` in Section 2.4.

`double save_delta_t`: In time intervals of length `save_delta_t`, data points are written to the output file whose name is stored in `simulation->out_file`. The default value is preset by the macro `SAVE_DELTA_T` in the file *init.h*. The value is changed by the run-time option `-s` (see Section 2.4).

`simulation->state_vector`: This is a matrix of the `Matrix` data type. Each row vector corresponds to the state vector $\mathbf{z}$ of one of the vehicles. The vector $\mathbf{z}$ is defined by equation (1.29).

`simulation->steer_angle`: This vector of the `Vector` data type contains the fixed steering angles for each vehicle. These angles are defined in the platoon description file whose name is stored in `simulation->in_file`. MEDUSA reads them from `STEERING` keyword (cf. Section 2.3).

`simulation->constraint.#` These variables pertain to the contact detection and contact force calculation. The wildcard `#` stands for one of the following:

`Matrix multiplier`: For each pair of vehicles, this matrix stores the Lagrange multipliers γ_1 and γ_2 of the contact force equations (1.35).

`Vector **forces`: This is a pointer to a one-dimensional array of `Vector *` pointers. There is one `Vector *` pointer for each vehicle. Such a pointer points to a one-dimensional array of constraint force vectors (having the `Vector` data type).

`Vector **state`: This variable is a two-dimensional array of vectors having the `Vector` data type. There is one such vector for every pair of vehicles corresponding to the vector $\mathbf{x}$ in equation (1.40). These vectors define the most recent contact points for any two vehicles.

3.4 The Function `main()`

ANSI-C starts the execution of the MEDUSA program code with the function `main()`. `main()` does the following: it calls the function `init()` (see Section 3.5), initializes some global variables (cf. Section 3.2) and opens the output file. `main()` also prints some information pertaining to the simulation parameters to the screen for the convenience of the user. Finally, `main()` calls the function `integrate()` (cf. Section 3.6) which performs the actual simulation of the vehicles. The program ends thereafter.

3.5 The Initialization Module *init.c*

When MEDUSA is executed, the principal function `main()` (in the file *main.c*) first calls the function `init()` (in the file *init.c*). This file constitutes a module which performs several tasks. It evaluates the command line options specified by the user (see Section 2.4), it reads the files that have been provided by the user (see Sections 2.2 and 2.3) and it initializes the globally used data structures (see Section 3.3.3).

The initialization works as follows: First, the function `init()` writes some default values into the structure variable `simulation`. These values are defined in the header file *init.h* which supplements *init.c*. Next, the three functions

```
evaluate_cmd_line()
read_models()
read_vehicles()
```

are called in sequence, each of which performs one of the tasks of the initialization module: we will explain them individually below. The function `init()` returns two pointers to the calling function `main()`. The first is a pointer to the variable `simulation`. The other is a one-dimensional array whose elements are pointers to `vehicle_struct` data structures. We will discuss this array below when we describe the function `read_vehicles()`.

3.5.1 General Notes

The global variable[3] `simu_struct simulation` is declared within the initialization module. Various pointers to this variable are used throughout the MEDUSA program code. This module also uses a structure data type called `model_struct` which is listed in the file *init.h*. It is defined as follows:

```
typedef struct {
   vehicle_struct vehicle;
   double r3;
   Matrix F;
```

[3]Recall from Section 3.2 that in our terminology a global variable is known only to those functions within the same module (and file).

```
} model_struct;
```

As an example, let us define the pointer `model_struct* model`; the structure is accessed by `model->#`, where `#` stands for one of the following variables:

`vehicle`: This structure stores the parameters of a vehicle model (see also the discussion in Section 3.3.3).

`r3`: This scalar variable stores the equilibrium value of the vertical component of the position vector of the vehicle. This value is specified by the keyword `R3` in an `EQUILIBRIUM` section of the file *model.dat* (see Section 2.2).

`F`: This matrix corresponds to the deformation gradient **F** defined in equation (1.3). The matrix stores the equilibrium values of the three directors of a vehicle. These values are specified in the `EQUILIBRIUM` sections of the file *model.dat* (see Section 2.2).

To perform the task of reading one of the input files from the disk into a buffer string, the functions `read_models()` and `read_vehicles()` each first call the function `read_file()`. This function also strips the file contents of all the comments and eliminates multiple white-space characters from the buffer string[4]. The use of comments is explained in Section 2.2.

To search a buffer string for a keyword[5] and to read the number following the keyword into a variable, the aforementioned functions make repeatedly use of the function `read_expression()`. If no number needs to be read, the function `find_token()` is used instead. These two functions abort the program with an error messages if the guidelines for writing input files are violated. These guidelines are listed in the User Manual, Sections 2.2 and 2.3.

3.5.2 Evaluation of the Command Line: `evaluate_cmd_line()`

The contents of the command line is passed down from the function `main()` using the standard *ANSI-C* variables `argv` and `argc`. We refer to Kernighan and Ritchie [7] for a discussion of these two variables. The function `evaluate_cmd_line()` uses a `switch ... case` control structure to evaluate these variables according to the guidelines for command line options in Section 2.4. The following table shows which entry of the global variable `simulation` is associated with a given option:

`-e simulation.flags.energy`	`-f simulation.in_file`
`-v simulation.flags.velocity`	`-F simulation.out_file`
`-d simulation.integrate.delta_t`	`-s simulation.integrate.save_delta_t`
`-t simulation.end_time`	

[4] For a discussion of white-space characters, see Kernighan and Ritchie [7].

[5] Keywords are sometimes called tokens in the source code of the program.

3.5.3 Evaluation of the File *model.dat*: `read_models()`

After the function `read_models()` has obtained a buffer string (from `read_file`) containing the condensed contents of the file *model.dat*, this buffer is searched for the keyword `NUMBER_OF_MODELS` and the number of vehicle models is read into a variable. Using `malloc()`, an array of `model_struct` structures is then allocated; one structure for every vehicle model defined in the file *model.dat*. This array is given the name `model`. The expression `model[m]` denotes now the m-th element of this array corresponding to the m-th vehicle model. `model` is also the return value of the function `read_models()`.

According to the guidelines in the User Manual, Section 2.2, for each model m, the buffer string is searched for the proper sequence of keywords together with their values. Some values are written directly to the corresponding variable in the `model[m]` structure. Other values are used to calculate the parameters needed for the vehicle model using the equations of Chapter 1. The following list of keywords gives account of these correspondences. For further reference, see also the details in Section 3.3.3:

`IX, IY, IZ`: Using equations (1.8), (1.9) and (1.28), the following matrices are allocated and then calculated:

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

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

```
model[m].vehicle.Cosserat_point.lambda
model[m].vehicle.Cosserat_point.two_mu
model[m].vehicle.Cosserat_point.half_volume
```

`L1, L2, B, H1, H2, SPRING_REF, C1, C2, D1, D2`: These are read directly into the corresponding variables `model[m].vehicle.suspension.#`. Using equations (1.11) and (1.28), the influence matrix $\mathbf{A}$ is calculated and copied into the variable

```
model[m].vehicle.suspension.infl
```

`TYRE`: The inverse of this value is copied to the variable `model[m].vehicle.tyre.tau_inv`.

`A1, A2, A3`: The values of `Ai` are read into the variables

```
model[m].vehicle.contact.semi_axes[i]
```

`R3`: This is the value of `model[m].r3`.

`D11, ..., D33`: The matrix `model[m].F` is allocated and calculated using equation (1.3).

3.5.4 Reading the File Containing the Platoon Data: `read_vehicles()`

First, the condensed contents of the file which is specified by the string `simulation.in_file` is read into a buffer string by `read_file()`. Next, the buffer is searched for the keyword `NUMBER_OF_VEHICLES` and the number of vehicles is stored in the variable `simulation.n`.

Now that we know how many vehicles the program needs to simulate, we can allocate memory for various variables and initialize them. The initialization is explained in detail after the following list of the relevant variables:

`vehicle_struct **vehicle`: This is a one-dimensional array of `vehicle_struct *` pointers. There is one array element for each vehicle in the simulation. The value of `vehicle` is returned to the function `init()` which passes it on to `main()`.

`simulation.state_vector, simulation.steer_angle`: Memory is allocated.

`simulation.constraint.multiplier`: The elements of this matrix are set to zero.

`simulation.constraint.state`: This variable can be regarded as a two-dimensional array whose components are the 4-vectors $\mathbf{x}$ define in equation(1.40). The components of all vectors are set to $\frac{\pi}{4}$.[6]

`simulation.constraint.forces`: This variable can be regarded as a one-dimensional array whose components are matrices. The elements of these matrices are set to zero (i.e., the vehicle are assumed not to be in contact initially and all the contact forces are therefore zero.)

The initialization now works as follows: For each vehicle v, the buffer string is searched for the keyword `VEHICLE_HAS_MODEL` and the model number is read into the variable `m`. The following line now assigns the proper model data to the vehicle:

```
vehicle[v]=&model[m].vehicle;
```

Recall that the variable `simulation.state_vector[v]` is the state vector of the vehicle v as defined in equation (1.29). For simplicity, let us call it `z` here. The elements `z[i]` need to be initialized with the initial conditions of the simulation:

`z[1], z[2]`: The values for these variables come from the keywords `X` and `Y`, respectively.

`z[3]`: The value of this variable is copied from the variable `model[m].r3`.

`z[4], ..., z[12]`: These variables correspond to the initial deformation gradient $\mathbf{F}_0$ from equation (1.3). It is calculated from the equilibrium deformation gradient $\mathbf{F}_{eq}$ (stored in the matrix `model[m].F`) as follows:

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

where the orientation θ is given by the keyword `ORIENTATION` and $\mathbf{Q}$ is a planar rotation tensor.

[6] There is nothing special about the choice of these initial values. See also Section 1.4.

`z[13]`, `z[14]`: These two variables 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 (keyword `SPEED`) and the angle θ defined above.

`z[15]`, ..., `z[STATES]`: Set to zero.

3.6 Time Integration

The file *main.c* defines the function `integrate()` which calculates the left-hand sides of equations $(1.45)_{1,4,5,6}$ over a period of time at a certain time-rate. These parameters are given by the respective variables (Section 3.5.2):

```
simulation->integrate.end_time
simulation->integrate.delta_t
```

The function `set_constraint_forces()` supplements hereby the equations $(1.45)_{2,3}$ from which the constraint forces $\mathbf{c}_1$ and $\mathbf{c}_2$ in the right-hand side of equations $(1.45)_{4,5}$ are calculated (see Section 3.7.1)[7]. The right-hand sides of equations $(1.45)_{4,5,6}$ are supplied by the function `equations_of_motion()` which is discussed in Section 3.7.

To output intermediate integration steps to a file, `integrate()` calls the functions `save_data_point()` and `write_to_file()` which are discussed in Section 3.9.

It was pointed out in Section 1.5 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 3.7.1). 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`[8].

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 and Δt is still smaller than the value given by the `simulation->integrate.delta_t` variable.

3.7 The Vehicle Model: *vehicle.c*

The module contained in the file *vehicle.c* performs three tasks which are contained in three separate principal functions: `set_constraint_forces()` calculates the contact forces (1.35)

[7] Recall from Sections 3.3.3 and 3.5.4 that the variable `simulation->constraint.forces` stores the constraint forces. For brevity, it is substituted by `contact_forces` in the function `integrate()`.

[8] This macro is defined in the source code just before `integrate()`.

that act between the vehicles, `equations_of_motion()` calculates the equations of motion (1.34) for a single vehicle and `energy()` calculates the total energy (1.30) of a single vehicle. The latter function is presently used to calculate additional output of the program. In future modifications however, the total energy may also be used to control adaptive stepsize time integrator schemes.

3.7.1 Contact Forces: `set_constraint_forces()`

The function `set_constraint_forces()` repeatedly calls the function `detect_contact()` (see Section 3.8) to determine if any two vehicles with numbers cv and av ($cv < av$) are in contact. If so, the contact forces acting on the two vehicles are calculated. A non-zero value is then returned which is used in the function `integrate()` (see Section 3.6).

A few variable substitutions are made in the source code for brevity and readability: `z1` and `z2` denote the state vectors $\mathbf{z}_{(\mathtt{cv})}$ and $\mathbf{z}_{(\mathtt{av})}$ of the two vehicles as defined in equation (1.29)[9]. The matrices `F1` and `F2` are the deformation gradients of the two vehicles defined by equation (1.3).

The variable `ctr` is a vector of n integer counters, where n is the number of vehicles. The counters work as follows: Consider the vehicle with number v. Whenever $cv = v$ or $av = v$, a contact force is calculated for the vehicle v and `ctr[v]` is incremented by one. Obviously, when the function `set_constraint_forces()` is finished, `ctr[v]` has the value $n-1$ for all v as there are $n-1$ vehicle pairs containing the vehicle v.

Further substitutions are the variables `gamma1` and `gamma2` which are pointers to the Lagrange multipliers γ_1 and γ_2 of equation (1.35). These are stored in[10]

```
simulation->constraint.multiplier[cv][av]
simulation->constraint.multiplier[av][cv]
```

The vectors `c1` and `c2` are the contact forces defined in equation (1.35). They are stored in

```
simulation->constraint.forces[cv][ctr[cv]]
simulation->constraint.forces[av][ctr[av]]
```

The contact forces are now calculated as follows: If the function `detect_contact()` detects no contact between the vehicles cv and av, the vectors `c1` and `c2` and the Lagrange multipliers `*gamma1` and `*gamma2` are set to zero. If the vehicles are in contact, the constraint function ϕ_1 from equation $(1.32)_1$ is calculated[11]. The coordinates $X^i_{\sigma(\beta)}$ in equation $(1.32)_2$ are calculated using equations (1.2) and (1.3). They are used to calculate ϕ_2 from equation (1.33). `*gamma1` and `*gamma2` are now updated using equations $(1.45)_{2,3}$[12]. The contact forces `c1` and `c2` are finally calculated using equation (1.35).

[9] These are not the vectors $\mathbf{z}_1$ and $\mathbf{z}_2$ of equation (1.26).

[10] Obviously, the elements `simulation->constraint.multiplier[i][i]` are void.

[11] The variables $\mathbf{r}^*_{(1)}$ and $\mathbf{r}^*_{(2)}$ of equation (1.32) correspond to the variables `rho1` and `rho2` in the source code which are position vectors of the contact point with respect to the center of mass of vehicle cv.

[12] The penalties p_1 and p_2 in those equations correspond to two macros which are defined in the source code just before the function `set_constraint_forces()`.

3.7.2 Equations of Motion: `equations_of_motion()`

The function `equations_of_motion()` is really the heart of MEDUSA as it contains the vehicle model outlined in Section 1.1. It is programed in a straight forward manner. The input arguments of the function are the state vector `z` (see equation (1.29)) of a vehicle at time `t`, the model parameters of that vehicle in form of the pointer variable `vehicle_struct *model` (cf. Section 3.5.4), the array of contact forces (see Section 3.7.1) and the (constant) steering angle of the vehicle. The output argument is the time derivative `Vector dzdt` of the state vector `Vector z`.

To simplify matters and for brevity, the following global variables are declared to 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):

```
Vector r, d1, d2, d3, v, w1, w2, w3;
```

The function `equations_of_motion()` assigns to them the values (cf., equations (1.29) and (1.26))

```
r=&z[0], d1=&z[3], d2=&z[6], d3=&z[9];
v=&z[12],w1=&z[15],w2=&z[18],w3=&z[21];
```

Using these global variables, the forces that act on the Cosserat point are calculated by the following functions which take the pointer `vehicle_struct *model` as argument:

`constitutive_equations()` calculates the intrinsic forces $\mathbf{k}$ in equation (1.27) using equations (1.12) and $(1.26)_4$.

`steering_model()` calculates the wheel headings $\mathbf{h}_q$ using equations (1.14) and (1.15)[13].

`applied_forces()` calculates the force vector $\mathbf{f}$ from equation (1.27) using equations (1.16) and $(1.26)_5$. The functions `suspension_model()` and `wheel_lateral_force()` are hereby used to supplement the suspension forces F^q_{susp} (cf. equation (1.17)) and the tyre forces F^q_{lat} (cf. equations (1.21)–(1.24)).

The time derivative `Vector dzdt` of the state vector `Vector z` is now calculated using equations (1.29), (1.27) and (1.34). We note that this calculation involves the multiplication of a matrix with a vector. The dummy variable `long double dummy` is declared and used to store the intermediate results of this linear operation[14].

3.7.3 Energy: `energy()`

The function `energy()` calculates the total energy E of a single vehicle using equations (1.30) and (1.31). The input arguments for this function is the state vector $\mathbf{z}$ of that vehicle (defined by equation (1.29)) and the pointer to its `vehicle_struct` structure. A dummy variable `long double dummy` is used here to reduce the numerical error when calculating the kinetic energy T of the vehicle as it involves a matrix multiplication.

[13] In the source code, the vectors $\mathbf{h}_1 = \mathbf{h}_2$ are called `front` and the vectors $\mathbf{h}_3 = \mathbf{h}_4$ are called `rear`.

[14] See also the notes on numerics in Section 3.2.

3.8 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 1.4. This module returns the contact information needed by the function `set_constraint_forces()`(see Section 3.7.1).

3.8.1 Contact Detection: `detect_contact()`

The function `detect_contact()` 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)[15].
- `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 **x** defined in equation (1.40). 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.
- `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

- `pos()` calculates the position vector of the contact point by a equation (1.4) in Section 1.2.1.
- `norm()` calculates the unit outward normals at the contact points using equation (1.39) which is discussed in Section 1.4.1.
- `dot()` computes the dot product of two 3-vectors and returns a scalar.
- `dist()` provides the value of the distance function at the coordinates given by the vector variable `state`.

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

- `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()`.

- `minimize()` is the most significant function in this module. This function adopts the *variable metric* method (see Appendix A) to find the local minimum of a scalar-valued function with vector variables. The convergence requirement on zeroing the gradient is defined as GTOL and the difference between new and current points needs to be less than TOLX.

- `linesarch()` predicts the next acceptable point along the descent direction computed by the variable metric method. This function stops if either new point is too close to the current point or equation (B.3) has been satisfied (cf. Appendix B).

3.8.2 Unique Contact Point Detection: `pert()`

- `func()` calculates the value of the function $\hat{f}_{(\beta)}$ (cf. equation 1.44).

- `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 the values of the function $\hat{f}_{(\beta)}$ (cf. 1.44). 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 1.7). This function converges when the function $\hat{f}_{(2)}$ at these points is close to zero.

3.9 Data Output

The MEDUSA file *main.c* contains a simple algorithm to handle the output of data. Following the guidelines of the User Manual outlined in Section 2.5, the data is first saved to a buffer in time intervals `simulation->integrate.save_delta_t` by the function `save_data_point()`. When the buffer is full or at the end of the simulation, the buffer is written (i.e., flushed) to the output file by the function `write_to_file()`.

The output file is accessed using the file handle `FILE output_file` which is a global variable. When the file `simulation->out_file` is opened for writing by the function `main()`, this file handle is initialized.

The buffer is a global variable of data type `Matrix` which is created initially by the function `main()`. The number of rows is given by the macro `KMAX`. It is the number of time steps that are buffered before they are flushed to the output file. The number of columns is calculated as follows:

- One column for the time parameter t.
- 12 columns for each vehicle to store the position vector and the directors.
- 12 columns for each vehicle to store the velocities and director velocities if the flag `simulation->flags.velocity` is `TRUE`.
- One column for each vehicle to store the total energies of the individual vehicles if the flag `simulation->flags.energy` is `TRUE`.

We refer to Section 2.5 for details on the order in which the above quantities are stored.

The function `write_to_file()` uses the standard *ANSI-C*-function `fprintf()` (see Kernighan and Ritchie [7]) to write the output file. For this reason, all numbers are written using the standard *ASCII* extended floating point format which is applied by *ANSI-C* and can easily be read by other programs.

3.10 Adding User Supplied Code

MEDUSA currently uses a rather restricted mathematical model of a vehicle, as pointed out in Chapter 1. The user may therefore wish to make modifications and improvements to the suspension, tyre and road models, as well as the Cosserat point itself. Also, the time integration and output algorithms leave room for improvements.

It should be evident from the previous sections of this chapter that for each aspect of the vehicle model there is a corresponding function in the source code of MEDUSA. In order to change the mathematical model of, say, the suspension, the user only needs to consider the function `suspension_model()` in the file *vehicle.c* (cf. Section 3.7.2). The same holds for the tyre model and so forth. We refer therefore to the material presented earlier for details how the various functions work.

Quite probably, the modified program code will require additional parameters that ought to be read from the input files (i.e., *model.dat* and *platoon.dat*) or from the command line. We discussed in Section 3.5 how this has been implemented in MEDUSA. Additional parameters are introduced in two steps. First, the structures `vehicle_struct` or `simu_struct` (depending on the problem) must be redefined in order to accommodate the new variables. These structures are defined in the file *common.h* and were explained in Section 3.3.3. Then the functions that read the data from a file must be updated. These functions are all defined in the file *init.c* and were discussed in Sections 3.5.2, 3.5.3 and 3.5.4. The portions

of the source code that presently initialize a given variable can simply be duplicated and then modified to the needs of a new variable. The programmer is now free to use the new parameters like all the others through the data structures and the pointers thereto.

The source code of MEDUSA was conceived and written in such a way that modifications are reasonably simple as long as the present structure of function calls with the corresponding arguments is maintained. Beyond that, it is hard to give general guidelines. We note however that it is planned that future versions of this program will be able to handle program structures with greater flexibly. In particular, the program will be able to handle a variety of distinct mathematical models at once from which the user can choose one using the files *model.dat* and *platoon.dat*. The program will be written in a way that additional mathematical models may be added without the need to change those that are already in the program.

Bibliography

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

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

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

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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. 48, 1997 (to appear).

[12] 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.

[13] 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.

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

[15] 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.

Appendix A

The Variable Metric Method

In Section 1.4, the distance function from an arbitrary point outside the ellipsoid to any point located on the surface of the ellipsoid 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 [3, 2] 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 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 nonsigular matrix with the dimension of $\mathbf{x}$. Thus in the new variable space, the quadratic model around $\hat{\mathbf{x}}_j$ 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 [5]

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

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 $\mathbf{x} = \mathbf{x}_{j+1}$ which is the result by the line searches and backtracking scheme [12] 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 those respects.

Appendix B

Line Search and Backtracking

The strategy for proceeding from a solution estimated outside the convergence region of Newton's method is the method of line searches and backtracking [3]. 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. Therefore, 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 $\lambda = 1$ as the first try. 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 there is too small of a decrease in function values relative to the length of steps. 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 [3] 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 initial rate of decrease[3] in that direction. The second case arises when the steps are too

[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 [3] for the examples of such cases.

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

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

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

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 ψ is minimized. Initially, we have two pieces of information about $\psi(\lambda)$,

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

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 modeled as a cubic in 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.

Appendix C

Sample Parameter Files

C.1 *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   200.0e6  % 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  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]
 TYRE 0.0016    % lag parameter for tyre model [s]
 CONTACT
  A1 1.5       % semi axes of ellipsoidal [m]
  A2 1.0
```

```
  A3 1.0
 EQUILIBRIUM
  R3 -0.0373   % 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]
 TYRE 0.0016    % lag parameter for tyre model [s]
 CONTACT
  A1 1.5       % semi axes of ellipsoidal [m]
  A2 1.0
  A3 1.0
 EQUILIBRIUM
  R3 -0.0373   % 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
```

C.2 *platoon.dat*

The following is a printout of a sample file *platoon.dat*. It describes a platoon consisting of two vehicles which travel on n a straight line. Initially, their mass centers are 5 meters apart. The vehicle in the front is slower that the one following it. The vehicles are bound to collide.

```
%
% Sample initialization of a two vehicle platoon with collision
%
NUMBER_OF_VEHICLES 2

% 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 [.]
  SPEED 24.4444       % forward speed [m/s]
  STEERING 0.0        % steer angle [rad]

% Vehicle 2
VEHICLE_HAS_MODEL 1 INITIALLY_WITH
  X 5.0               % X coordinate of center of mass [m]
  Y 0.0               % Y coordinate of center of mass [m]
  ORIENTATION 0.0     % heading angle [.]
  SPEED 23.1111       % forward speed [m/s]
  STEERING 0.0        % steer angle [rad]
```

Appendix D

Structure Definitions

D.1 The Vehicle Model Structure

For each vehicle model that is specified in the file *model.dat*, MEDUSA initializes a structure of the data type `vehicle_struct` which contains the model parameters. This data type is listed below:

```
typedef struct {
  struct {
    double m;        /* mass of car in kg */
    Matrix I;        /* inertia matrix */
    Matrix I_inv;    /* inverse inertia matrix */
    double lambda, two_mu;   /* elastic properties of Cosserat point */
    double half_volume;      /* half volume of the Chassis in m^3 */
  } Cosserat_point;
  struct {
    double L1, L2;   /* distance from cg to front/rear axle in m */
    double half_B;   /* half the track of axle in m */
    double H1, H2;   /* vert. dist. from cg to front/rear assembly pts. */
    double spring_ref;  /* reference length of suspension spring in m */
    double C1, C2;   /* spring constant for front/rear suspension in N/m */
    double D1, D2;   /* damping coeff. for front/rear suspension Ns/m */
    Matrix infl;     /* Influence matrix */
  } suspension;
  struct {
    double tau_inv;  /* lag parameter for tyre model in 1/s */
  } tyre;
  struct {
    double semi_axes[4];  /* semi axes of ellipsoidal */
  } contact;
} 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;           /* default is platoon.dat */
  char *out_file;          /* default is data.asc */
  struct {                 /* defines what will be included in output data */
    int velocity, energy;
  } flags;
  int n;                   /* number of vehicles */
  struct {
    double end_time;       /* simulation runs for end_time seconds */
    double delta_t;        /* integration stepsize */
    double save_delta_t;   /* time intervals for saving a data point */
  } integrate;
  Vector *state_vector;    /* array of state vectors */
  Vector steer_angle;
  struct {
    Matrix multiplier;     /* The Lagrange multiplier are in a n by n matrix */
    Vector **forces;       /* array of pointers to arrays of constraint forces */
    Vector **state;        /* array of vectors holds previous contact points */
  }constraint;
} simu_struct;
```

Appendix E

Dependencies

E.1 File Dependencies

- *contact.c* $\longrightarrow$ *common.h*
- *common.c* $\longrightarrow$ *common.h*
- *init.c* $\longrightarrow$ *init.h*, *common.h*
- *main.c* $\longrightarrow$ *common.h*
- *vehicle.c* $\longrightarrow$ *common.h*

E.2 Function Dependencies

- `applied_forces()` ..... *vehicle.c*
  - `suspension_model()` ..... *vehicle.c*
  - `wheel_lateral_force()` ..... *vehicle.c*
- `cmderror()` ..... *init.c*
- `constitutive_equations()` ..... *vehicle.c*
- `detect_contact()` ..... *contact.c*
  - `pos()` ..... *contact.c*
  - `norm()` ..... *contact.c*
  - `dot()` ..... *contact.c*
  - `dist()` ..... *contact.c*
  - `d_dist1()` ..... *contact.c*
  - `d_dist2()` ..... *contact.c*
  - `d_dist()` ..... *contact.c*

- minimize() ..... *contact.c*
- pert() ..... *contact.c*

- energy() ..... *vehicle.c*
- equations_of_motion() ..... *vehicle.c*
  - applied_forces() ..... *vehicle.c*
  - constitutive_equations() ..... *vehicle.c*
  - steering_model() ..... *vehicle.c*
- evaluate_cmd_line() ..... *init.c*
  - cmderror() ..... *init.c*
  - print_options() ..... *init.c*
- find_token() ..... *init.c*
- g() ..... *vehicle.c*
- init() ..... *init.c*
  - evaluate_cmd_line() ..... *init.c*
  - read_models() ..... *init.c*
- integrate() ..... *main.c*
  - equations_of_motion() ..... *vehicle.c*
  - free_matrix() ..... *common.h*
  - matrix() ..... *common.h*
  - save_data_point() ..... *main.c*
  - set_constraint_forces() ..... *vehicle.c*
  - write_to_file() ..... *main.c*
- main() ..... *main.c*
  - free_matrix() ..... *common.h*
  - init() ..... *init.c*
  - integrate() ..... *main.c*
  - matrix() ..... *common.h*
- minimize() ..... *contact.c*
  - linsearch() ..... *contact.c*
- print_options() ..... *init.c*
- pert() ..... *contact.c*
  - func() ..... *contact.c*
  - piksrt() ..... *contact.c*
  - opp() ..... *contact.c*

- read_expression() ..... *init.c*
  - find_token() ..... *init.c*
- read_file() ..... *init.c*
  - nrerror() ..... *common.h*
- read_models() ..... *init.c*
  - find_token() ..... *init.c*
  - matrix() ..... *common.h*
  - matrix_inverse() ..... *common.h*
  - nrerror() ..... *common.h*
  - read_expression() ..... *init.c*
  - read_file() ..... *init.c*
- read_vehicles() ..... *init.c*
  - find_token() ..... *init.c*
  - free_matrix() ..... *common.h*
  - matrix() ..... *common.h*
  - nrerror() ..... *common.h*
  - read_expression() ..... *init.c*
  - read_file() ..... *init.c*
- save_data_point() ..... *main.c*
  - energy() ..... *vehicle.c*
  - write_to_file() ..... *main.c*
- set_constraint_forces() ..... *vehicle.c*
  - detect_contact() ..... *contact.c*
  - free_ivector() ..... *common.h*
  - free_matrix() ..... *common.h*
  - ivector() ..... *common.h*
  - lin_solve() ..... *common.h*
  - matrix() ..... *common.h*
- steering_model() ..... *vehicle.c*
- suspension_model() ..... *vehicle.c*
- wheel_lateral_force() ..... *vehicle.c*
  - g() ..... *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	5 pages
init.h	1 page
init.c	8 pages
vehicle.c	8 pages
contact.c	14 pages
common.c	5 pages