



TNF

Technisch-Naturwissenschaftliche  
Fakultät

# Fluid-Structure Interaction – Coupling of flexible multibody dynamics with particle-based fluid mechanics

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Technische Physik

Eingereicht von:

Markus Schörghumer

Angefertigt am:

Institute of Technical Mechanics

in Kooperation mit dem Institute of Fluid Mechanics and Heat Transfer  
und dem Austrian Center of Competence in Mechatronics, sowie dem Linz Center of Mechatronics

Beurteilung:

Priv.-Doz. Dipl.-Ing. Dr. Johannes Gerstmayr (Betreuung)

O. Univ.-Prof. Dr. Urbaan M. Titulaer

Mitwirkung:

Dipl.-Ing. Dr. Peter Gruber

Dipl.-Ing. Dr. Stefan Pirker

Dipl.-Ing. Andreas Aigner

Linz, 7, 2012



# Contents

<b>Acknowledgements</b>	<b>4</b>
<b>Abstract</b>	<b>5</b>
<b>Symbols and notation</b>	<b>6</b>
<b>1 Introduction</b>	<b>8</b>
1.1 The problem of fluid-structure interaction . . . . .	8
1.2 Motivation and objectives . . . . .	9
<b>2 MBD and HOTINT</b>	<b>11</b>
2.1 A short course on multibody dynamics . . . . .	11
2.1.1 Definition . . . . .	11
2.1.2 Mathematical formulation . . . . .	12
2.1.3 Numerical point of view . . . . .	14
2.1.3.1 Spatial discretization and the transformation of the system of PDEs into a system of ODEs . . . . .	14
2.1.3.2 Constraints . . . . .	15
2.1.3.3 Time integration . . . . .	16
2.1.3.4 Notes on the numerical solution of linear and non-linear al- gebraic equations . . . . .	21
2.2 HOTINT . . . . .	25
2.2.1 What is HOTINT? . . . . .	25
2.2.2 Problem definition . . . . .	25
2.2.3 Notes on the implementation . . . . .	26
<b>3 SPH and LIGGGHTS</b>	<b>29</b>
3.1 Particle-based methods in fluid mechanics . . . . .	29
3.1.1 Introduction . . . . .	29
3.1.2 Smoothed particle hydrodynamics (SPH) . . . . .	30
3.1.2.1 Integral representation of a function and its derivatives . .	30
3.1.2.2 Particle approximation . . . . .	32
3.1.2.3 Notes on the smoothing kernel . . . . .	33
3.1.2.4 The SPH formulation for the Navier-Stokes equations . . .	37

3.1.2.5	Boundary conditions . . . . .	41
3.1.2.6	2D versus 3D . . . . .	42
3.1.3	Numerical point of view . . . . .	43
3.2	LIGGGHTS . . . . .	44
3.2.1	What is LIGGGHTS? . . . . .	44
3.2.2	Problem definition . . . . .	44
3.2.3	Notes on the implementation . . . . .	45
<b>4</b>	<b>Fluid-Structure Interaction – Direct coupling of flexible MBD and SPH</b>	<b>48</b>
4.1	Coupling concept – the main idea . . . . .	48
4.1.1	Introduction . . . . .	48
4.1.2	A non-monolithic approach via TCP/IP . . . . .	49
4.2	The contact formalism – SPH-wall interaction . . . . .	50
4.3	Discretized force calculation and mechanical equilibrium . . . . .	54
4.4	Interface and synchronization . . . . .	60
<b>5</b>	<b>Coupling of HOTINT MBD and LIGGGHTS SPH – Implementation</b>	<b>64</b>
5.1	Introduction and overview . . . . .	64
5.2	Interface . . . . .	67
5.3	Exchange of double-precision numbers . . . . .	69
5.3.1	Conversion and platform-/architecture independent exchange . . . . .	69
5.3.2	Notes on the performance . . . . .	71
5.4	HOTINT / Windows - sided implementation . . . . .	74
5.4.1	exchange_class_windows . . . . .	74
5.4.2	fsi_communication_element . . . . .	76
5.5	LIGGGHTS / LINUX - sided implementation . . . . .	84
5.5.1	exchange_class_linux . . . . .	84
5.5.2	fix_FSI_SPH . . . . .	86
5.5.3	wrapper code LINUX . . . . .	91
5.6	Initialization and the coupled program flow . . . . .	92
<b>6</b>	<b>Example problems and simulations</b>	<b>101</b>
6.1	Introduction . . . . .	101
6.2	Details on the configuration - parameters and stability . . . . .	102
6.3	Volume, density, and weight - a static consistency test . . . . .	104
6.4	Laminar flow around a cylinder . . . . .	110
6.5	Pump-driven channel flow with two valves . . . . .	116
<b>7</b>	<b>Outlook and conclusions</b>	<b>121</b>
7.1	Outlook . . . . .	121
7.2	Conclusions . . . . .	127

<b>Declaration of authenticity</b>		<b>130</b>
<b>Bibliography</b>		<b>131</b>
<b>A C++ source code</b>		<b>135</b>
A.1	interface_baseclass . . . . .	135
A.2	dn.h . . . . .	137
A.3	dn.cpp . . . . .	138
A.4	exchange_class_windows.h . . . . .	145
A.5	exchange_class_windows.cpp . . . . .	147
A.6	fsi_communication_element.h . . . . .	165
A.7	fsi_communication_element.cpp . . . . .	173
A.8	exchange_class_linux.h . . . . .	200
A.9	exchange_class_linux.cpp . . . . .	203
A.10	fix_FSI_SPH_v2_1.h . . . . .	218
A.11	fix_FSI_SPH_v2_1.cpp . . . . .	222
A.12	wrapper code LINUX.cpp . . . . .	245

# Acknowledgements

Hereby, I want to say thanks to all the people who have contributed to this master thesis and supported me all along the way. In particular, I want to mention Peter Gruber who patiently always had (and still has) an open ear for any question or problem whatsoever, and my supervisor Johannes Gerstmayr for his guidance and great motivation. Amongst the members of the Institute of Fluid Mechanics and Heat Transfer, I want to thank first of all Andreas Aigner who was my go-to person whenever I needed help with LIGGGHTS and his SPH implementation, and Stefan Pirker for his open mind.

The funding of my work by the “Austrian Center of Competence in Mechatronics (ACCM)” is most appreciated, as is the cooperation with the “Linz Center of Mechatronics (LCM)”.

# Abstract

Fluid-structure interaction (FSI), as one of the most important representatives of the field of the so-called “multi-physics problems”, is concerned with any kind of static, moving, and/or deforming structural/solid components in contact with a fluid. It includes a vast variety of problems, ranging from large-scale examples such as dynamic instabilities in structural engineering (e.g. a bridge subject to a strong wind) or the mechanical strain and deformation in aircraft wings or turbine blades, to applications in biomechanics and medicine (e.g. the blood stream through thin flexible vessels). As it is the case with many complex physical problems, here, too, apart from experiments, numerical modelling and simulation are essential for the investigation of the system under consideration. Consequently, much effort has been expended on the development and application of numerical methods and computational approaches, however, despite the high attention there still is a lack of established methods which are capable of dealing with a general problem of FSI, offering accuracy, robustness, as well as efficiency.

In the present work an unconventional approach to FSI is developed and implemented, based on the direct coupling of flexible multibody system dynamics (MBD) on the structural side with smoothed particle hydrodynamics (SPH) for the representation of the fluid. In contrast to classical advanced approaches [1], it offers both a very accurate and sophisticated modelling of the systems structural components, and an efficient model on the fluid side, capable of dealing with arbitrary and arbitrarily moving and/or deforming boundaries/geometry as well as free surface flows. At that, the whole implementation is based on the coupling of two already existing simulators – HOTINT for the flexible multibody system and LIGGGHTS for the SPH fluid dynamics.

After the discussion of the underlying theory on each side and a short documentation of HOTINT and LIGGGHTS a suitable contact formalism for the interaction between structure and fluid is introduced, followed by the development of a coupling scheme and interface for the two simulators. Then, an overview of the actual implementation and the coupled program flow is presented, and finally, the work is concluded by the application of the approach to various test examples for investigation of consistency, stability, computational performance, and quantitative verification.

# Symbols and notation

- $x$  ..... scalar quantity
- $\mathbf{r}$  ..... vector quantity (column vector)
- $r_i = (\mathbf{r})_i$  .....  $i$ -th Cartesian component of vector  $\mathbf{r} = r_i \mathbf{e}_i$
- $|q|, |\mathbf{v}|$  ..... absolute value of the scalar  $q$ ,  $l^2$ -norm of the vector  $\mathbf{v}$
- $\mathbf{M}$  ..... second-rank tensor quantity
- $M_{ij}$  ..... component  $(i, j)$  of the matrix representation of a second-rank tensor  $\mathbf{M}$  in Cartesian coordinates
- $\mathbf{M}^\top$  ..... transpose of a matrix  $\mathbf{M}$
- $\mathbf{M}^{-1}$  ..... inverse of a matrix  $\mathbf{M}$
- $\mathbf{I} = \mathbf{I}^\top$  ..... unity tensor  $\mathbf{I}$
- $\mathbf{a} \cdot \mathbf{b}$  ..... dot product of the vectors  $\mathbf{a}$  and  $\mathbf{b}$
- $\mathbf{a} \times \mathbf{b}$  ..... vector cross product of the vectors  $\mathbf{a}$  and  $\mathbf{b}$
- $\mathbf{a} \circ \mathbf{b}$  ..... dyadic product of the vectors  $\mathbf{a}$  and  $\mathbf{b}$
- $\mathbf{M}\mathbf{a}$  ..... matrix-vector product of the matrix  $\mathbf{M}$  and the vector  $\mathbf{a}$
- $\frac{d^i}{dq^i} f, \frac{\partial^i}{\partial q^i} f$  .....  $i$ -th total and partial derivative of  $f$  with respect to a parameter  $q$   
for  $i = 0$  this is equivalent  $f$  itself
- $\frac{d}{dt} q = \dot{q}$  ..... time derivative of a quantity  $q$
- $\nabla_{\mathbf{r}} f = \frac{\partial}{\partial \mathbf{r}} f$  ..... gradient of the scalar function  $f$  with respect to  $\mathbf{r} = r_i \mathbf{e}_i$  in Cartesian  
 $= \mathbf{e}_i \frac{\partial}{\partial r_i} f$  coordinates
- $\frac{\partial \mathbf{g}}{\partial \mathbf{r}} = (\nabla_{\mathbf{r}} \circ \mathbf{g})^\top$  ... gradient of the vector function  $\mathbf{g}$  with respect to  $\mathbf{r} = r_i \mathbf{e}_i$  in Cartesian  
 $\left(\frac{\partial \mathbf{g}}{\partial \mathbf{r}}\right)_{ij} = \frac{\partial g_i}{\partial r_j}$  coordinates (yielding a second-rank tensor)
- $\nabla_{\mathbf{r}} \cdot \mathbf{g} = \frac{\partial}{\partial r_i} g_i$  ..... divergence of  $\mathbf{g}$  with respect to  $\mathbf{r} = r_i \mathbf{e}_i$  in Cartesian coordinates
- $\Delta_{\mathbf{r}} f = \sum_i \frac{\partial^2}{\partial r_i^2} f$  ... laplace operator applied to  $f$  with respect to  $\mathbf{r} = r_i \mathbf{e}_i$  in Cartesian  
coordinates

$A_q f(q, q', \dots)$  . . . . . arbitrary operator  $A$  with respect to  $q$  applied to  $f = f(q, q', \dots)$ ,  
 evaluated at the point  $(q, q', \dots)$

$A_q f(q, \dots)|_{q_0, \dots}$  . . . . . arbitrary operator  $A$  with respect to  $q$  applied to  $f = f(q, \dots)$ ,  
 or  $Af|_{q_0, \dots}$  . . . . . evaluated at the point  $(q_0, \dots)$ ; arguments may be omitted if clear  
 from the context

Furthermore, unless noted otherwise, the Einstein summation convention is used for subscript indices, i.e. when a subscript index variable appears twice in a single term it effectively means the summation of that term over all possible values of that index.

$$a_i b_i = \sum_i a_i b_i = \mathbf{a} \cdot \mathbf{b}$$

# 1. Introduction

## 1.1. The problem of fluid-structure interaction

Along with the steady and substantial increase of computational power in the course of the last three decades, a wide range of computational methods and techniques have been developed in order to address complex problems in various fields of scientific research and applied sciences. Naturally, this development started from solution approaches to specific examples, and eventually led to powerful general methods applicable to all kinds of problems or classes of problems. Considering the latter, the class dealing with problems where more than one physical effect is involved comprises the so-called “multi-physics problems”, among the most important of which is fluid-structure interaction (FSI), challenging with respect to both modelling and computational issues [2].

Fluid-structure interaction is concerned with interactions of some movable and/or deformable elastic structure/solid with an internal or surrounding fluid flow, one of the most relevant and most intensely studied coupled problems; the variety of FSI occurrences is abundant. Despite the high attention there still is a lack of established computational methods which offer accuracy, robustness, efficiency, as well as flexibility, allowing to model and simulate general problems in this inherently multi-disciplinary field [3]. In the approach of developing a method with these capabilities, amongst many other issues the following key questions need to be considered [2]:

- How can the coupling itself – the mechanisms of the actual interaction between the fluid and the structural components – be described appropriately?
- What are the advantages and drawbacks of the various discretization schemes – the numerical mathematical models – used on the flow and on the structure side?
- What are the possibilities and limits of monolithic and non-monolithic, i.e. partitioned or hybrid, coupling schemes?
- How can a flexible data and geometry model look like – especially against the background of large geometric or even topological changes?
- How reliable are the results, and what about error estimation? Which “benchmark examples” can be used for quantitative verification and examination of specific properties?

- What can be said about the design of robust and efficient solvers?
- How can sensitivity and optimization issues enter the game?

## 1.2. Motivation and objectives

Fluid-structure interaction (FSI) plays an important role in the complex field of multi-physics phenomena. In fact, almost any real-life problem in fluid dynamics or mechanics also involves the other respective field as well, however, in many cases a reasonably simplified model can be introduced with the focus being exclusively on one of the two sides, while either incorporating the other one only in simple boundary conditions or neglecting its influence at all. Then again, a variety of problems arise from inherent mechanisms of the interaction between fluid and solid structures, and thus constitute the actual representatives in the field of FSI requiring a fully coupled numerical modelling. Well-known examples would be dynamic instabilities in structural engineering, e.g. a bridge subject to a strong wind, the mechanical strain and deformation in aircraft wings or turbine blades, or the resistance of embankments and barriers to water waves or avalanches.

Many conventional attempts to deal with FSI drastically simplify either the fluid or the structural side (see, for example, [7]), which may lead to insufficient model accuracy. One advanced (fully coupled) classical approach would be the introduction of a particle-based model, such as smoothed particle hydrodynamics (SPH), for the fluid part and some kind of lattice model for the flexible solids part. The advantage here lies in the similar structure and compatibility in the implementation, as well as high computational efficiency and accurate representation of the fluid; the models for the structural side, however, are usually of low convergence order, and may even lead to numerical instability in case of stiff systems. The second classical advanced approach is based on a finite element formulation for the solid structures and a finite volume model for the fluid, which both are the most well established and widely used methods in either respective field, consequently yielding overall accurate solutions. The downside, on the other hand, are the high computational costs, since global remeshing is required in every time step, and difficulties arise particularly with free surface flows, or deformations superimposed to large rigid body motion on the structural side.

By direct coupling of the meshfree particle-based method SPH for the representation of the fluid and advanced methods of flexible multibody dynamics for the description of the solid parts the thesis at hand is an attempt of merging the benefits of both of above discussed approaches. Starting with the very general problem situation and theoretical background from both the solid and the fluid perspective, followed by the introduction and development of a mathematical model, along with an interface definition to allow for simulation and numerical investigation of coupled fluid-structure problems, the main objective was the development and implementation of a stable, flexible and expandable method for the coupling of two already existing simulators – LIGGGHTS for the fluid simulation and HOTINT

for the multibody system dynamics. Conclusively, the approach is applied to benchmark problems and simple problem situations for the investigation and analysis of the whole systems behavior, stability, consistency, as well as quantitative verification.

Of course, there will always be some perspective with regards to what has been discussed in the previous section. However, some questions will be merely touched upon, some even will be left open. Therefore, one should bear in mind that this thesis is not and cannot be intended to make a claim of completeness, neither with respect to the problem of FSI in general, nor the considered specific approach itself.

## 2. MBD and HOTINT

### 2.1. A short course on multibody dynamics

#### 2.1.1. Definition

Multibody dynamics, or, in short, MBD, is concerned with the analysis of the dynamics of so-called “multibody systems” (MBS). At that, a multibody system is a general mechanical and structural system consisting of individual subsystems or components which themselves may be rigid or deformable bodies, interconnected to each other or (kinematically) constrained in whatsoever way. The components may undergo large translational and rotational motion as well as any kind of deformation, induced by external forces, internal stress, contact and constraint forces.

Clearly, the underlying basis to an analysis of dynamics of multibody systems is the understanding of the behavior of the subsystems. The motion of material bodies formed the subject of some of the earliest researches pursued in three different fields, namely, rigid body mechanics, structural mechanics, and continuum mechanics. The term rigid body implies that the deformation of the body under consideration is assumed small such that the body deformation has no effect on the gross body motion. Hence, for a rigid body, the distance between any two of its particles remains constant at all times and all configurations. The motion of a rigid body in space can be completely described by using six generalized coordinates. However, the resulting mathematical model in general is highly nonlinear because of the large body rotation. On the other hand, the term structural mechanics has come into wide use to denote the branch of study in which the deformation is the main concern. Large body rotations are not allowed, thus resulting in inertia-invariant structures. In many applications, however, a large number of elastic coordinates have to be included in the mathematical model in order to accurately describe the body deformation. From the study of these two subjects, rigid body and structural mechanics, the vast field known as continuum mechanics has evolved, wherein the general body motion is considered, resulting in a mathematical model including the difficulties of both of the previous cases, i.e. nonlinearity and large dimensionality [4].

Thus, the equations governing the problems of multibody dynamics usually present themselves as complex non-linear problems without exact analytical solutions; consequently,

the focus here is on the development and implementation of numerical or computational methods.

### 2.1.2. Mathematical formulation

In general we are looking at a system consisting of various bodies with numbers  $i \in \{1, \dots, n_b\}$  and corresponding generalized coordinates  $\mathbf{q}^i$ , which may undergo arbitrary translational and rotational motion as well as elastic or plastic deformation, while satisfying a set of  $n_c$  algebraic constraint equations:

$$C_j(\mathbf{q}^1, \mathbf{q}^2, \dots, \mathbf{q}^{n_b}, t) = 0 \quad j \in \{1, \dots, n_c\}. \quad (2.1)$$

The system of equations of motion can be written down by the use of Lagrange's equations, where  $T^i$  denotes the kinetic energy of body  $i$ ,  $\mathbf{Q}^i$  the vector of the corresponding generalized forces, and  $\lambda_j$ ,  $j \in \{1, \dots, n_c\}$ , Lagrange multipliers to account for the constraint equations [8],

$$\frac{d}{dt} \left( \frac{\partial T^i}{\partial \dot{q}_k^i} \right) - \frac{\partial T^i}{\partial q_k^i} + \lambda_j \frac{\partial C_j}{\partial q_k^i} = Q_k^i \quad j \in \{1, \dots, n_c\} \text{ and } k \in \{1, \dots, n_f^i\}, \quad (2.2)$$

where  $n_f^i$  is the number of generalized coordinates or, equivalently, degrees of freedom (DOF) of body  $i$  without constraints.

In many practical applications, a multibody system can be described with the set of equations (2.1) and (2.2); mathematically, it is a coupled, usually highly nonlinear system of  $N_f$  partial differential equations and  $n_c$  algebraic equations in space and time of in total  $n_c + N_f$  variables, where  $N_f = \sum_i n_f^i$  denotes total number of generalized coordinates of the whole system; the total number of degrees of freedom is, due to (2.1), given by  $N_f - n_c$ , assuming that the constraints are not redundant.

At that, the kinetic energy is defined as

$$T^i = \frac{1}{2} \int_{V^i} \rho^i \dot{\mathbf{r}}^i \cdot \dot{\mathbf{r}}^i dV^i, \quad (2.3)$$

with the mass density  $\rho^i$  and the volume  $V^i$  of body  $i$ , and the global coordinate vector  $\mathbf{r}^i = \mathbf{r}^i(\mathbf{q}^i)$  of a material point  $P$  on the body [8]. The generalized forces  $Q_k^i$  associated with the generalized coordinates  $q_k^i$  can be determined using the principle of virtual work which states that for all admissible virtual displacements  $\delta \mathbf{q}^i$  the total work done  $\delta W_{(c)}$  by the constraint forces  $\mathbf{Q}_{(c)}^i$  vanishes [33],

$$\delta W_{(c)} = \sum_{i=1}^{n_b} \delta W_{(c)}^i = \sum_{i=1}^{n_b} \mathbf{Q}_{(c)}^i \cdot \delta \mathbf{q}^i = 0. \quad (2.4)$$

Hence, the total work  $\delta W$  done on the system for any admissible virtual displacements comprises of internal contributions  $\delta W_{(s)}$  and contributions due to external forces  $\delta W_{(e)}$

[4],

$$\delta W = -\delta W_{(s)} + \delta W_{(e)}, \quad (2.5)$$

which is used to define the generalized forces on each body  $i$  via

$$\begin{aligned} \delta W &= \sum_{i=1}^{n_b} \mathbf{Q}^i \cdot \delta \mathbf{q}^i \\ \mathbf{Q}^i &= \frac{\partial W}{\partial \mathbf{q}^i}. \end{aligned} \quad (2.6)$$

Thus, for the computation of the right-hand side of equation (2.2), formally, the total internal and external virtual work  $\delta W_{(s)}$  and  $\delta W_{(e)}$  needs to be determined. Of course,  $\delta W$  often can be decomposed in independent contributions associated with the individual components of the multibody system and added up subsequently. For instance, considering a virtual displacement  $\delta \mathbf{r}^i$  an external force  $\mathbf{f}_{(e)}^i$  acting in the center of gravity of a rigid body  $i$  would yield the contribution

$$\delta W_{(e)}^i = \mathbf{f}_{(e)}^i \cdot \delta \mathbf{r}^i = \mathbf{f}_{(e)}^i \cdot \frac{\partial \mathbf{r}^i}{\partial \mathbf{q}^i} \delta \mathbf{q}^i = \mathbf{Q}_{(e)}^i \cdot \delta \mathbf{q}^i \quad (2.7)$$

to the external virtual work, and consequently

$$\mathbf{Q}_{(e)}^i = \mathbf{f}_{(e)}^i \cdot \frac{\partial \mathbf{r}^i}{\partial \mathbf{q}^i} \quad (2.8)$$

as corresponding generalized forces. As another example, the internal contributions in case of a deformable body  $j$  can be derived from an integral formulation of the strain energy  $W_{(s)}^j(\mathbf{q}^j)$  which depends on the chosen (tensorial) measure of strain (e.g. Green strain tensor, Almansi strain tensor) and stress (e.g. Cauchy stress, Second Piola-Kirchhoff stress).

For a rigid body the set of generalized coordinates  $\mathbf{q}^i$  consists of three translational and three rotational DOF in 3D (two translational and one rotational DOF in 2D), whereby the former usually are chosen as the coordinates of its center of gravity, and the latter, for instance, may be represented either by three independent Euler angles or four dependent Euler parameters. In case of a deformable object  $\mathbf{q}^i$  typically contains generalized nodal coordinates of some sort of spatial discretization, e.g. finite element formulations which are typically based on Ritz' method with the definition of a spatial interpolation of any field variable  $u^i(\mathbf{r}, t)$  via

$$u^i(\mathbf{r}^i, t) \approx \sum_{k=1}^n N_k^i(\mathbf{r}^i) q_k^i(t) \quad (2.9)$$

with the nodal values  $q_k^i$  and  $n$  space-dependent shape functions  $N_k^i(\mathbf{r}^i)$ ,  $k \in \{1, \dots, n\}$  defined within the domain of the body. One commonly used way to deal with large displacement and rotation as well as deformation is the so-called "floating frame of reference

formulation” (FFRF). Here a moving (“floating”) reference frame is chosen such that there is no rigid body motion between the reference frame and the body itself. Rigid body translation and rotation therefore is defined by the relative motion between this reference frame and the global fixed coordinate system; the deformation part of motion, on the other hand, is described by a set of relative generalized coordinates, the so-called “generalized elastic coordinates”, with respect to the floating frame of reference, e.g. with a finite element interpolation such as (2.9), where  $u^i(\mathbf{r}^i, t)$  would be the displacement field, and  $q_k^i$  the nodal displacements, respectively.

Up to now, we have merely touched the very basics of multibody dynamics; going into detail at this point, however, would go beyond the scope of this thesis. For more detailed information on kinematics, analytical and numerical techniques (cf. also the subsequent sections), mechanics of deformable bodies, the FFRF and finite element formulation, the absolute nodal coordinate formulation (ANCF) [5], and many other issues forming the fundament of MBD, see, for instance, the classic book of Shabana [4], [8, 9], or [6]. Furthermore, it shall only be mentioned that the next things to look at then would be various important formulations and concepts to deal with certain elastic components – structural (finite) elements such as beam, plate, and shell elements – and the respective underlying mathematical theory as well as numerical implementation, which in fact form an important part of the current scientific research in this field, as well as other specific issues, such as contact, friction, or elasticity and plasticity.

### 2.1.3. Numerical point of view

As already discussed in the previous section, the mathematical problem of MBD in general is a system of coupled non-linear partial differential equations (PDE) and algebraic equations due to the constraints. Since it is almost exclusively impossible to find closed analytical solutions, the only option is an approximate solution by means of numerical (computational) methods. At that, the sequence of steps that need to be taken in order to numerically solve a problem such as (2.1) and (2.2) shall be sketched in the following:

#### 2.1.3.1. Spatial discretization and the transformation of the system of PDEs into a system of ODEs

The general approach for the transformation of space- and time-dependent partial differential equations to ordinary differential equations with respect to time resembles the idea of the method of separation of variables. At first, a spatial interpolation for the quantities under consideration is introduced, typically based on a spatial discretization of the problem domain (see, for instance, equation (2.9)). Importantly, for continuous field problems, this represents the transition from a continuous problem, i.e. a problem defined by an infinite

number of generalized DOFs – which, of course, cannot be dealt with numerically – into a numerically feasible discrete system described by a finite set of unknowns.

On that basis, the spatial differential operators then can either be written directly in a discretized form, i.e. substituted by appropriate difference quotients, or the spatial interpolation is inserted into a weak formulation, or some kind of variational or integral formulation of the system of partial differential equations, where the differential operators can be applied directly to the chosen interpolation functions. The former is typically used in finite difference (FD) approaches, the latter is the standard approach in finite element approaches (FE). At this step, also the boundary conditions of the problem need to be considered. After that, separation of time and space, in the sense that at any given time  $t$  the spatial dependency can be treated separately, has been accomplished.

Again, the multitude of computational methods for the spatial solution of partial differential equations, and even the fundamental description of some well-established representatives such as the FD or FE method, lies beyond the scope of the present work. For the finite element method, see, for example, the classic books of Bathe [28], Zienkiewicz and Taylor [29, 30, 31] or [32]; otherwise, the reader is referred to the respective literature.

In either case, any  $k$ -th order PDE is transformed into a system of coupled, in general non-linear  $k$ -th order ODEs with respect to time only, in terms of a set  $\mathbf{q}$  discrete values representing the quantity under consideration either directly as spatially discrete values or by means of an interpolation of some sort. The resulting system can either be written in explicit form,

$$\mathbf{q}^{(k)} = \mathbf{F}(\mathbf{q}, \mathbf{q}^{(1)}, \dots, \mathbf{q}^{(k-1)}, t) \quad \text{with} \quad \mathbf{q}^{(i)} = \frac{d^i}{dt^i} \mathbf{q}, \quad (2.10)$$

or, in the most general case, as implicit system of differential equations:

$$\mathbf{G}(\mathbf{q}, \mathbf{q}^{(1)}, \dots, \mathbf{q}^{(k)}, t) = \mathbf{0} \quad \text{with} \quad \mathbf{q}^{(i)} = \frac{d^i}{dt^i} \mathbf{q}. \quad (2.11)$$

In the case of multibody dynamics, equations (2.10) or (2.11) typically are of order 2 and originate from the discretization of the governing PDE (2.2), where  $\mathbf{q}$  is a set of generalized coordinates which needs to comply with the constraint and boundary conditions. More information on the numerical solution of above two equations follows in Subsection 2.1.3.3.

### 2.1.3.2. Constraints

Constraint equations in MBD impose additional restrictions on the generalized coordinates  $\mathbf{q}$ , reducing the total number of degrees of freedom of the system, and take the form of (in general nonlinear) algebraic equations (cf. equation (2.1)) in terms of  $\mathbf{q}$  and  $t$ . Typically, they are used to describe the mutual connection of two (or more) MBS components by means of joints, or to constrain specific degrees of freedom. Together with the system

of ODEs (2.10) or (2.11) which corresponds to the governing equation (2.2), they form a system of so-called “differential algebraic equations” (DAEs).

A very important characteristic of a DAE is its differentiation index which is defined as the smallest number of differentiations of some of the given equations necessary to get rid of all algebraic equations, i.e. to reduce the system of DAEs to a system of ODEs. Kinematical constraint equations which are based on position coordinates usually are of index 3, velocity level constraints of index 2 and acceleration level constraints of index 1 [8]. In general, the higher the index of a DAE or constraint equation, the more difficult it is to solve the whole system of equations by means of numerical methods. The latter issue is detailed in the next subsection.

It should be noted that there are also alternative ways to account for constraints, apart from the exact constraint equations (2.1) combined with the method of Lagrange multipliers. One possibility would be the use of penalty formulations, which – roughly speaking – lead to a significant increase of the systems potential energy if the respective constraint is violated. For example, if one wanted to connect two material points  $\mathbf{r}_1$  and  $\mathbf{r}_2$  (in absolute coordinates) of two bodies 1 and 2 by means of a spherical joint, instead of the algebraic position-level constraint equation

$$|\mathbf{r}_1 - \mathbf{r}_2| = 0 \quad (2.12)$$

a penalty approach would introduce – in the simplest case – additional forces

$$\mathbf{f}_{1,2} = \mp k (\mathbf{r}_1 - \mathbf{r}_2) = -\frac{\partial}{\partial \mathbf{r}_{1,2}} \left( \frac{1}{2} k (\mathbf{r}_1 - \mathbf{r}_2)^2 \right) = -\frac{\partial W_{constraint}}{\partial \mathbf{r}_{1,2}} \quad (2.13)$$

on those bodies in the systems governing equations (e.g. on the right-hand side of equation (2.2)). With the penalty formulation, the constraint is not fulfilled exactly, however, by adjusting  $k$  (compared to the actual stiffnesses in the system) the deviations can be controlled.

For more information on constraints in MBD, see the respective literature (e.g. [4] and the corresponding references therein).

### 2.1.3.3. Time integration

With the spatial discretization of a dynamic problem and the consideration of constraints, in general a nonlinear system of ODEs (cf. equation (2.10), or (2.11) for the most general case) and additional algebraic equations (cf. equation (2.1)), i.e. a system of DAEs, with respect to time remains to be solved, or “integrated”. Hence, this procedure is referred to as numerical “time integration”, and its basic mathematical principles as well as some important integration schemes shall be discussed in the following.

Since most numerical time integration schemes are designed for explicit first-order systems, the first step here – as it can be done for any  $k$ -th order differential equation – is the

transformation of the equations (2.10) and (2.11) into a system of  $k$  first-order differential equations by the introduction of

$$\mathbf{q}^{(i)} = \frac{d^i}{dt^i} \mathbf{q} = \frac{d}{dt} \mathbf{q}^{(i-1)} \quad 1 \leq i \leq k-1 \quad (2.14)$$

as  $k-1$  new variables, and at the same time, additional differential equations [10]. In terms of the new set of unknowns,

$$\tilde{\mathbf{q}} = \{\mathbf{q}, \mathbf{q}^{(1)}, \dots, \mathbf{q}^{(k-1)}\}, \quad (2.15)$$

the first-order system equivalent to (2.10) is given by

$$\frac{d}{dt} \tilde{\mathbf{q}} = \begin{pmatrix} \mathbf{q}^{(1)} \\ \mathbf{q}^{(2)} \\ \vdots \\ \mathbf{q}^{(k-1)} \\ \mathbf{F}(\tilde{\mathbf{q}}, t) \end{pmatrix} =: \tilde{\mathbf{F}}(\mathbf{q}, t), \quad (2.16)$$

and analogously, equation (2.11) can be written as

$$\tilde{\mathbf{G}}(\tilde{\mathbf{q}}, \dot{\tilde{\mathbf{q}}}, t) = \mathbf{0}. \quad (2.17)$$

Since there are no universally applicable methods to solve a general implicit problem such as given in the latter equation, it is furthermore transformed into a semi-explicit DAE consisting of a system of explicit ODEs and algebraic equations by the simple variable substitution  $\mathbf{z} = \dot{\tilde{\mathbf{q}}}$  [11]:

$$\begin{aligned} \dot{\tilde{\mathbf{q}}} &= \mathbf{z} \\ \mathbf{0} &= \tilde{\mathbf{G}}(\tilde{\mathbf{q}}, \mathbf{z}, t). \end{aligned} \quad (2.18)$$

Hence, considering equations (2.16) or (2.18), as well as algebraic constraint equations (2.1), in the most general case the methods for numerical time integration have to deal with first-order DAEs of the form

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{f}(\mathbf{q}, \mathbf{z}, t) \\ \mathbf{0} &= \mathbf{g}(\mathbf{q}, \mathbf{z}, t), \end{aligned} \quad (2.19)$$

where  $\mathbf{q}$  are the variables associated with ODEs, and  $\mathbf{z}$  are the so-called ‘‘algebraic variables’’ [11].

Keeping the latter result in mind, in the following, the basic theory of time integration schemes shall be discussed. At first, the example of an explicit ODE with one unknown shall be considered only. Then, a few remarks are made on the generalization to any number of unknowns, the computational costs, the direct integration of higher-order systems, and

finally on the handling of the algebraic parts in equation (2.19), or, the constraint equations, respectively.

As it was the case with the spatial dependency (see Subsection 2.1.3.1), numerical time integration again is based on a domain discretization by splitting the time axis in discrete intervals, the so-called “time steps” of size  $dt$  which, hereafter, occasionally just shall be referred to as “time step” also; starting from any initial time  $t_0$ , the solution then is calculated successively stepwise. For a time-dependent quantity  $q(t)$  governed by the explicit ordinary differential equation

$$\dot{q}(t) = f(q(t), t), \quad (2.20)$$

at any given time  $t$  the exact solution for the next time step can be written as

$$q(t + dt) = q(t) + \int_t^{t+dt} f(q(t'), t') dt', \quad (2.21)$$

given that  $q(t)$  is known [25]. Now, a numerical integration algorithm can be defined by the approximation of the integral by means of some kind of quadrature rule, with dependence on  $dt$ ; numerical time integration thus is always closely related to quadrature rules for numerical integration, and hence, to function interpolation. The simplest case would be the explicit Euler method, obtained from the left-sided rectangle method:

$$q(t + dt) \approx q(t) + f(q(t), t) dt. \quad (2.22)$$

A very important class of algorithms which are based on the known value of the last time step are called Runge-Kutta methods (RK). For the problem considered above, the general definition of an  $s$ -stage RK method, corresponding to a general approximation scheme for the integral in equation (2.21) with  $s$  intermediate sub-intervals, can be written as [25]

$$q(t + dt) = q(t) + b_i k_i dt, \quad (2.23)$$

where  $k_i(t)$ ,  $i \in \{1, \dots, s\}$  are defined by the system of equations

$$k_i(t) = f(q(t) + a_{ij} k_j(t) dt, t + c_i dt) \quad \text{for } i, j \in \{1, \dots, s\}, \quad (2.24)$$

and the coefficients can be written in tableaux – also called “Butcher arrays” – in the form

$$\begin{array}{cccccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} & \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} & \\ \vdots & \vdots & & & \vdots & \\ c_s & a_{s1} & & \ddots & a_{ss} & \\ & b_1 & \cdots & \cdots & b_s & \end{array} \quad \text{with } 0 \leq c_1 \leq c_2 \leq \dots \leq c_s \leq 1. \quad (2.25)$$

If the matrix consisting of  $a_{ij}$  is strictly lower triangular, the integration scheme is called explicit, since in that case  $k_i(t)$  can be successively evaluated directly from  $i = 1$  to  $i = s$ , corresponding to an effectively decoupled system of equations (2.24). Otherwise, we are talking about an implicit RK-method, where at any given time  $t$  first the coupled, in general non-linear equations (2.24) have to be solved with respect to  $k_i(t)$ , which is usually only possible by means of numerical methods, and then the final integration step for the considered quantity (2.23) can be evaluated. Implicit schemes thus are computationally more expensive than explicit ones, but on the other hand, they offer higher accuracy and significantly higher stability.

At that, the accuracy of the interpolation or integral approximation which the integration method is based on defines the local error in every time step and the global error for the final solution (which is typically one order lower than the local error), and is limited by the number of stages  $s$ . In case of explicit methods, the maximum consistency order is equal to  $s$ , which means that the global error of the final solution is on the order  $\mathcal{O}(dt^s)$ ; for implicit methods the maximum consistency order is  $2s$ , respectively.

Note that all of above considerations can also be, and usually are applied to (large) systems of explicit first-order differential equations, which is done in complete analogy by just replacing  $f$ ,  $q$ , and  $k_i$  appropriately by vector quantities  $\mathbf{f}$ ,  $\mathbf{q}$ , and  $\mathbf{k}_i$ . Of course, the computational effort increases with  $s$ , particularly for implicit methods. It shall be noted that for a  $k$ -th order system in originally  $N$  variables the size of the system that actually has to be solved numerically (2.24) with respect to the variables  $\mathbf{k}_i$  in case of an  $s$ -stage implicit method is  $n_u = s \cdot k \cdot N$ , if the first-order transformation (2.15) is used. Since the resulting equations form usually a non-linear coupled system the numerical solution – using, for instance, the Newton or modified Newton method (cf. Subsection 2.1.3.4) – may be of considerable computational costs crucially depending on  $n_u$ . The effort for a direct solution via the Newton method, involving the direct inversion of the systems Jacobian using Gauss elimination, scales with  $n_u^3$ , whereas an explicit method is only quadratic in  $n_u$ ; hence, much effort has been expended in the development of efficient methods for the numerical computation of an approximate solution of linear and non-linear algebraic equations (see Subsection 2.1.3.4).

As an alternative to the class of RK methods which are, as discussed above, in case of implicit schemes with several stages computationally expensive, since the system size increases linearly with the number of stages, the so-called “multi-step methods” have been developed. Multi-step methods use extrapolation and/or interpolation functions based on the solution of the current step and the known solutions of  $k$  previous steps ( $k$ -step method) in order to approximate the integral in (2.21), and can also be either implicit or explicit schemes, depending on whether the solution of the next step is a part of those interpolation functions, or not. Importantly, the system size is always given by the original number of unknowns, independently of the number of steps. As important examples the Adams-Bashforth (ex-

licit, consistency order  $k$ ) or Adams-Moulton (implicit, consistency order  $k + 1$ ) formulas, as well as the backward differentiation formulas (BDF, implicit, consistency order  $k$ , stable for  $k \leq 6$ ; most relevant for DAEs) shall be mentioned.

Apart from that, it should be noted that there are also algorithms to integrate higher-order equations directly without the transformation to a first-order system, and thus without a multiplication of the system size; the most well-known representative of those would be the Newmark or HHT (Hilber-Hughes-Taylor) method (both implicit, consistency order 2) for the direct integration of second-order differential equations.

The main advantage of implicit methods is their high accuracy as well as stability which is particularly important, for instance, in case of mechanically stiff systems, i.e. systems including both very low and very high stiffnesses. Because of that, they moreover allow for a significantly larger time step as compared to explicit time integration, and thus, require less integration steps in total.

After the discussion of the integration of (systems of) ODEs, some notes on the algebraic part in equation (2.19) are in order. For that, let us keep in mind the application in MBD, typically with a governing system of DAEs of index 3 (cf. Subsection 2.1.3.2) originating from a second-order PDE and some kinematical constraint equations. Of course, a coupled solution of the differential and algebraic part of the governing equations is required, which can be achieved using one of the following two strategies: One can either attempt to solve the original system directly, which poses various difficulties to numerical solvers and is computationally expensive (index-3 solvers, e.g. HHT, or adapted versions of very stable higher-order implicit integration methods), or perform index reduction by explicit differentiation of (some of) the algebraic equations first, which results in a system which is easier to deal with numerically, but may need stabilization techniques to inhibit drift-off effects due to accumulated small numerical errors [8]. In any case, if there are any algebraic equations left, they are solved together with the equations of the integration scheme. For detailed information on the analysis and numerical methods in context of stiff and/or differential-algebraic systems see e.g. [11, 12].

Hence, for reasons of stability and consistency, numerical time integration in problems of MBD – possibly stiff, differential-algebraic, occasionally discontinuous (e.g. due to contact, friction, switching external forces, control) and nonlinear dynamic systems – is typically based on very stable, higher-order implicit integration schemes, for example implicit Runge-Kutta schemes (IRK) such as the RadauII or LobattoIII algorithms, or the HHT integration scheme.

In respect of the coupling between MBD and SPH it should be pointed out that the governing equations of MBD can not be solved with explicit methods which, on the other hand, are standard in the field of molecular dynamics, discrete particle methods, or SPH simulations (cf. Subsection 3.1.3 and 3.2.3). Because of that it is neither possible to directly include above discussed formulations of MBD in an SPH code, nor the other way around.

A thorough discussion on how this inherent problem is dealt with in the coupling approach developed in the work at hand can be found in Chapter 4, especially Subsection 4.4.

For a detailed summary and further information on time integration methods, stability, and applicability to the problems in multibody dynamics, see the chapter about time integration in [9]; cf. also Subsection 2.2.3.

#### 2.1.3.4. Notes on the numerical solution of linear and non-linear algebraic equations

Considering the previous sections and especially the implicit numerical time integration schemes used in MBD, we eventually end up with a system of algebraic equations in each time step. In fact, virtually any numerical treatment of any (complex) problem leads to systems of linear algebraic equations

$$\mathbf{Ax} = \mathbf{b} \tag{2.26}$$

or non-linear algebraic equations of the form

$$\mathbf{F}(\mathbf{x}) = \mathbf{0} \tag{2.27}$$

at some point, making the latter a topic of crucial importance in the field of numerical mathematics in theory and application. Therefore, an overview about standard solution strategies and numerical methods is presented in the following.

##### Linear systems

For linear systems, either direct or iterative solution methods can be used. The former yield the exact solution of (2.26) after a certain number of steps if the calculation is done exactly, and is also often referred to as factorization of  $\mathbf{A}$  in the product of a lower and an upper triangular matrix (LU factorization or LU decomposition of a square matrix), which in turn can be used to compute the solution or the inverse  $\mathbf{A}^{-1}$  and is the crucial factor determining the efficiency of the method. With a given LU decomposition

$$\mathbf{LU} = \mathbf{A} \tag{2.28}$$

the linear equation (2.26) can be solved using

$$\mathbf{y} = \mathbf{Ux} \tag{2.29}$$

and solving successively

$$\mathbf{Ly} = \mathbf{b} \tag{2.30}$$

$$\mathbf{Ux} = \mathbf{y}, \tag{2.31}$$

where the latter can be done directly and efficiently via backwards substitution because of the triangular form of the coefficient matrices [39]. The inverse of  $\mathbf{A}$  is calculated analogously, just by solving the system of linear systems of equations

$$\mathbf{A}\mathbf{X} = \mathbf{B} \tag{2.32}$$

for several column vectors  $\mathbf{x}$  and corresponding right-hand sides  $\mathbf{b}$ , forming the square matrix  $\mathbf{X}$  and  $\mathbf{B} = \mathbf{I}$ , respectively; the solution then, of course, is given by  $\mathbf{X} = \mathbf{A}^{-1}$ . Note that, for the latter, the LU factorization of  $\mathbf{A}$ , which typically is the computationally most expensive part, has to be computed only once.

Examples for important direct methods are [9]

- Gaussian elimination: For general problems; computational effort on the order  $n^3$ , where  $n$  is the system size; stabilization possible via pivoting.
- Cholesky method: Only for symmetrical problems; computational effort on the order  $n^3$ , however by a factor less than Gaussian elimination; stable without pivoting.
- LU factorization: Particularly efficient for unsymmetrical problems with band structure.

Iterative solvers only approximate the solution of the linear system, yielding higher accuracy the more number of iterations are carried out. Here, the following methods should be mentioned [9]:

- Gauß-Seidel method: Based on a fixed-point iteration; single equations of the system are solved iteratively under the presumption that the other variables are already the exact solution; typically slow convergence.
- CG-method (conjugate-gradient): For positive, symmetric problems; based on a minimization problem, the exact solution of which is given by the exact solution of the linear equation; starting from some initial value, one moves iteratively towards the solution of this minimization problem based on carefully chosen directions; the exact solution is reached after a certain maximum number of steps, however, with appropriate pre-conditioning (see below) high accuracy can be achieved within significantly less iterations.
- multigrid-methods (MG): Methods based on several grids with different coarsity; used for enhancement of global convergence (absolute solution values), as regular iterative procedures generally converge locally (with respect to the differences of solutions at adjacent grid-points) faster than globally; various techniques for adaptive “mesh refinement” / selection of the grid, error estimators, smoothing routines, pre-conditioning (see below)...; not efficient due to the overhead for those additional enhancements / optimization strategies until certain problem sizes are exceeded (typically used for very large problems).

As a compromise between the exact direct solution and iterative procedures the highly efficient approach of factorization and exact solution in combination with approximate minimum degree ordering (AMD) for large sparse (symmetric) problems should be mentioned. AMD ordering, which is based on graph-theoretical considerations and the correspondence between symmetric matrices and undirected graphs, aims at minimizing the computational costs of the factorization by minimizing the so-called “fill-in” with an appropriate reordering of the system matrix [37, 38]. At that, the “fill-in” corresponds to the number of matrix entries which change from zero to a non-zero value during the factorization, therefore crucially determines the number of floating point operations necessary in the process, and thus, the computational effort. Algorithms based on direct inversion of sparse systems with AMD ordering are implemented in high-performance solvers such as “SuperLU” or “PARDISO”.

Note: The condition number of a matrix in the context of a linear system of equations (2.26) is a measure of the worst-case error propagation, thus of the accuracy that might be lost on top of the arithmetic errors of the chosen solution method itself. It is defined with respect to a matrix norm  $\|\cdot\|_M$  as

$$\text{cond}_M(\mathbf{A}) = \left\| \mathbf{A}^{-1} \right\|_M \|\mathbf{A}\|_M \quad (2.33)$$

with

$$\frac{\|\Delta \mathbf{x}\| / \|\mathbf{x}\|}{\|\Delta \mathbf{b}\| / \|\mathbf{b}\|} \leq \text{cond}_M(\mathbf{A}), \quad (2.34)$$

where  $\|\cdot\|$  is a vector norm which  $\|\cdot\|_M$  is compatible to, i.e.  $\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\|_M \|\mathbf{x}\|$  [39]. In other words,  $\text{cond}_M(\mathbf{A})$  is an upper limit for the relative error in the solution in relation to the relative error in the right-hand side. The spectral norm in case of an Hermitian matrix  $\mathbf{A}$ , for example, would yield

$$\text{cond}_M(\mathbf{A}) = |\lambda_{max}| / |\lambda_{min}| \quad (2.35)$$

with  $\lambda_{max}$  and  $\lambda_{min}$  the largest and smallest eigenvalue of  $\mathbf{A}$ , respectively. Pre-conditioning refers to strategies for transforming the system (2.26) in order to get a better (i.e. lower) condition number; in a very simplifying way, this can be done, for example, by multiplication of the system with an appropriate matrix  $\mathbf{P}$ :

$$\mathbf{P}\mathbf{A}\mathbf{x} = \mathbf{P}\mathbf{b}. \quad (2.36)$$

A thorough mathematical discussion of this topic can be found, for instance, in [39].

### Non-linear systems

Finally, the basis for solvers of non-linear algebraic equations (2.27) includes the methods

- fixed-point iteration,
- Newton and modified Newton method,

- bisection method,
- secant method, line search / trust region, BFGS (Broyden-Fletcher-Goldfarb-Shannon) method,...

the first two of which shall be outlined below.

The fixed-point iteration treats equations of the form

$$\mathbf{F}(\mathbf{x}) - \mathbf{x} = \mathbf{0}, \quad (2.37)$$

where starting with a value  $\mathbf{x}_0$  an iteration is performed via [39]

$$\mathbf{x}_{i+1} = \mathbf{F}(\mathbf{x}_i) \quad (2.38)$$

and convergence is at least linear for sufficiently smooth functions, i.e. the error  $\varepsilon_i$  associated with iteration step  $i$  decreases as  $\varepsilon_{i+1} < m\varepsilon_i$ ,  $m \in [0, 1]$ .

The Newton method is based on a first-order Taylor expansion of  $\mathbf{F}(\mathbf{x})$  of equation (2.27) around a point  $\mathbf{x}_0$  [39],

$$\mathbf{F}(\boldsymbol{\xi}) = \mathbf{0} = \mathbf{F}(\mathbf{x}_0) + (\boldsymbol{\xi} - \mathbf{x}_0) \cdot \nabla_{\mathbf{x}} \circ \mathbf{F}(\mathbf{x}_0), \quad (2.39)$$

$$= \mathbf{F}(\mathbf{x}_0) + \left. \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right|_{\mathbf{x}_0} \cdot (\boldsymbol{\xi} - \mathbf{x}_0). \quad (2.40)$$

which can be formally solved with respect to  $\boldsymbol{\xi}$  by

$$\boldsymbol{\xi} = \mathbf{x}_0 - \mathbf{J}^{-1}(\mathbf{x}_0)\mathbf{F}(\mathbf{x}_0), \quad (2.41)$$

with the Jacobian matrix  $\mathbf{J}(\mathbf{x}_0) = \left. \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right|_{\mathbf{x}_0}$ , or iteratively written as

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i) = \boldsymbol{\Phi}(\mathbf{x}_i) \quad (2.42)$$

with the iteration matrix  $\boldsymbol{\Phi}(\mathbf{x})$ .

It features superlinear, quadratic convergence, i.e. the error decreases as  $\varepsilon_{i+1} < m\varepsilon_i^2$ ,  $m \in \mathbb{R}$  (with  $m\varepsilon_i < 1$ ). The main part of the computational costs arise from the computation of the Jacobian and solution of the linear system (2.39); thus, this is the critical point to be considered for efficiency and optimization potential. For example, if possible in any way, the Jacobian for a given system should always be calculated analytically or pre-computed automatically. The main idea which modified Newton methods are based on is that here the Jacobian is not updated in every iteration step, but reused as long as convergence still is regarded as sufficient.

For detailed information, refer to [9] or [39] and other respective literature in (applied) numerical mathematics.

## 2.2. HOTINT

### 2.2.1. What is HOTINT?

HOTINT is a comprehensive multibody code, written in object-oriented C++ with a graphical user interface (GUI) for MS Windows, allowing to perform simulation, online visualization during the process of computation, and analysis of general flexible multibody systems. It is mainly based on the multibody kernel and an element library, the solver and linear algebra libraries, along with the graphics and user interface. HOTINT can handle static as well as dynamic systems comprising rigid bodies and bodies with superimposed small deformations, classical finite elements and a broad range of structural finite elements in various formulations, any kinds of loads and kinematical constraints or other conditions, offers the reduction of the system size by a component mode synthesis (CMS), and features adaptive high-order implicit time integration based on implicit Runge-Kutta methods. From the mathematical point of view, the numerical core of HOTINT is a high-order implicit solver for a system of stiff, non-linear and – because of contact, or switching external forces, for instance – discontinuous DAEs (cf. Subsection 2.1.3).

Detailed information on the features of HOTINT, its development history, the basic structure, the implemented algorithms, and much more, is given in the manual [34] and respective literature referenced therein.

### 2.2.2. Problem definition

Although there is a GUI, access to the whole functionality is only given on source code level. Problem definition therefore splits into two parts: the creation of a so-called “model file” in C++ source code, and the specification or adaption of a wide range of various parameters via input text files. The latter include settings for the solver, such as the range for the size of the time steps, the integration scheme, or precision or convergence goals, moreover parameters concerning the data processing / management, visualization, and optionally definitions of parameters which can be used in the model files.

The process of a problem set-up and simulation thus is given by the following steps:

- Creation of the model file: Here elements can be created and connected, loads created and applied, sensors (for measurement of quantities) or constraints defined, and initial or boundary conditions set; in other words, this is the part where the multibody system is defined.
- Definition of parameters in the input file(s): As already mentioned above, options and parameters concerning any part of the whole package, are defined here.
- Compiling the model file and linking the corresponding libraries: This includes the model file in the framework of HOTINT and makes it accessible via the GUI.

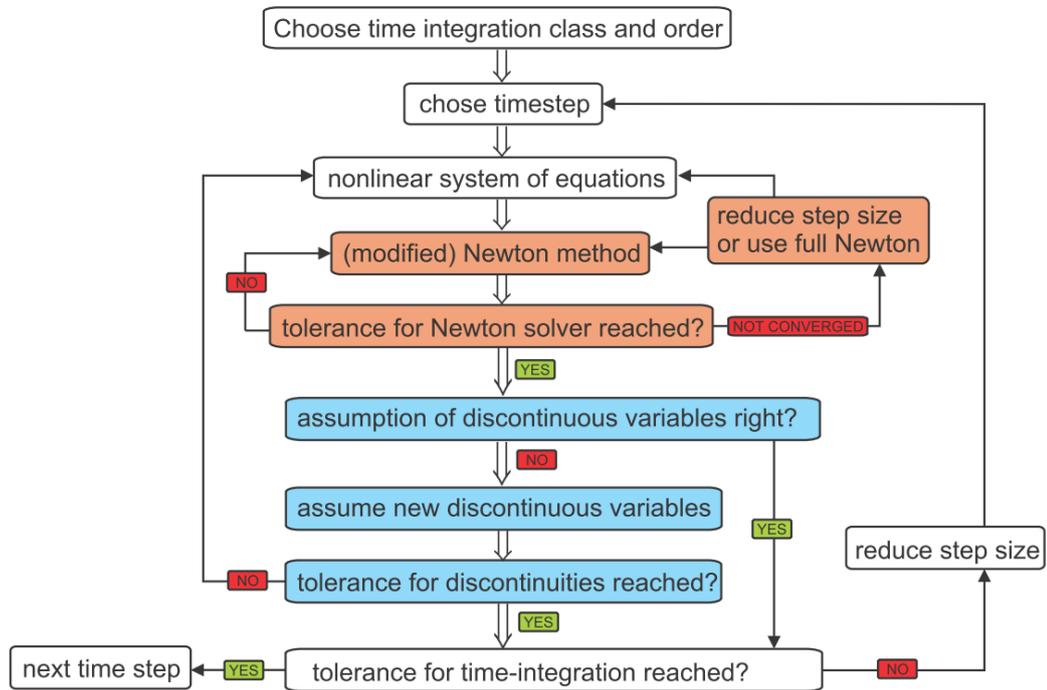
- Running the executable: The GUI of HOTINT is started, where the model can be selected and the corresponding input files are read in. Now, the simulation can be started, data can be visualized, evaluated and processed; there is a wide range of options which can be adapted before and partly even during a simulation. The main advantage of parameters defined in the input file and used in the model file is that these values – whatever they may exactly specify – in contrast to hard-coded values can be easily changed without the need for re-compilation. Furthermore, HOTINT allows to specify an automatic, predefined change of any of these values in the course of a parameter variation; the simulation then is be run multiple times with correspondingly varying parameters.

Again, this was just a very coarse overview; the HOTINT manual [34] includes detailed insights into the issue of problem set-up and simulation of a multibody system.

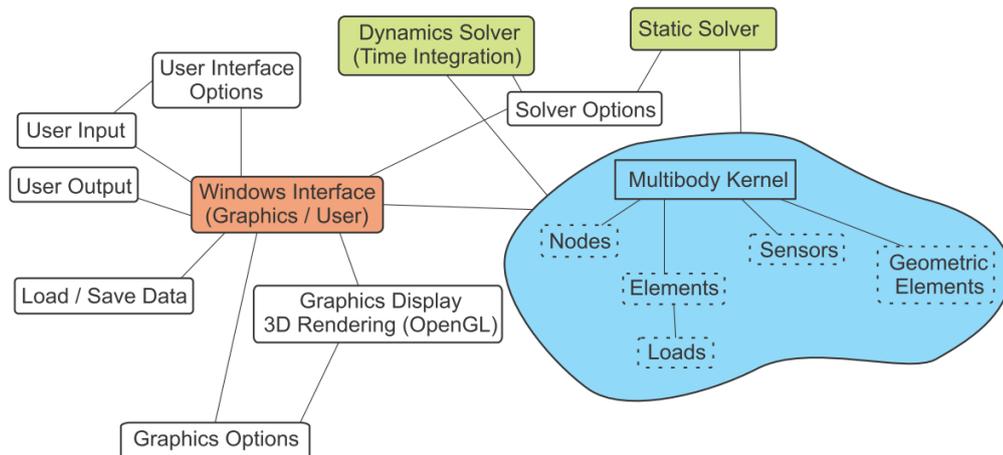
### 2.2.3. Notes on the implementation

Once more, the following can only be a very rough sketch of some features and structure of HOTINT. Figure 2.2 shows a diagram of the basic structure of HOTINT with the MS Windows interface; the class and inheritance structure of available elements, which basically constitute any multibody system in HOTINT is given in Figure 2.3. For a block diagram of the dynamic solver, based on high-order implicit RK methods and an index-2 treatment of constraints (cf. Subsection 2.1.3), see Figure 2.1.

For any further details on HOTINT – the core implementation, general structure, data and class structures, available elements, the static and dynamic solver, and many other issues, as well as the implemented algorithms and numerical methods for multibody dynamics, including the time integration strategy – again, refer to [34] and the references given there. In particular, the publications [36], [40] and, concerning time integration, [35] shall be mentioned here explicitly.



**Figure 2.1.:** Scheme of the dynamic solver. Source: [34].



**Figure 2.2.:** Structure of the HOTINT multibody system core and Windows interface. Source: [34].

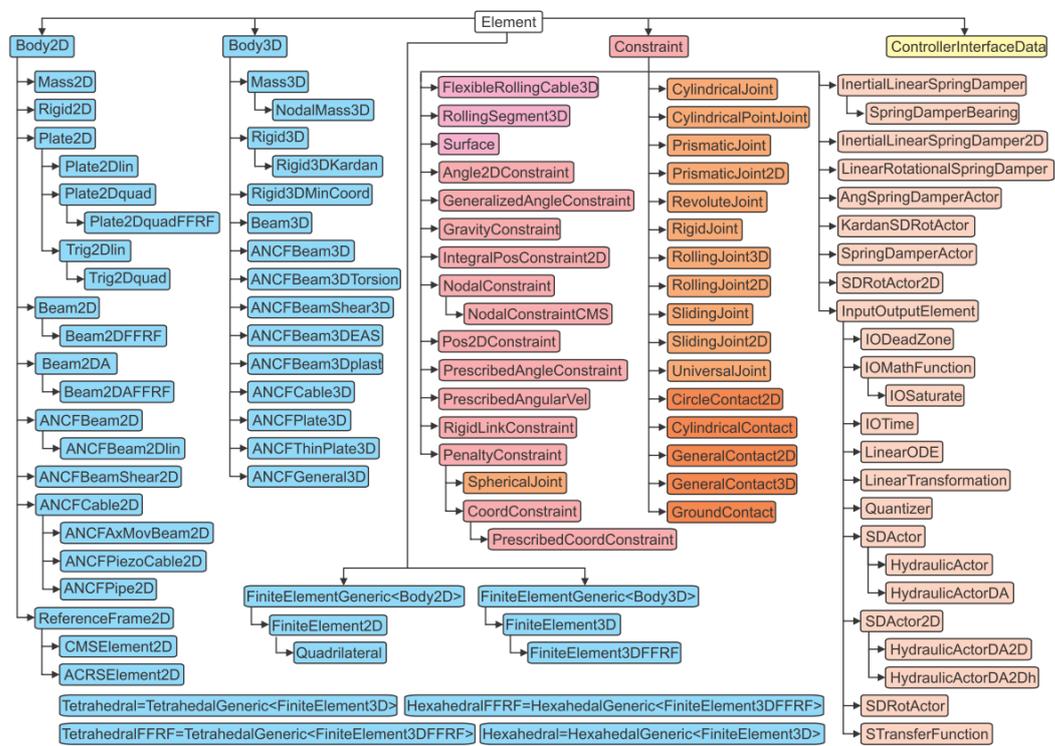


Figure 2.3.: The class structure of basic elements in HOTINT. Source: [34].

## 3. SPH and LIGGGHTS

### 3.1. Particle-based methods in fluid mechanics

#### 3.1.1. Introduction

Basically, numerical methods can be divided into two groups – grid-based methods and meshfree methods. In the field of computational fluid dynamics, today more than 90% of commercial codes are based on the finite volume method (FVM) which belongs to the class of grid-based Eulerian methods. Regarding the spatial discretization, in contrast to the material point of view in case of Lagrangian descriptions which use locally fixed points on the material itself, Eulerian methods utilize information in spatially fixed points. Thus, a FV model builds upon a spatially fixed grid which is usually generated beforehand. Despite the great success of grid-based methods, they suffer from inherent difficulties, especially with regards to mesh generation when it comes to complex geometries, arbitrarily moving and deforming boundaries, or free surfaces. Concepts and methods have been developed to deal with those difficulties, using for example techniques of adaptive remeshing. However, this usually comes at considerable costs in computational time and mesh quality.

In the case of meshfree methods, on the other hand, the problem is defined or spatially discretized via arbitrarily distributed, unconnected nodes, depending on the class of problem – a continuous field problem (e.g. continuum solid and fluid mechanics), or actually a system consisting of discrete, physical particles (discrete element methods, DEM). When used as solution approach for the former, the main advantage of meshfree over the well-established conventional grid-based methods lies at hand – adaptivity, versatility and ease of refinement, especially significant in those difficult situations mentioned above. Of course, meshfree methods are no magic cure, and they certainly have their drawbacks as well; in the end, the exact problem definition, along with the quantities and physical effects under consideration determine which numerical approach is most suitable – possibly, and particularly in case of multi-physics problems, a coupled approach which merges the advantages of both grid-based and meshfree methods suits best.

In the following, the key concept of smoothed particle hydrodynamics (SPH), a meshfree, particle-based Lagrangian method, and its application in fluid dynamics shall be presented.

### 3.1.2. Smoothed particle hydrodynamics (SPH)

In contrast to particle methods dealing with real, discrete physical particles, SPH in fluid dynamics actually deals with the fluid as a continuum. In other words, the “particles” here shouldn’t be thought of having a 1:1 correspondence in the real physical world; sheerly mathematically speaking, they are defined as nodes forming the basis of an unstructured spatial interpolation for continuous field quantities such as the density, the velocity field or the pressure field.

The latter – a spatial interpolation based on discretized values of the quantities under consideration – again serves as the starting point (cf. Subsection 2.1.3.1) for the numerical solution of the governing equations which, in case of viscous Newtonian fluids, are the full Navier-Stokes equations. The resulting system of ODEs with respect to time then is solved by numerical integration, in this case typically by means of an explicit integration scheme. It should be noted that by the application of SPH the fluid is treated in the sense of direct numerical simulations (DNS), i.e. no turbulence model is applied; in other words, the model resolution of the flow, in particular of turbulences, corresponds to and therefore is limited by the spatial resolution, which, in this case, is the spatial density of SPH particles.

The following subsections shall capture the basic theory of SPH and its application to general dynamic fluid flows, and are mainly based on the book of Liu [13], which shall also be cited as first reference for more detailed information; an interesting breakdown on numerical interpolation in general and the SPH formalism in particular is given by [14]. Originally, the SPH method was developed and at first successfully used in the field of astrophysics for the investigation of complex gas dynamics involving high-speed collisions, nuclear reactions and radiation in the ’70s, and later was introduced also in the field of fluid dynamics by Monaghan [16].

#### 3.1.2.1. Integral representation of a function and its derivatives

The first key operation for the basic SPH formulation is the definition of an integral representation of the considered field quantities, such as the velocity or the density field, and their derivatives. Using Dirac’s delta function  $\delta(\mathbf{r})$ , by definition the identity

$$\int_{\Omega} f(\mathbf{r}') \delta(\mathbf{r} - \mathbf{r}') d^3\mathbf{r}' = \begin{cases} f(\mathbf{r}) & \mathbf{r} \in \Omega \\ 0 & \text{else} \end{cases} \quad (3.1)$$

holds, and with a limit representation of the delta function

$$\lim_{h \rightarrow 0} W(\mathbf{r}, h) = \delta(\mathbf{r}) \quad (3.2)$$

we can write [13]

$$\lim_{h \rightarrow 0} \int_{\Omega} f(\mathbf{r}') W(\mathbf{r} - \mathbf{r}', h) d^3 \mathbf{r}' = \begin{cases} f(\mathbf{r}) & \mathbf{r} \in \Omega \\ 0 & \text{else} \end{cases}. \quad (3.3)$$

The integral function approximation of a function  $f(\mathbf{r})$ , hereafter denoted with a superscript “\*” as  $f^*(\mathbf{r})$ , then can be defined as

$$f^*(\mathbf{r}) \approx \int_{\Omega} f(\mathbf{r}') W(\mathbf{r} - \mathbf{r}', h) d^3 \mathbf{r}' \quad \text{for } \mathbf{r} \in \Omega, \quad (3.4)$$

with the so-called “SPH smoothing function”, “smoothing kernel”, or, in short, “kernel”  $W(\mathbf{r}, h)$  and the “smoothing length”  $h$  [13]. The kernel must be positive,

$$W(\mathbf{r} - \mathbf{r}', h) > 0 \quad \text{for } \mathbf{r} \in \Omega, \quad (3.5)$$

and has to satisfy the normalization condition

$$\int_{\Omega} W(\mathbf{r} - \mathbf{r}', h) d^3 \mathbf{r}' = 1 \quad \text{for } \mathbf{r} \in \Omega, \quad (3.6)$$

the delta function property (3.2) and the condition of compact support:

$$W(\mathbf{r} - \mathbf{r}', h) = 0 \quad \text{for } |\mathbf{r} - \mathbf{r}'| \geq \kappa h \quad \text{with } \kappa \in \mathbb{R}^+. \quad (3.7)$$

The latter requirement localizes the integration over the whole problem domain to an integration over just the support domain of the smoothing kernel; thus, hereafter  $\Omega$  shall denote the corresponding support domain.

A first estimate for the accuracy of this approximation can be obtained from a Taylor expansion of the integrand in (3.4):

$$f^*(\mathbf{r}) \approx \int_{\Omega} \left( f(\mathbf{r}) + (\mathbf{r}' - \mathbf{r}) \cdot \nabla f|_{\mathbf{r}} + p((\mathbf{r}' - \mathbf{r})^2) \right) W(\mathbf{r} - \mathbf{r}', h) d^3 \mathbf{r}'. \quad (3.8)$$

If the kernel is spherically symmetrical, which is the case with all kernel functions of the form  $W(\mathbf{r} - \mathbf{r}') = W(|\mathbf{r} - \mathbf{r}'|)$ , the integral over the linear term vanishes, yielding

$$f^*(\mathbf{r}) \approx f(\mathbf{r}) + \mathcal{O}(h^2) \quad (3.9)$$

and thus second-order accuracy with respect to the smoothing length.

The representation of spatial derivatives is calculated similarly; using the divergence theo-

rem, the approximation of the divergence of a vector field  $\mathbf{v}$  is given by

$$\begin{aligned}
 (\nabla_{\mathbf{r}} \cdot \mathbf{v}(\mathbf{r}))^* &= \int_{\Omega} (\nabla_{\mathbf{r}'} \cdot \mathbf{v}(\mathbf{r}')) W(\mathbf{r} - \mathbf{r}', h) d^3\mathbf{r}' \\
 &= \int_{\Omega} (\nabla_{\mathbf{r}'} \cdot (\mathbf{v}(\mathbf{r}')W(\mathbf{r} - \mathbf{r}', h)) - \mathbf{v}(\mathbf{r}') \cdot \nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}', h)) d^3\mathbf{r}' \\
 &= \int_{\partial\Omega} \mathbf{v}(\mathbf{r}')W(\mathbf{r} - \mathbf{r}', h) \cdot d\mathbf{S} - \int_{\Omega} \mathbf{v}(\mathbf{r}') \cdot \nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}', h) d^3\mathbf{r}' \\
 &= - \int_{\Omega} \mathbf{v}(\mathbf{r}') \cdot \nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}', h) d^3\mathbf{r}', \tag{3.10}
 \end{aligned}$$

since the surface term vanishes due to the compact support condition (3.7), if the support domain  $\Omega$  lies in the interior of the problem domain [13]. Otherwise, i.e. if the support domain only partially lies in the interior of the problem domain, the surface term – accounting for the boundary conditions – does not vanish and either has to be computed or, if neglected, accounted for in some other way. The latter is the case in this work and the utilized SPH implementation; see Subsection 3.1.2.5 for further information on that and a general discussion of the boundary treatment in SPH.

As it can be seen by the example of equation (3.10) a general characteristic of the method of SPH is the transmission of any differential operation on a field function to a differential operation on the smoothing kernel, in analogy to weak form methods where the order of differential equation effectively is reduced, and thus are the continuity requirements of the field function approximations.

### 3.1.2.2. Particle approximation

The second key operation in the SPH formalism is the spatial discretization, which is done by assigning mass, volume, and density to arbitrarily distributed SPH particles. The continuous integral approximation (3.4) then is transformed to a discrete summation over all particles lying within the kernel support domain, known as particle approximation, or SPH interpolation.

Thus, for a set of  $N$  particles with numbers  $i \in \{1, \dots, N\}$  and positions  $\mathbf{r}_i$  we assign a mass  $m_i$ , a density  $\rho_i$ , and consequently a corresponding volume  $\Delta V_i = m_i/\rho_i$  to each of them, and then define the discretized form of the integral function representations (3.4) and (3.10) via

$$\int f(\mathbf{r}') d^3\mathbf{r}' \rightarrow \sum_{i=1}^N f(\mathbf{r}_i) \Delta V_i = \sum_{i=1}^N \frac{m_i}{\rho_i} f(\mathbf{r}_i) \tag{3.11}$$

as

$$f^*(\mathbf{r}) \approx \int_{\Omega} f(\mathbf{r}') W(\mathbf{r} - \mathbf{r}', h) d^3\mathbf{r}' \approx \sum_{i=1}^N \frac{m_i}{\rho_i} f(\mathbf{r}_i) W(\mathbf{r} - \mathbf{r}_i, h) \tag{3.12}$$

and

$$(\nabla_{\mathbf{r}} \cdot \mathbf{v}(\mathbf{r}))^* \approx - \int_{\Omega} \mathbf{v}(\mathbf{r}') \cdot \nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}', h) d^3 \mathbf{r}' \approx \sum_{i=1}^N \frac{m_i}{\rho_i} \mathbf{v}(\mathbf{r}_i) \cdot \nabla_{\mathbf{r}_i} W(\mathbf{r} - \mathbf{r}_i, h), \quad (3.13)$$

where the summation in fact only runs over the particles within the respective support domain  $\Omega$  [13]. For the discretized system equations and the numerical solution procedure, the two relations above actually only are used for vectors  $\mathbf{r}$  corresponding to particle positions,  $\mathbf{r} = \mathbf{r}_j$ ,  $j \in \{1, \dots, N\}$ . Hence, the computational evaluation effectively is some kind of evaluation procedure for pair interaction, an iteration over all neighboring particles  $i$  within the kernel support distance of a given particle  $j$ . At that,  $h$  is typically chosen such that approximately 30-100 particles lie within the kernel support domain.

Doing so, we have introduced mass and density of the SPH particles, which makes this approach particularly suitable and convenient for the application in fluid mechanics, since there the density is a key field variable. If the SPH particle approximation is applied in other fields, e.g. solid mechanics, special treatment is required; one possibility would be to use the SPH approximation to create shape functions and use them for numeric integration in conventional meshfree methods based on weak (integral) forms of the governing equations in order to obtain spatially discrete system equations (see, for instance, [15]). However, this comes with some stability problems, especially in case of negative pressure, or stress, respectively (“tensile instability”), and might be also problematic since it is usually required for such methods that the number of sampling points for integration is larger than the number of field points (particles). For meshfree methods based on weak forms and application in solid mechanics, the stability may be restored using stabilization terms [13].

### 3.1.2.3. Notes on the smoothing kernel

As it has been discussed earlier, for any numerical approach for the solution of PDEs it is necessary to introduce a finite number of unknowns – in whatever way this is done – in order to break down a continuous field problem into a discrete system of ODEs. The basis of such a strategy is the link between the artificially introduced finite set of variables and the actual continuous field variable, which is defined by means of some kind of interpolation or function approximation. Thus, it is of central importance to quantify the quality of this function approximation, which is a key point deciding on numerical errors and accuracy, consistency and stability – or, in short – convergence of the numerical method: for infinitely many discrete unknowns, e.g. for an infinitely fine spatial discretization, the approximated solution should match the exact solution.

In FE approaches, convergence requires a certain degree of consistency, and with that, continuity. The degree of consistency here is associated with the order of polynomial, which the approximation can reproduce exactly, and it is closely related to the order of

accuracy it yields in the vicinity of any specified spatial point. In case of the SPH we shall use exactly this interpretation to classify the quality of the function representation (3.4).

Similarly to (3.8) and (3.9) Taylor expansion in the form of

$$f(\mathbf{r}') = \sum_{k=0}^{\infty} \frac{1}{k!} \frac{\partial^k}{\partial r_{n_1} \partial r_{n_2} \dots \partial r_{n_k}} f(\mathbf{r}) (\mathbf{r}' - \mathbf{r})_{n_1 n_2 \dots n_k}^k \quad (3.14)$$

with

$$(\mathbf{r}' - \mathbf{r})_{n_1 n_2 \dots n_k}^k = \begin{cases} 1 & k = 0 \\ (r'_{n_1} - r_{n_1})(r'_{n_2} - r_{n_2}) \dots (r'_{n_k} - r_{n_k}) & \text{else} \end{cases} \quad (3.15)$$

and

$$n_1, n_2, \dots, n_k \in \{1, 2, 3\} \quad (3.16)$$

can be used to show [13] that  $n$ -th order accuracy, i.e.

$$f(\mathbf{r}) = f^*(\mathbf{r}) + \mathcal{O}(h^n), \quad (3.17)$$

requires for the moments  $M^k$  of the kernel to satisfy

$$\begin{aligned} M^0 &= 1 \\ M^1 &= 0 \\ &\vdots \\ M^n &= 0, \end{aligned} \quad (3.18)$$

where  $M^k$  is a tensor of rank  $k$  with the Cartesian components

$$M_{n_1 n_2 \dots n_i}^k = \int_{\Omega} (\mathbf{r}' - \mathbf{r})_{n_1 n_2 \dots n_k}^k W(\mathbf{r} - \mathbf{r}') d^3 \mathbf{r}'. \quad (3.19)$$

Similar relations can be derived for  $n$ -th order accuracy of first and second derivatives of field functions, consequently involving moments of the first and second derivatives of the smoothing kernel (see [13] for more details), as well as the following additional surface term conditions:

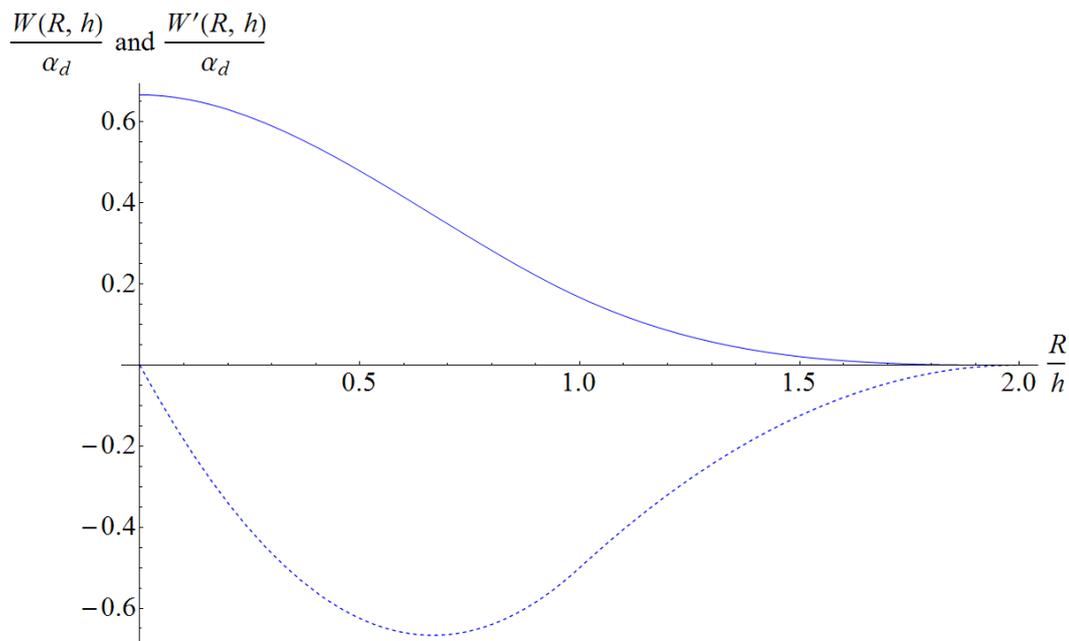
$$W(\mathbf{r} - \mathbf{r}')|_{\partial\Omega} = 0 \quad (3.20)$$

$$\nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}')|_{\partial\Omega} = 0 \quad (3.21)$$

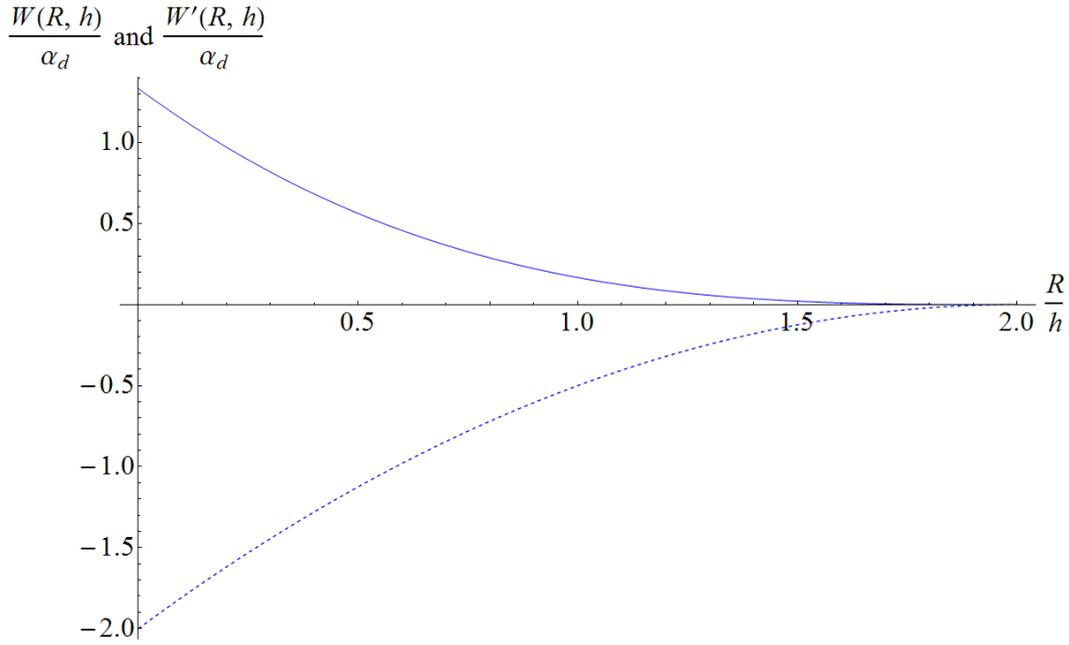
It shall be noted that  $M^0 = 1$  is equivalent to the normalization criterion (3.6), and that  $M^1 = 0$  is fulfilled in the already mentioned case of spherical symmetric smoothing functions of the form  $W(\mathbf{r} - \mathbf{r}') = W(|\mathbf{r}' - \mathbf{r}|)$ ; moreover, equation (3.20) is equivalent to the condition of compact support (3.7).

Furthermore, it can be shown that the set of conditions (3.18) to ensure  $n$ -th order accuracy of the integral function approximation also guarantees  $C^n$ -consistency (consistency of degree  $n$ ), i.e. the exact representation of a polynomial of the order  $n$ .

All this, however, in general does not apply to the SPH approximation (3.12) or (3.13) based on discrete particle summations; irregular particle distributions and the boundary regions always distort the relations obtained from the continuous integral considerations. One way to restore  $C^n$  consistency for the particle approximation would be to use a polynomial kernel with different coefficients for each particle; transforming equations (3.18) and (3.19) into discretized form and inserting the kernel function yields a linear system of algebraic equations for the unknown coefficients, for every SPH particle. The downside at that clearly are the computational costs, since it has to be done for every particle in every time step, and furthermore, in respect of the application in fluid dynamics, it may result in kernel functions which are partly negative or increase with increasing inter-particle distance, and hence lead to unphysical effects such as negative densities or negative energy. This issue, however, needs a more detailed investigation, which again lies beyond the scope of this thesis. Therefore, and since only one constant kernel function for all SPH particles shall be used in the following, two popular choices of kernel functions – among the variety of kernels that have been proposed and employed – shall be presented here concludingly:



**Figure 3.1.:** The cubic spline kernel (solid line) and its derivative (dashed), with support radius  $2h$ .



**Figure 3.2.:** The spiky kernel (line) and its derivative (dashed), with support radius  $2h$ .

Based on a cubic spline function, the “cubic spline kernel” (see Figure 3.1)

$$W(\mathbf{r} - \mathbf{r}', h) = W(|\mathbf{r} - \mathbf{r}'|, h) = W(R, h) = \alpha_d \cdot \begin{cases} \frac{2}{3} - R^2 + \frac{1}{2}R^3 & 0 \leq R < 1 \\ \frac{1}{6}(2 - R)^3 & 1 \leq R < 2 \\ 0 & \text{else} \end{cases}$$

with  $R = |\mathbf{r} - \mathbf{r}'|/h$  (3.22)

is introduced [13], with a support radius of  $2h$ . It resembles a Gaussian function with the advantage of having a narrower compact support, and the Gaussian function in turn – with the well-known limit representation of the Dirac delta function – represents the kernel suited best for finding a physical interpretation of the SPH equations according to Monaghan [17]. Up to now, the cubic spline kernel is the most widely used kernel in the SPH literature.

The constant  $\alpha_d$  is the normalization factor to satisfy condition (3.6), given by

$$\alpha_d = \begin{cases} \frac{15}{7\pi h^2} & \text{for } d = 2 \text{ (2D)} \\ \frac{3}{2\pi h^3} & \text{for } d = 3 \text{ (3D)} \end{cases} \quad (3.23)$$

depending on the dimensionality  $d$  of the problem. Note that the derivations up to this point have only been considering three-dimensional space; more details on the changes for the two-dimensional case are discussed in Subsection 3.1.2.6.

The second kernel which shall be mentioned here is the so-called “spiky kernel” (cf. Figure

3.2) [13], defined by

$$W(\mathbf{r} - \mathbf{r}', h) = W(|\mathbf{r} - \mathbf{r}'|, h) = W(R, h) = \alpha_d \cdot \begin{cases} \frac{1}{6}(2 - R)^3 & 0 \leq R < 2 \\ 0 & \text{else} \end{cases}$$

with  $R = |\mathbf{r} - \mathbf{r}'|/h$  (3.24)

with

$$\alpha_d = \begin{cases} \frac{15}{8\pi h^2} & \text{for } d = 2 \text{ (2D)} \\ \frac{45}{32\pi h^3} & \text{for } d = 3 \text{ (3D)}. \end{cases}$$
(3.25)

Note that the support radius is the same as for the cubic spline kernel (equation 3.22). The important difference here is that the absolute value of its first derivative is monotonously increasing as  $R$  – or the pairwise particle distance – approaches zero. This property plays a decisive role concerning the stability in the actual implementation (cf. Section 6.2).

#### 3.1.2.4. The SPH formulation for the Navier-Stokes equations

The governing equations for dynamic fluid flows are derived from the conservation laws of mass, momentum, and energy. Since SPH is a Lagrangian particle-based method, we use the Navier-Stokes equations for Newtonian fluids in Lagrangian description, the first two of which are given by the continuity equation,

$$\frac{D\rho}{Dt} = -\rho \nabla \cdot \mathbf{v},$$
(3.26)

and the momentum equation,

$$\frac{D\mathbf{v}}{Dt} = \frac{1}{\rho} (-\nabla p + \nabla \cdot \mathbb{T}) + \mathbf{f}$$
(3.27)

with the viscous stress tensor

$$\mathbb{T} = \mu \left( \nabla \circ \mathbf{v} + (\nabla \circ \mathbf{v})^\top - \frac{2}{3} (\nabla \cdot \mathbf{v}) \mathbf{I} \right),$$
(3.28)

the dynamic viscosity  $\mu$  and an acceleration term  $\mathbf{f}$  due to an external force field;  $\mathbf{v}$  denotes the fluid velocity field,  $\rho$  the density field, and  $p$  the absolute, thermodynamic pressure field [13]. As a side note, in the following the latter simply is referred to as “pressure field” or “pressure”. The operator  $\frac{D}{Dt}$  designates the so-called “substantial” or “material time derivative” in an Lagrangian frame of reference fixed on the body – here, infinitesimal fluid

volumes – and is connected to the Eulerian description with field variables via

$$\frac{D}{Dt}A_{Lagrangian} = \left( \frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla \right) A_{Eulerian} \quad (3.29)$$

for any quantity  $A$  [19]. Equations (3.26) and (3.27), together with a scalar equation of state,

$$p = p(\rho), \quad (3.30)$$

form the complete set of equations for the description of the dynamics of Newtonian fluids, provided that temperature effects, temperature dependence of material properties, heat conduction and dissipation are neglected, which shall be assumed in the following.

Now, the SPH particle approximation (3.12) and (3.13) is applied to above equations, for all particles  $i \in \{1, \dots, N\}$  at their respective positions  $\mathbf{r}_1 \dots \mathbf{r}_n$ , yielding a discrete set of  $N$  ordinary differential equations with respect to time [13]:

$$\frac{D\rho_i}{Dt} = \sum_{j=1}^N m_j \mathbf{v}_{ij} \cdot \nabla_{\mathbf{r}_i} W_{ij} \quad (3.31)$$

$$\frac{D\mathbf{v}_i}{Dt} = \sum_{j=1}^N m_j \left( \frac{\mathbf{S}_i}{\rho_i^2} + \frac{\mathbf{S}_j}{\rho_j^2} \right) \cdot \nabla_{\mathbf{r}_i} W_{ij} + \mathbf{f}_i \quad (3.32)$$

with

$$\begin{aligned} \mathbf{v}_{ij} &= \mathbf{v}_i - \mathbf{v}_j \\ W_{ij} &= W(\mathbf{r}_i - \mathbf{r}_j, h) \\ \mathbf{S}_i &= -p_i \mathbf{I} + \mathbf{T}_i, \end{aligned}$$

where  $\{\mathbf{r}_i, \mathbf{v}_i, p_i, m_i, \rho_i, \mathbf{f}_i\}$  is complete set of quantities describing SPH particle  $i$ . Once again, for a detailed derivation the reader is referred to [13].

However, two things remain to be pointed out – firstly, the calculation of the density, and secondly, the inclusion of the viscous terms. For the density calculations, one can either use the so-called “continuity density approach” given in equation (3.31), or the summation density approach directly using the SPH particle approximation (3.12) [13]:

$$\rho_i = \sum_{j=1}^N \rho_j \frac{m_j}{\rho_j} W_{ij} = \sum_{j=1}^N m_j W_{ij} \quad (3.33)$$

The main advantages of (3.31) over (3.33) is that the initial density can be set and only changes in case of relative particle movement and moreover, that it yields more accurate results in vicinity of the domain boundary; in these regions, the kernel support domain lies partly outside the problem domain, and thus the summation approach leads to an unphysical drop in density, and consequently, in pressure. One possibility to reduce the

errors with the summation approach in boundary regions is an additional normalization of the sum [13],

$$\rho_i = \frac{\sum_{j=1}^N m_j W_{ij}}{\sum_{j=1}^N \frac{m_j}{\rho_j} W_{ij}}. \quad (3.34)$$

However, both the modified summation as well as the continuity approach have the disadvantage that contrary to the SPH interpolant (3.33) exact conservation of mass is not retained [17].

As to the viscosity, there are also various different approaches: Originally, SPH was introduced without viscous terms for problems without or with very low dissipation, corresponding to Eulerian fluids, and, with  $\mu = 0$  and  $\mathbb{T} = 0$ , instead of (3.32) the inviscid SPH momentum equation

$$\frac{D\mathbf{v}_i}{Dt} = - \sum_{j=1}^N m_j \left( \frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \cdot \nabla_{\mathbf{r}_i} W_{ij} + \mathbf{f}_i. \quad (3.35)$$

For inclusion of viscosity, one can now either use (3.32) with the SPH particle approximation of  $\mathbf{S}_i$ , or add an artificial viscosity term  $\Pi$ , many forms of which have been proposed, to the Euler equation (3.35) [13]:

$$\frac{D\mathbf{v}_i}{Dt} = - \sum_{j=1}^N m_j \left( \frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} + \Pi_{ij} \right) \cdot \nabla_{\mathbf{r}_i} W_{ij} + \mathbf{f}_i. \quad (3.36)$$

The probably most widely-used approach is the artificial viscosity introduced by Monaghan and Gingold [18], given by

$$\Pi_{ij} = \begin{cases} \frac{-\alpha c_{ij} \mu_{ij} + \beta \mu_{ij}^2}{\rho_{ij}} & \mathbf{r}_{ij} \cdot \mathbf{v}_{ij} < 0 \\ 0 & \mathbf{r}_{ij} \cdot \mathbf{v}_{ij} > 0 \end{cases} \quad \text{with } \mu_{ij} = \frac{h \mathbf{r}_{ij} \cdot \mathbf{v}_{ij}}{|\mathbf{r}_{ij}|^2 + \eta^2}, \quad (3.37)$$

where  $c$  is the speed of sound,  $h$  denotes the smoothing length,  $\alpha$  and  $\beta$  are two constant parameters,  $\eta \approx 0.1h$  is a factor inserted to prevent numerical divergence, and the notation  $A_{ij} = (A_i + A_j)/2$  is used for any occurring scalar quantity, which approximates a dynamic viscosity of  $\mu \approx \alpha h c \rho$  corresponding to a kinematic viscosity of [16]

$$\nu = \frac{\mu}{\rho} \approx \alpha h c. \quad (3.38)$$

The latter can be shown by assuming  $\beta \approx 0$ ,  $\eta \approx 0$ ,  $|\mathbf{r}_{ij}| \approx h$ ,  $\mathbf{r}_{ij} \cdot \mathbf{v}_{ij} \approx h \mathbf{v}_{ij}$  and equal mass  $m$  for all particles, and comparing the resulting viscosity term in equation (3.36) to the viscous term we would get for an incompressible fluid ( $\nabla \cdot \mathbf{v} = 0$ ) in the momentum equation (3.27),

$$\frac{1}{\rho} \nabla \cdot \mathbb{T} = \frac{\mu}{\rho} \Delta \mathbf{v}, \quad (3.39)$$

written directly in SPH particle approximation (cf. Subsection 3.1.2.2), where

$$\Delta \mathbf{v}_i \approx \frac{m}{\rho} \sum_j \mathbf{v}_{ij} \Delta_{\mathbf{r}_i} W_{ij}. \quad (3.40)$$

Note that derivatives of functions are transferred to derivatives of the smoothing kernel in the SPH formalism, as discussed in Subsection 3.1.2.1. With  $\Delta_{\mathbf{r}_i} W_{ij} \approx h^{-1} |\nabla_{\mathbf{r}_i} W_{ij}|$  the above mentioned comparison (of absolute values) finally yields

$$\frac{\mu}{\rho} \frac{1}{\rho h} |\nabla_{\mathbf{r}_i} W_{ij}| \mathbf{v}_{ij} \approx \frac{\alpha c h^2 \mathbf{v}_{ij}}{\rho h^2} |\nabla_{\mathbf{r}_i} W_{ij}|, \quad (3.41)$$

and thus

$$\mu \approx \alpha h c \rho. \quad (3.42)$$

Furthermore, it shall be noted that the condition  $\mathbf{r}_{ij} \cdot \mathbf{v}_{ij} < 0$  in the definition (3.37) of the artificial viscosity term is satisfied if two SPH particles  $i$  and  $j$  are approaching each other, and that it is the SPH equivalent of  $\nabla \cdot \mathbf{v} < 0$ , which – considering the continuity equation (3.26) – results in viscous forces only for locally increasing density.

The artificial viscosity approach of equation (3.37) is also implemented in the particle simulation code LIGGGHTS discussed in Section 3.2; for more information on actually used parameters cf. Section 6.2. For some further details concerning various SPH matters see also Subsection 3.2.3.

Concludingly, a short illustrative interpretation of the governing equations for fluid dynamics in the SPH formalism with the continuity density approach shall be given. To this end, let us consider the spiky kernel given in equation (3.24) for a particle  $i$  and a neighboring particle  $j$ . Since it is a spherically symmetrical function depending on the inter-particle distance only, the gradient is given by

$$\nabla_{\mathbf{r}_i} W_{ij} = \nabla_{\mathbf{r}_i} W(|\mathbf{r}_i - \mathbf{r}_j|, h) \propto \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|} \frac{\partial W}{\partial R}, \quad (3.43)$$

a vector proportional to  $\mathbf{r}_{ij}$ , the vector of the relative position of particle  $i$  with respect to particle  $j$ . From equation (3.31) and the fact, that for the spiky kernel  $\frac{\partial W}{\partial R} < 0$  over the whole support domain, it follows that the density increases, if the relative velocity  $\mathbf{v}_{ij}$  is oriented “against” the relative particle position; otherwise it decreases. In other words, when two particles approach each other, both of their densities increase, and the other way around. Disregarding the viscous forces, similarly, the momentum equation (3.35) yields a force in direction of  $\mathbf{r}_{ij}$  in case of – roughly speaking – positive pressure, and a force in the opposite direction for negative pressure (relative to the outer, environmental pressure); thus, positive pressure tries to keep particles separate, and vice versa. Hence, for locally higher pressure the particles are pushed apart, resulting in a decrease of their densities and with that, the associated pressure, and then consequently a decrease of the force, until the

situation is reversed, the pressure changes sign and the force changes direction, causing the particles to move back together. It should be noted that this mechanism is a prerequisite for stability.

### 3.1.2.5. Boundary conditions

In fluid dynamics there is a variety of possible boundary conditions for the respective field variables, including the specification of  $\mathbf{v}$ , the pressure  $p$  or the total pressure  $p + \frac{1}{2}\rho\mathbf{v}^2$  at the boundary (Dirichlet conditions), the pressure gradient  $\nabla p$  at the boundary (Neumann condition), or the mass flux  $\rho\mathbf{v}$  through the boundary, in order to account for any kind of inlet, outlet, or moving wall geometries. In any case, for flows confined by impervious walls, the “no-penetration” condition

$$\mathbf{v}_{rel,\perp} = \mathbf{0} \text{ at the boundary,} \quad (3.44)$$

and in case of viscous fluids, additionally the “no-slip” condition

$$\mathbf{v}_{rel,\parallel} = \mathbf{0} \text{ at the boundary} \quad (3.45)$$

with the relative velocity  $\mathbf{v}_{rel,\perp} = ((\mathbf{v} - \mathbf{v}_{wall}) \cdot \mathbf{n}) \cdot \mathbf{n}$  in direction normal to the wall, i.e. parallel to surface unit normal vector  $\mathbf{n}$ , and the corresponding parallel component  $\mathbf{v}_{rel,\parallel} = -\mathbf{n} \times \mathbf{n} \times (\mathbf{v} - \mathbf{v}_{wall})$ , have to be satisfied [19].

The implementation of boundary conditions in SPH still is a matter of current research, and yet no method for boundary treatment has established itself as standard without compromise. As it has already been mentioned in the previous section, inherent difficulties arise near the boundaries, where the kernel support domain partly lies outside the problem domain, and thus, the particle approximation of the function interpolation is impaired due to truncation of the integrals. Typical approaches are based on additional (virtual) SPH particles, possibly with other properties than the fluid particles, but included in the SPH summations of the latter, which are placed on the walls, or behind the walls, fixed in space or free in motion, and constructed such, e.g. by means of mirroring techniques, that the boundary condition under consideration is satisfied. Other variants include the explicit computation of surface terms (cf. equation (3.10)), modifications of the kernel in the vicinity of the boundary, or, for enforcement of above no-penetration and no-slip condition, the direct use of additional force fields based on a measure of distance (e.g. the shortest normal distance) between the SPH fluid particles and the boundary surface [41].

Hence, the treatment of boundary conditions in the SPH formalism is a difficult issue on its own, and shall not be investigated here in more detail. As far as the approach to fluid-structure interaction and the corresponding implementation of boundaries go in this thesis, the focus lies on the two essential equations (3.44) and (3.45) only, and is discussed

thoroughly in Chapter 4. More information about recent approaches and the investigation of the issue of boundary conditions, as well as a wide range of other specific subjects in fluid dynamics – with and without the SPH methodology – can be found in [42].

### 3.1.2.6. 2D versus 3D

The transition from 3D to 2D is performed considering a quasi-twodimensional 3D-case in the  $xy$ -layer, where for all field quantities  $q(\mathbf{r}, t) = q(x, y, z, t)$  at any given point  $(x, y)$  and time  $t$  the conditions

$$q(x, y, z, t) = q(x, y, 0, t) \quad \forall z \in \mathbb{R} \quad (3.46)$$

and, of course,

$$v_z \equiv 0 \quad (3.47)$$

hold, and the considered geometrical domain is assumed to have infinite extension in  $z$ -direction, resulting in an invariance of the problem under translation in  $z$ -direction. With that, instead of an infinitesimal Cartesian volume element  $dxdydz$  we can equivalently consider volume elements  $dxdy \cdot z_0$  with an arbitrarily chosen depth  $z_0$ , and formulate the governing equations in analogy to 3D based on the conservation principles. Correspondingly, a volume integral is effectively transformed to a plane surface integral times the depth unit  $z_0$

$$\int f(\mathbf{r}) dV \rightarrow z_0 \int f(x, y) dxdy = z_0 \int f(x, y) dA, \quad (3.48)$$

and, after division of any integral relations by  $z_0$  in order to eliminate the  $z_0$ -dependency the equations formally remain unchanged except for surface instead of volume integration and independence of  $z$ , provided that all “integral” or extensive quantities are defined relative to a unit depth. For example, in the 3D case, the mass, as such a quantity,

$$m = \int \rho dV, \quad (3.49)$$

is transformed to a mass  $m^{2D}$  per depth unit,

$$m^{2D} = \frac{m}{z_0} = \int \rho dA, \quad (3.50)$$

whereas the density, on the other hand, stays unchanged as an intensive, i.e. scale-invariant system property.

Of course, the SPH interpolation (3.4) now needs to be built upon the two-dimensional Dirac delta function, since a spatial interpolation for two coordinates is required. Consequently, all interpolation integrals become plane surface integrals, with a compact support domain in form of a circle, and, importantly, the normalization factor for a given smoothing kernel to satisfy (3.6), or the corresponding condition in 2D, changes depending on the dimensionality (cf. equations (3.23) and (3.25)). For the integral representation of the derivative of a

function (cf. (3.10)), the divergence theorem for two dimensions is used, and again, the boundary term vanishes due to the compact support condition, if the support domain lies inside the problem domain. Finally, the particle approximation is realized by the use of discrete surface elements  $\Delta A_i$  associated with every particle,

$$\int f(\mathbf{r}) dA \rightarrow \sum_{i=1}^N f(\mathbf{r}_i) \Delta A_i = \sum_{i=1}^N \frac{m_i/z_0}{\rho_i} f(\mathbf{r}_i) = \sum_{i=1}^N \frac{m_i^{2D}}{\rho_i} f(\mathbf{r}_i). \quad (3.51)$$

In summary, all of the derivations in above subsections can be made in complete analogy for the 2-dimensional case, and hence shall not be repeated here; furthermore, keeping in mind that all extensive quantities are related to the unit depth, the governing equations in SPH particle approximation (3.31) and (3.32) for 2D are – except for the missing (or trivial)  $z$ -component – formally identical to the 3D case.

### 3.1.3. Numerical point of view

Numerically, the problem is very much different than the situation in multibody dynamics (cf. Subsection 2.1.3). Equations (3.31) and (3.32) can be integrated directly by means of explicit methods, where the application and implementation of SPH is quite analogous to implementations of simulators for problems of many-particle physics with a short-ranged pair potential. In fact, for an arbitrarily chosen SPH particle  $i$  the right-hand side of (3.31) and (3.32) merely consists of pair-wise contributions of particle  $i$  and any other particle which lies within the kernel support domain around particle  $i$ . Hence, the kernel plays the role of a pair potential, and the radius of the compact support corresponds to its cut-off radius, i.e. the inter-particle distance at which (and from which on) the potential or interaction force vanishes. Thus, the main task – except for the more or less straightforward time integration and analysis of the simulation data – is an efficient calculation of the pair interaction. Using brute-force, i.e. moving through all possible pairs of all particles in every time step, results in a computational effort proportional to  $N^2$ , where  $N$  is the total number of particles, which is not feasible for large systems. One possibility to reduce the effort to the order  $N$  would be the so-called “cell method” of MD [27]. Here, a regular grid is created with cell sizes corresponding to the cut-off (or, in case of SPH, the kernel support radius) and in every time step all particles are hashed on that grid (which itself is a operation linear in  $N$ ). Consequently, for any given particle  $i$  located in a certain cell, only the other particles in that cell and in the adjacent cells need to be considered. More sophisticated approaches, e.g. Verlet lists (neighbor lists), also take into account the current particle velocities, for instance, in order to estimate possible interaction partners for the next time step based on the actual particle configuration and to make the pair evaluation procedure even more efficient, resulting in a computational effort still on the order  $N$ , of course, but with a smaller proportionality factor.

## 3.2. LIGGGHTS

### 3.2.1. What is LIGGGHTS?

LIGGGHTS stands for LAMMPS Improved for General Granular and Granular Heat Transfer Simulations. The latter, LAMMPS, short for Large-scale Atomic/Molecular Massively Parallel Simulator, is a classical molecular dynamics code widely used in the field of molecular dynamics and for approaches based on discrete element methods (DEM) that models an ensemble of particles in a liquid, solid, or gaseous state, simulating atomic, polymeric, biological, metallic, granular, and coarse-grained systems in 2D or 3D using a variety of force fields and boundary conditions. It is designed for parallel computers allowing the investigation of systems consisting of a few up to billions of particles. However, it does not offer a GUI or any post-processing, analysis of the simulation data, or visualization [24].

The current version is written in C++ and utilizes the standardized message-passing system MPI (“message-passing interface”, see [50, 51], or Subsection 5.1) for parallelization; it is a freely-available open-source code, distributed under the terms of the GNU Public License by Sandia National Labs, and was originally developed under a US Department of Energy CRADA (Cooperative Research and Development Agreement) between several Department of Energy laboratories and companies [24].

Now, LIGGGHTS, itself an open-source project, aims to improve those capabilities with the goal of application in the industry, and adds, amongst other features, import and handling of complex wall geometries from CAD, a moving mesh feature to account for moving geometry, additional inclusion of optional cohesive and rolling friction forces, as well as heat conduction between particles in contact, and, importantly, an implementation of the SPH formalism (cf. Subsection 3.1.2) [22].

Further information on the capabilities of LAMMPS and LIGGGHTS can be found in the LIGGGHTS documentation [23].

### 3.2.2. Problem definition

With LIGGGHTS the simulation set-up and problem definition is done entirely via a so-called “input script”. A broad range of LAMMPS and LIGGGHTS commands is available for the specification of

- any parameters concerning the problem (mass, density, initial conditions,...) and the solver (e.g. size of time step, total number of time steps,...),
- the problem domain and particle arrangement, as well as simple (and complex static) wall definitions,
- the type of integration to be used, the algorithms for pair evaluation and neighbor list builds, the way thermodynamic information is evaluated and processed,

- additional filters, temperature control, smoothing algorithms,... to be applied in every time step,
- and, finally, the output style and configuration, as well as logging options.

Again, it would go beyond the scope of this thesis to go more into detail here; see [23] for a complete reference over the functionality, as well as the how-to of installing LIGGGHTS, setting-up, running and evaluating a simulation.

### 3.2.3. Notes on the implementation

In the most general sense, LAMMPS integrates Newton's equations of motion for collections of atoms, molecules, or macroscopic particles that interact via short- or long-range forces with a variety of initial and/or boundary conditions, using explicit Velocity-Verlet integration (or optionally the rRESPA algorithm which consists of a series of Velocity-Verlet-like algorithms [21]).

The Velocity-Verlet algorithm is an explicit symplectic integrator well known in the field of molecular dynamics (MD); it has the convergence order 2, and is defined by

$$\begin{aligned}
 \mathbf{r}_i(t + dt) &= \mathbf{r}_i(t) + \mathbf{v}_i(t)dt \\
 \mathbf{v}_i(t + dt) &= \mathbf{v}_i(t) + \frac{1}{2m_i}\mathbf{F}_i(\mathbf{R}(t)) + \frac{1}{2m_i}\mathbf{F}_i(\mathbf{R}(t + dt))dt \\
 &\text{with } i \in \{1, \dots, N\} \\
 &\text{and } \mathbf{R}(t) = \{\mathbf{r}_1(t), \mathbf{r}_2(t), \dots, \mathbf{r}_N(t)\}
 \end{aligned} \tag{3.52}$$

where  $\mathbf{r}_i(t)$ ,  $\mathbf{v}_i(t)$ , and  $m_i$  is the position, velocity and mass of particle  $i$ ,  $N$  the total number of particles, and  $dt$  the time step; the force  $\mathbf{F}_i(\mathbf{R}(t))$  on particle  $i$  is assumed to depend merely on the particle positions [26]. Note that the computation of the forces, which is the most time-consuming part, has to be done only once per time-step.

For computational efficiency neighbor lists are used to keep track of nearby particles (see also Subsection 3.1.3) which are optimized for systems with particles that are repulsive at short distances, i.e. systems with limited local density of particles. On parallel machines, LAMMPS uses spatial decomposition techniques to partition the simulation domain into small 3D subdomains, one of which is assigned to each process. At that, a process is associated with one or several CPU cores together with a memory space. Communication between the processes is done via MPI routines (see [50, 51]), where the communicated data consists of information about so-called "ghost atoms" which are all atoms that border their subdomain within the range of interaction. In other words, the only thing one process needs to know about the rest of the system is the data of particles which lie outside its own subdomain, but still within range of interaction and thus yield contributions to the computation of the interaction terms for the local particles. The highest efficiency in a parallel sense is reached for systems the particles of which fill a 3D rectangular box with

roughly uniform density; of course, for small systems, i.e. in case of small numbers of particles, there is a limit for the number of processes, at which efficiency starts to decrease due to communication overhead. Technical details of the algorithms used in LAMMPS can be found in [20, 21]; see also Section 5.5 for additional information concerning the parallelization.

In conclusion, since LIGGGHTS is used here for the SPH simulation, some notes on the corresponding implementation are in order. The SPH formalism is implemented in LIGGGHTS for viscous fluids with an artificial viscosity according to Monaghan (cf. equation (3.37) and (3.36)), but with a constant value of the speed of sound defined in the input script, and uses Tait's equation of state [16],

$$p(\rho) = \frac{c^2 \rho_0}{\gamma} \left( \left( \frac{\rho}{\rho_0} \right)^\gamma - 1 \right), \quad (3.53)$$

where  $\rho_0$  designates the nominal density,  $c$  is the speed of sound, and  $\gamma$  the polytropic constant which is typically chosen as 7. This equation establishes a stiff relation between the density and the pressure, i.e. small variations in density result in large changes in pressure, and is a common choice for the SPH simulation of weakly compressible fluids. Hereby, the parameter  $c$  needs to be chosen carefully – taking the real value for nearly incompressible fluids, such as water, comes at the high prize of an extremely small time step in the integration procedure to retain stability, taking a value too small results in comparatively large density fluctuations. In practise, the speed of sound is defined as approximately ten times the maximum bulk flow velocity  $v_{bulk}$ , for which the density variations  $\Delta\rho = |\rho - \rho_0|$  can be estimated according to Monaghan [16] by

$$\frac{\Delta\rho}{\rho_0} \approx \left( \frac{v_{bulk}}{c} \right)^2 \approx 1\%. \quad (3.54)$$

Hence, almost incompressible fluids are modelled by orders of magnitude more compressible than they actually are, but still rather incompressible, allowing for a larger time step and, therefore, reducing the computational effort.

As to the smoothing function, originally only the cubic spline kernel (see equation (3.22)) in 3D was implemented in LIGGGHTS; additionally I implemented the cubic spline kernel normalized for 2D simulations, as well as the spiky kernel for 2D and 3D (cf. equation (3.24)). In any case, the simplest approach with one kernel function with a predefined smoothing length for all particles is used (cf. Subsection 3.1.2.3).

The density is calculated via the continuity approach (3.31), using the explicit Euler integration (cf. (2.22)). Since the pressure field in SPH generally is subject to considerable oscillations, an optional filter can be additionally applied to the density field to smooth out those oscillations. The algorithm implemented is called “Shepard filter”, and consists of nothing else than the application of the modified density approach (cf. equation (3.34))

once in every  $n$  time steps, where  $n$  can be set arbitrarily via the input script.

Finally, one thing concerning the use of the Velocity-Verlet integrator for the momentum equations (3.36) should be noted: The right-hand side actually also depends on the particle velocities due to the viscous terms, whereas the algorithm (3.52) considers forces depending on the particle positions only. Thus, every recalculation of  $\mathbf{F}(\mathbf{R}(t), \mathbf{V}(t)) = \mathbf{F}(t)$  is based on the updated particle positions, however, on the velocities from one substep before,

$$\begin{aligned} \mathbf{r}_i(t + dt) &= \mathbf{r}_i(t) + \mathbf{v}_i(t)dt \\ \tilde{\mathbf{v}}_i(t + dt) &= \mathbf{v}_i(t) + \frac{1}{2m_i} \mathbf{F}_i(t)dt \\ \mathbf{F}_i(t + dt) &= \mathbf{F}_i(\mathbf{R}(t + dt), \tilde{\mathbf{V}}(t + dt)) \\ \mathbf{v}_i(t) &= \tilde{\mathbf{v}}_i(t + dt) + \frac{1}{2m_i} \mathbf{F}_i(t + dt)dt \end{aligned}$$

with  $i \in \{1, \dots, N\}$  and

$$\begin{aligned} \mathbf{R}(t) &= \{\mathbf{r}_1(t), \mathbf{r}_2(t), \dots, \mathbf{r}_N(t)\} \\ \tilde{\mathbf{V}}(t) &= \{\tilde{\mathbf{v}}_1(t), \tilde{\mathbf{v}}_2(t), \dots, \tilde{\mathbf{v}}_N(t)\}, \end{aligned} \tag{3.55}$$

which probably leads to a reduced order of convergence of the integration scheme. Nevertheless this does not impair the global solution considerably, since the viscous terms are typically much smaller than the pressure-related terms, and the density is integrated with a convergence order of only one anyways.

For more detailed information concerning the implementation, the reader is referred to [22], the LIGGGHTS documentation [23], and the source code which is also freely available at [22].

## 4. Fluid-Structure Interaction – Direct coupling of flexible MBD and SPH

### 4.1. Coupling concept – the main idea

#### 4.1.1. Introduction

As it has already been pointed out in the introduction (cf. Chapter 1), the central subject of this thesis is an approach to FSI by means of direct coupling of flexible MBD with SPH for general dynamic fluid flows by coupling of HOTINT and LIGGGHTS, with the objective of combining the strengths of both – a highly effective treatment of complex multibody systems on the one hand, and an efficient and flexible model for the fluid simulation on the other hand. The desired key features then are stability and consistency, flexibility with respect to arbitrary geometry as well as geometrical and possibly topological changes, and performance and efficiency.

In general, any coupled approach to FSI inherently consists of two more or less separate and different parts of implementation, depending on the models used for the description of the fluid and the structural parts. The choice of these models is restricted by their mutual compatibility, effectively from a mathematical point of view. Moreover, an implementation can be classified either as monolithic or non-monolithic design, according to whether it is one single application or consists of two separate codes, possibly even on two separate machines. In any case, the first and crucial step to be taken on the way to a coupled approach to these, or in general, all multiphysics problems, is the definition of an interface.

The interface comprises the data structure as well as routines for data exchange, where the former is defined by the mathematically necessary information for the connection of the models, i.e. all the data to be exchanged mutually between the models in order to obtain a well-posed problem situation on either side, and the latter are determined by the design of the implementation. In case of similar models for the fluid and the solid part, e.g. a particle method and a lattice-type model, respectively, the coupling can be achieved relatively easily, from both the mathematical as well as the implementation point of view, particularly, if the design is monolithic, and thus, the data exchange and communication can be done directly within the same process on the main memory. For a non-monolithic design, the data exchange is more difficult, especially when working on two different machines with the need for a dedicated system for communication.

### 4.1.2. A non-monolithic approach via TCP/IP

Considering the above, the approach at hand is based on very different models for the fluid and structural side, respectively, and inherently non-monolithic, since LIGGGHTS is a fully-parallel 64-bit application for UNIX/LINUX systems only, and the employed version of HOTINT is a 32-bit multi-threaded software package running exclusively under MS Windows.

The main steps for the realization of this approach are

- the development of a basic idea for a possible coupling solution under the given circumstances,
- the development of a suitable and flexible interaction model / a contact formalism between the multibody formulations and the SPH fluid particles,
- the definition of an interface (data and data exchange),
- the implementation of communication between HOTINT and LIGGGHTS,
- the implementation of the coupling formalism on both sides,
- and, last but not least, synchronization.

The main idea for the coupling solution is based on the introduction of a client/server relation between the two sides, assigning HOTINT the server and LIGGGHTS the client role; TCP/IPv4 shall be utilized to realize the data transfer and communication, providing a maximum of flexibility and independence. The reason why the roles are assigned this and not the other way around is not to far to seek – contrary to LIGGGHTS, HOTINT offers comprehensive possibilities for data (post-)processing, evaluation and real-time visualization, along with a GUI, and thus qualifies to be the central application which the complete management and control of the whole system is residing with.

At that, the coupling shall be accomplished effectively within the time stepping: HOTINT calculates one time step and updates positions, velocities and deformation of the components of the multibody system accounting for additional forces due to the fluid, then exchanges this information with LIGGGHTS, which in turn computes the next time step in the fluid simulation on that basis, recalculates the forces on the bodies, and communicates them with HOTINT, completing the necessary data for the computation of the next multibody time step, and so on.

Concludingly, the tasks of either side can be summarized as the following:

#### **Server / HOTINT MBD:**

- calculation of MBD with additional forces due to the fluid
- coordination of the simulation and communication, control of LIGGGHTS

- problem set-up for both sides
- data management and data processing, visualization

**Client / LIGGGHTS SPH:**

- fluid simulation with given, arbitrarily moving/deforming boundaries (MPI parallel)
- communication with the server

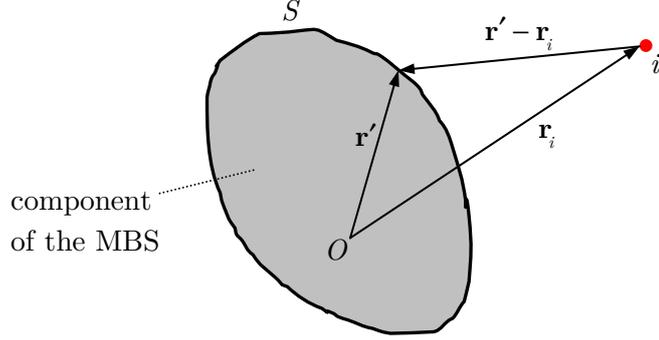
Based on the interaction formalism between SPH particles and MBD which is discussed thoroughly in the Sections 4.2 and 4.3, an interface and above outlined synchronization strategy are defined (see Section 4.4); Chapter 5 details the complete implementation.

## 4.2. The contact formalism – SPH-wall interaction

The objective of this section is the development of an interaction formalism between the SPH particles and the solid structures in the multibody system. In other words, we need a way to model the contact between the fluid and the solid, to describe the forces exerted on the structural parts by the fluid, while accounting for the essential boundary conditions (3.44) and (3.45) for a viscous fluid interacting with any solid, impervious structure (cf. Subsection 3.1.2.5).

From a strictly physical point of view, the interaction between a fluid and a boundary, defined by the surface of any solid body, would be described by a quantum-mechanical model on an atomic scale. In a quasi-classical, still atomic or molecular picture the quantum-mechanical description is replaced by the introduction of classical interaction forces or potentials, e.g. based on electrostatic dipole-dipole interaction, between the molecules of fluid and solid, respectively. We, however, regard both the fluid and the solid as idealized continua in a macroscopic description, spatially discretized to allow numerical treatment. Nevertheless, considering the above mentioned actual mechanism of interaction, it seems reasonable, not to say, natural, to base the contact between the fluid in SPH formulation and the components of the multibody system on force fields, or, to be exact, on force field densities defined over the surface of the solid structures.

Analogous to the approach in [43], two short-ranged force field surface densities,  $\mathbf{f}^{rep}(\mathbf{r}, \mathbf{r}')$  for the generation of repulsive forces and enforcement of the no-penetration condition (3.44), and  $\mathbf{f}^{visc}(\mathbf{r}, \mathbf{v}, \mathbf{r}', \mathbf{v}')$  to account for viscous forces, or, in other words, wall friction, and the corresponding no-slip boundary condition (3.45) shall be introduced. According to the sketch in Figure 4.1, the forces  $\mathbf{F}_i^{rep}$  and  $\mathbf{F}_i^{visc}$  on an SPH particle  $i$  at  $\mathbf{r}_i$  with velocity  $\mathbf{v}_i$  in the vicinity of a solid body and its surface denoted as  $S$  are given by the surface



**Figure 4.1.:** Interaction between the surface  $S$  of a solid structure and SPH particle  $i$  at position  $\mathbf{r}_i$ .

convolution integrals

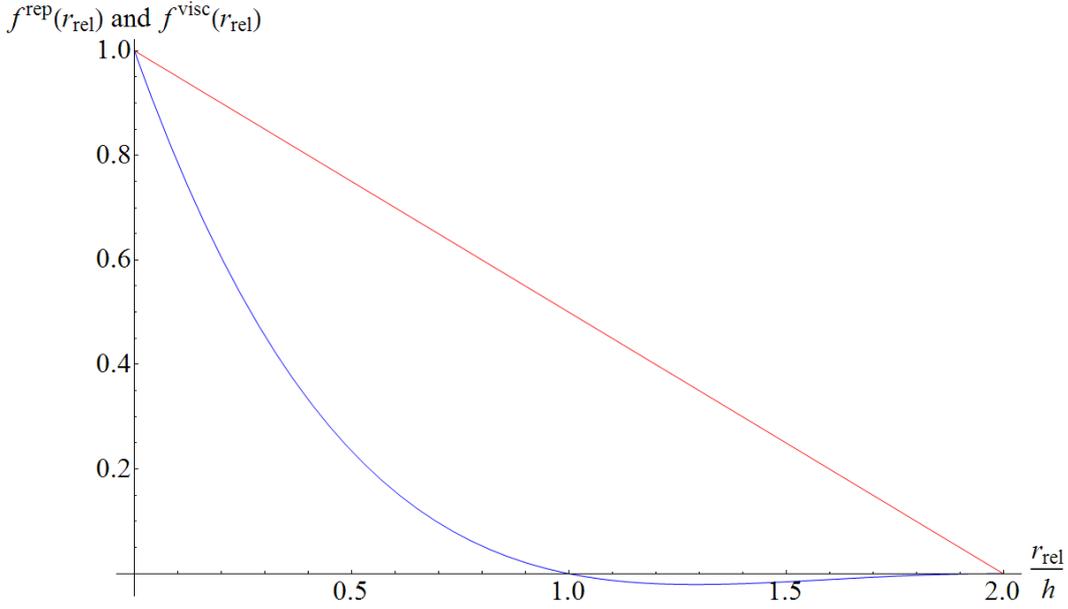
$$\begin{aligned}\mathbf{F}_i^{rep} &= \int_S \mathbf{f}^{rep}(\mathbf{r}_i, \mathbf{r}') dS' \\ \mathbf{F}_i^{visc} &= \int_S \mathbf{f}^{visc}(\mathbf{r}_i, \mathbf{v}_i, \mathbf{r}', \mathbf{v}') dS',\end{aligned}\quad (4.1)$$

with [43]

$$\begin{aligned}\mathbf{f}^{rep}(\mathbf{r}_i, \mathbf{r}') &= \mathbf{f}^{rep}(\mathbf{r}_{rel}) = \begin{cases} k \frac{(2h-r_{rel})^4 - (2h-r_0)^2 (2h-r_{rel})^2}{4h^2 r_0 (4h-r_0)} \frac{\mathbf{r}_{rel}}{r_{rel}} & 0 \leq r_{rel} \leq 2h \\ 0 & r_{rel} > 2h \end{cases} \\ \mathbf{f}^{visc}(\mathbf{r}_i, \mathbf{v}_i, \mathbf{r}', \mathbf{v}') &= \mathbf{f}^{visc}(\mathbf{r}_i, \mathbf{r}', \mathbf{v}_{rel}) = \begin{cases} -\frac{t}{2} \mathbf{v}_{rel} \frac{1}{\alpha} \Delta_{\mathbf{r}_i} W(\mathbf{r}', \mathbf{r}_i, h) & 0 \leq r_{rel} \leq 2h \\ 0 & r_{rel} > 2h \end{cases} \\ \mathbf{r}_{rel} &= \mathbf{r}_i - \mathbf{r}' \\ r_{rel} &= |\mathbf{r}_{rel}| \\ \mathbf{v}_{rel} &= \mathbf{v}_i - \mathbf{v}'.\end{aligned}\quad (4.2)$$

At that,  $\mathbf{r}'$  and  $\mathbf{v}'$  denote the position and velocity vector of a local point on the body surface  $S$ ; the cut-off, i.e. the interaction range of the force field densities is given by  $2h$ ,  $k$  and  $t$  are the respective scaling parameters for the repulsive and the viscous force, and  $0 < r_0 \leq 2h$  is the equilibrium distance for the former.  $W(\mathbf{r}', \mathbf{r}_i, h)$  represents an SPH smoothing kernel (cf. Subsection 3.1.2.3, equations (3.22) and (3.24)), and  $\alpha$  is another scaling factor (for re-normalization of the kernel). For better illustration, a sketch of  $\mathbf{f}^{rep}(\mathbf{r}, \mathbf{r}')$  and  $\mathbf{f}^{visc}(\mathbf{r}, \mathbf{v}, \mathbf{r}', \mathbf{v}')$  based on the spiky kernel (3.24) is given in Figure 4.2.

It shall be noted that the cut-off  $2h$  neither necessarily has to be identical for the repulsive and the viscous contribution, nor equal to the radius of the kernel support domain of the SPH simulation itself, however, it is reasonable to choose a value in the same order of magnitude; as explained below, the parallel between  $\mathbf{f}^{visc}$  and the SPH viscosity calculation



**Figure 4.2.:** Absolute values of the SPH-wall interaction force densities  $f^{rep}(r_{rel}) = |\mathbf{f}^{rep}(\mathbf{r}_{rel})|$  (blue line) and  $f^{visc}(r_{rel}) = |\mathbf{f}^{visc}(\mathbf{r}_{rel}, \mathbf{v}_{rel})|$  (red line) with  $|\mathbf{v}_{rel}| = 1$ , the spiky kernel (3.24) and the parameters  $k = t = 1$ ,  $r_0 = 1$ ,  $\alpha = 1/\alpha_d$  (cf. equation (3.25)).

for the fluid suggests to actually use equal values here.

The repulsive force is a central force acting along the line between a local point on the surface and the SPH particle position, consisting of a 4-th order repulsion term to keep SPH fluid particles from penetrating the boundary, and a 2-nd order attractive term to model adhesive effects; for  $r_0 = 2h$  the adhesive term vanishes. Of course, there is a wide range of possible and reasonable choices for the repulsive force densities, such as a Lennard-Jones like potential

$$\mathbf{f}^{rep}(\mathbf{r}_{rel}) = k \left( \left( \frac{r_0}{r_{rel}} \right)^{p_1} - \left( \frac{r_0}{r_{rel}} \right)^{p_2} \right) \frac{\mathbf{r}_{rel}}{r_{rel}}, \quad (4.3)$$

well-known from intermolecular interaction with typical values  $p_1 = 4$ ,  $p_2 = 2$  or  $p_1 = 12$ ,  $p_2 = 6$  and, for instance, suggested and implemented in [16], which corresponds exactly to above discussed semi-classical perspective on an atomic level.

The viscous force, on the other hand, is directed against the relative velocity between the SPH particles and the local surface points for  $\Delta_{r_i} W(\mathbf{r}', \mathbf{r}_i, h) > 0$ , which clearly approximates the no-slip condition (3.45). With the proportionality to the Laplacian of the smoothing kernel and the velocity difference it resembles the SPH approximation of the viscous term – the real one, not the artificial viscosity – in case of incompressible fluids which in turn is proportional to the Laplacian of the velocity field (note that derivatives of functions are transferred to derivatives of the smoothing kernel in the SPH formalism, as it has been discussed in Subsection 3.1.2.1; cf. also Subsection 3.1.2.4). Since the in-

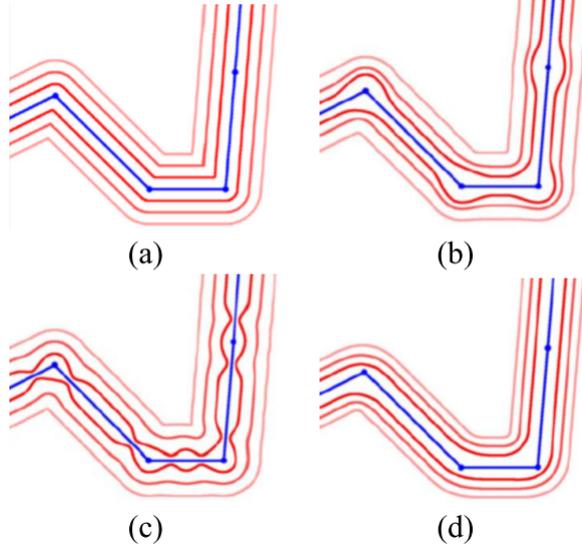
egration has to be performed numerically, i.e. based on certain discrete sampling points on the boundary (cf. Section 4.3), one could actually identify those boundary points as another type of SPH particles and include their contribution to the viscous force terms in the original SPH summation (cf. equations (3.32) or (3.35),(3.36)); this would then be in complete analogy with the boundary treatment via fixed boundary particles proposed by Monaghan [16].

Now, equations (4.1) and (4.2) define the forces on any SPH particle due to interaction with the surface  $S$  of any body in contact with the fluid; of course, the corresponding counterforce reacts upon that body according to Newton’s second law. More details on the balance of forces and moments are given in the next section.

The approach via convolution integrals lies somewhere inbetween a boundary particle treatment, i.e. another type of SPH particles placed and fixed along the boundaries and included in all SPH summations, and a force-field approach based on some kind of measure of distance between the SPH fluid particles and the boundaries. However, there are critical advantages over both of them:

In the first case, a known problem is boundary penetration; this can happen because of “holes” in the repulsive potential wall (created by the boundary particles) due to the short interaction range. Even though with the approach at hand the integrals have to be computed numerically and thus are reduced to sampling-point-wise contributions along the boundary (see Section 4.3) – which, from this point of view, makes it very similar to the method of boundary particles – refinement here, in contrast to the latter (consider the fixed boundary particles), comes into play very naturally, by just refining the discrete boundary elements, or using a higher-order integration scheme (cf. Section 4.3).

In the second case, the forces between the fluid particles and the boundaries depend on a measure of relative distance  $d$ , for example, the minimum normal distance of a given particle to the surface, or in the discretized case, the surface elements, respectively. This comes with two problems – firstly, in case of complex geometries the computation of  $d$  might be difficult, and secondly,  $d$  usually is only  $C^0$ -continuous, and so is the force field. However, at any edge or vertex – and there are a lot of those for geometrically discretized surfaces, e.g. by a finite element triangular mesh in 3D or connected line segments in 2D, which have to be used for almost any geometry (and body) – the derivative of  $d$  and the force field are discontinuous, resulting in artifacts such as the “cooking” of SPH particles in concave regions (i.e. instable or jittery motion of the particles) and reducing the general stability of the simulation [43]. Of course, by weighting of the distances to adjacent surface elements or some other smoothening techniques, these effects can be reduced, but the iso-lines of the force field are still bulgy in the vicinity of vertices. The convolution integrals used here, on the other hand, yield smooth force iso-lines everywhere, provided that the numerical integration is performed sufficiently accurate; see Figure 4.3 for illustration.



**Figure 4.3.:** (a) Iso-surfaces of the Euclidean distance field of a piecewise linear curve (blue) with discontinuous first derivatives near concavities. (b) Weighted sums yield smooth iso-surfaces with bulges. (c) Normalization does not remove the artifact. (d) Convolution yields bulge-free smooth iso-surfaces. Source: [43].

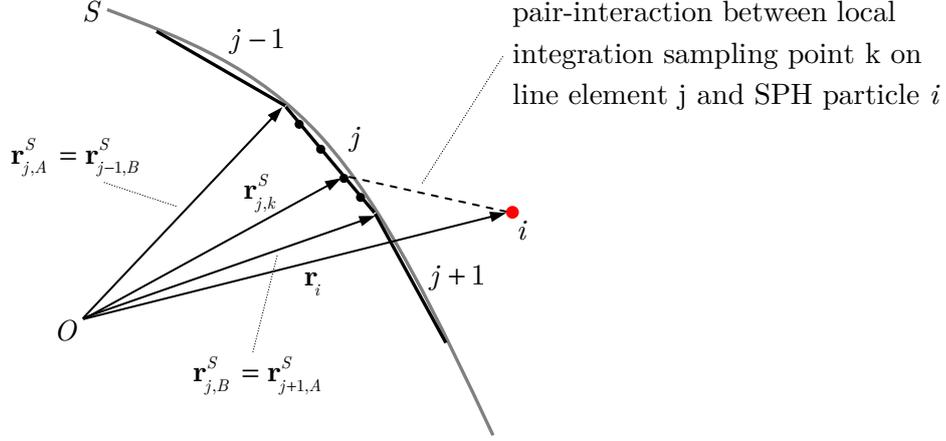
### 4.3. Discretized force calculation and mechanical equilibrium

Depending on the spatial discretization used in the model for the solid components of the MBS, the surface of those components is also discretized, or analytically defined (e.g. in case of structural finite elements), and, in any case, can be represented by some kind of surface mesh. In case of two spatial dimensions, an arbitrary surface can be defined as a sequence of connected line elements, for 3D we usually have a surface mesh consisting of triangular and/or quadrilateral elements. In the following, we shall focus on the 2D-case, since this also was the one actually implemented within the scope of this master thesis, and moreover, the 3D-case can be treated quite similarly.

For a 2D surface  $S$  discretized by  $N_S$  line segments, a numerical integration scheme for the computation of a surface integral (4.1) in general (see also [39]) is based on sampling points  $\mathbf{r}_{j,1}^S, \dots, \mathbf{r}_{j,M_j}^S$  along the line and corresponding velocities  $\mathbf{v}_{j,1}^S, \dots, \mathbf{v}_{j,M_j}^S$  for each line element  $j$  of length  $l_j$  (see Figure 4.4) with respective weights  $w_1, \dots, w_{M_j}$ , and is given by

$$\begin{aligned} \mathbf{F}_i^{rep} &\approx \sum_{j=1}^{N_S} \frac{1}{2} l_j \sum_{k=1}^{M_j} w_k \mathbf{f}^{rep}(\mathbf{r}_i, \mathbf{r}_{j,k}^S) \\ \mathbf{F}_i^{visc} &\approx \sum_{j=1}^{N_S} \frac{1}{2} l_j \sum_{k=1}^{M_j} w_k \mathbf{f}^{visc}(\mathbf{r}_i, \mathbf{v}_i, \mathbf{r}_{j,k}^S, \mathbf{v}_{j,k}^S). \end{aligned} \quad (4.4)$$

Note that the line segments are defined via their two vertices, denoted as  $\mathbf{r}_{j,A}^S$  and  $\mathbf{r}_{j,B}^S$ ,



**Figure 4.4.:** Sketch of the 2D surface discretization via connected line segments  $j = 1 \dots N_S$  and the numerical surface integration of the SPH-wall interaction based on local sampling points  $\mathbf{r}_{j,k}^S$ .

and the corresponding velocities  $\mathbf{v}_{j,A}^S$  and  $\mathbf{v}_{j,B}^S$  for the  $j$ -th element; based on that data, the sampling points  $\mathbf{r}_{j,k}^S$  and velocities  $\mathbf{v}_{j,k}^S$  are calculated according to the chosen integration scheme, using linear spatial interpolation for both with an interpolation factor  $g_{j,k}^S$  normalized by the length of the corresponding line element:

$$\begin{aligned} \mathbf{r}_{j,k}^S &= \mathbf{r}_{j,A}^S + (\mathbf{r}_{j,B}^S - \mathbf{r}_{j,A}^S) g_{j,k}^S = (1 - g_{j,k}^S) \mathbf{r}_{j,A}^S + g_{j,k}^S \mathbf{r}_{j,B}^S \\ \mathbf{v}_{j,k}^S &= \mathbf{v}_{j,A}^S + (\mathbf{v}_{j,B}^S - \mathbf{v}_{j,A}^S) g_{j,k}^S = (1 - g_{j,k}^S) \mathbf{v}_{j,A}^S + g_{j,k}^S \mathbf{v}_{j,B}^S \end{aligned} \quad (4.5)$$

with

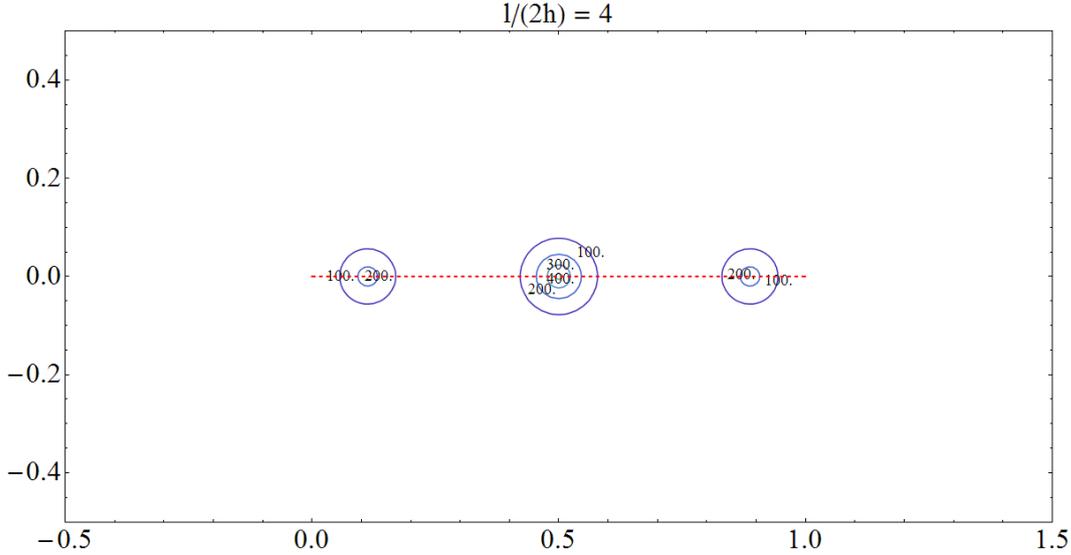
$$g_{j,k}^S = \frac{|\mathbf{r}_{j,k}^S - \mathbf{r}_{j,A}^S|}{|\mathbf{r}_{j,B}^S - \mathbf{r}_{j,A}^S|} \in [0, 1]. \quad (4.6)$$

As it has already been mentioned in the previous section, due to the short-ranged force fields (range  $2h$ , cf. equation (4.2)) and the discrete point-wise numerical evaluation of the surface convolution integrals (4.1), the distances  $d$  between the local sampling points must be sufficiently small. The maximum distance is given by  $d_{max} = 4h$ , however, depending on the characteristics of the force fields, it typically should be significantly lower. Otherwise, the boundary would contain “holes” between any pair of sampling points with  $d > d_{max}$ , since an SPH particle passing, for instance, exactly midway between such pairs is not within interaction range anymore, and thus is not subject to any kind of force (or just a very small force) due to the boundary interaction; we could say that, in such cases, the particle does not “see” any boundary, and hence, the no-penetration condition (3.44) is violated.

For a given line segment with length  $l$ , consequently, at least  $N_p = l/d_{max}$  local points are necessary, which can be accomplished either by using an integration scheme based

$k$	$g_{j,k}^S$	$w_k$
1	$\frac{1}{2}(1 - \sqrt{3/5})$	5/9
2	$\frac{1}{2}$	8/9
3	$\frac{1}{2}(1 + \sqrt{3/5})$	5/9

**Table 4.1.:** Sampling points and weights for 3-point Legendre-Gauss quadrature [44] over the interval  $[0, 1]$ .



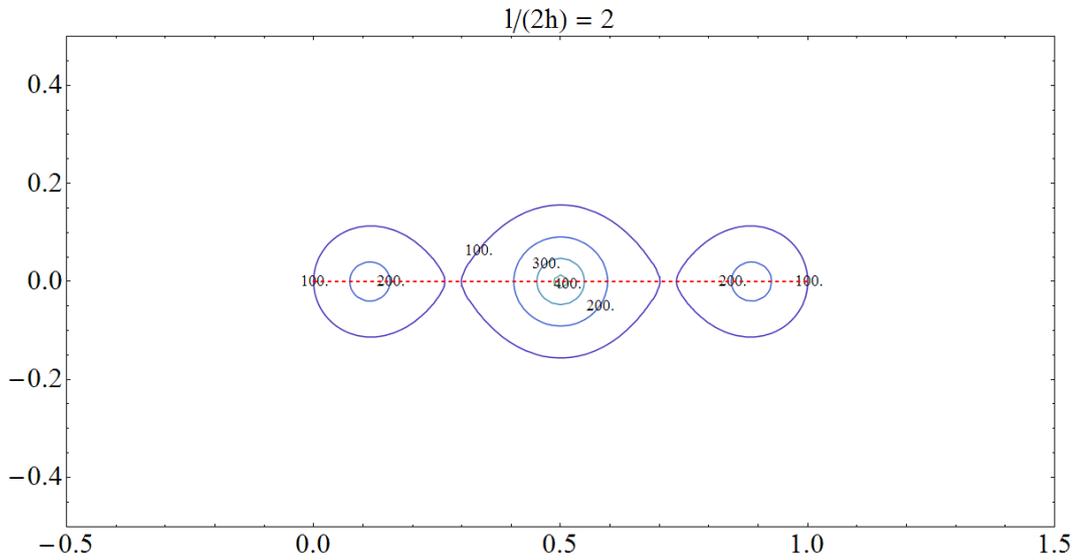
**Figure 4.5.:** Iso-lines of  $|\mathbf{F}_i^{rep}|$  according to equation (4.4) and (4.2) with parameters  $k = 1000$  and  $r_0 = 2h$  around a line element (red dashed) using 3-point Legendre-Gauss quadrature, with various length/cut-off ratios  $l/(2h) = 4$ .

on  $N_{int} > N_p$  sampling points, or, if  $N_{int} < N_p$ , by appropriate mesh-refinement, i.e. here, by subdivision of the considered line segment in subelements with lengths  $l_{sub} < d_{max}N_{int}$ . Using adaptive mesh-refinement is preferable to the application of different numerical integration schemes, or, different orders of one specific integration algorithm on the un-refined mesh, since the refinement can be performed easily locally to an arbitrary level of resolution, and all elements are treated equally from a numerical point of view. To get an idea of an appropriate refinement resolution in case of a 3-point Legendre-Gauss quadrature ( $N_{int} = 3$ ) for numerical computation of the line integrals (cf. Table 4.1), Figures 4.5 to 4.8 show the force field iso-lines of  $|\mathbf{F}_i^{rep}|$  according to equation (4.4) around one line element with various length/cut-off ratios  $l/(2h)$ .

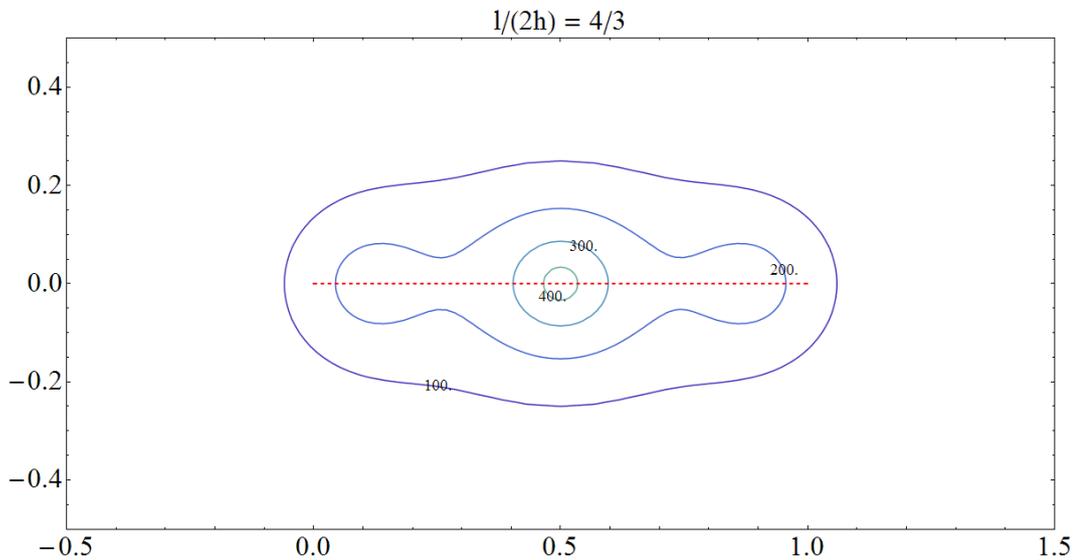
Taking above parameters and integration scheme, in order to get a closed force iso-line around the line element with at least 20% of the maximum repulsive force, i.e. here  $0.2k = 200$ , apparently the length of the line element should be approximately equal to the force field range,

$$l \approx 2h, \quad (4.7)$$

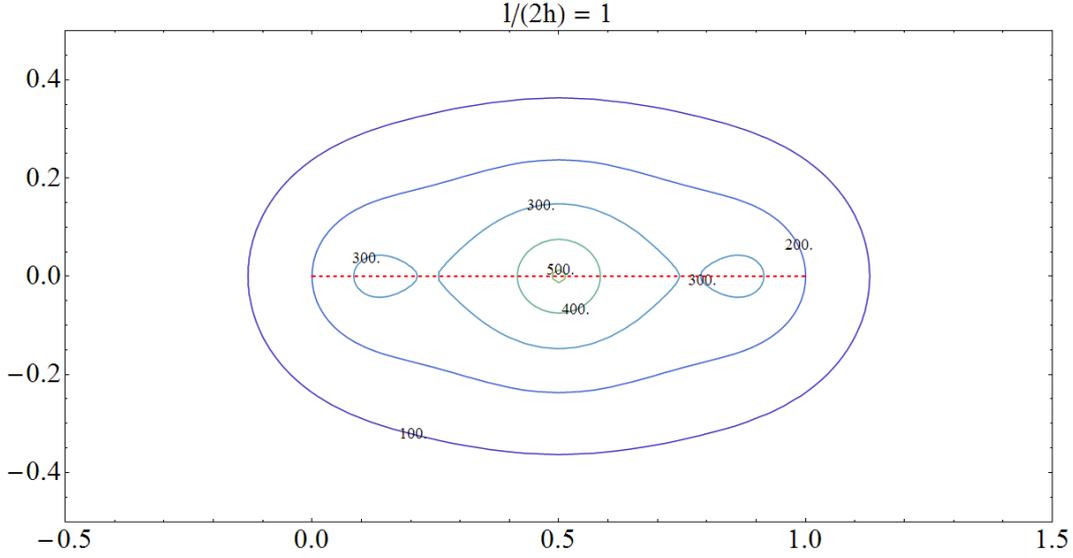
which is by a factor 6 smaller than above mentioned maximum length  $l_{sub}$ . It is important



**Figure 4.6.:** Iso-lines of  $|\mathbf{F}_i^{rep}|$  according to equation (4.4) and (4.2) with parameters  $k = 1000$  and  $r_0 = 2h$  around a line element (red dashed) using 3-point Legendre-Gauss quadrature, with various length/cut-off ratios  $l/(2h) = 2$ .



**Figure 4.7.:** Iso-lines of  $|\mathbf{F}_i^{rep}|$  according to equation (4.4) and (4.2) with parameters  $k = 1000$  and  $r_0 = 2h$  around a line element (red dashed) using 3-point Legendre-Gauss quadrature, with various length/cut-off ratios  $l/(2h) = 4/3$ .



**Figure 4.8.:** Iso-lines of  $|\mathbf{F}_i^{rep}|$  according to equation (4.4) and (4.2) with parameters  $k = 1000$  and  $r_0 = 2h$  around a line element (red dashed) using 3-point Legendre-Gauss quadrature, with various length/cut-off ratios  $l/(2h) = 1$ .

to note that only closed iso-lines represent a closed surface (without “holes”), and thus are the critical point to look at in order to get impervious boundaries. Consequently, the force scaling factor  $k$  must be chosen sufficiently high, and the lengths of the line segments need to be sufficiently small. See Subsection 5.5.2 for details on the implementation regarding the issue of discrete force calculation and adaptive mesh refinement; see also Section 6.2 for some notes on the choice of the parameters of the fluid-structure interaction forces.

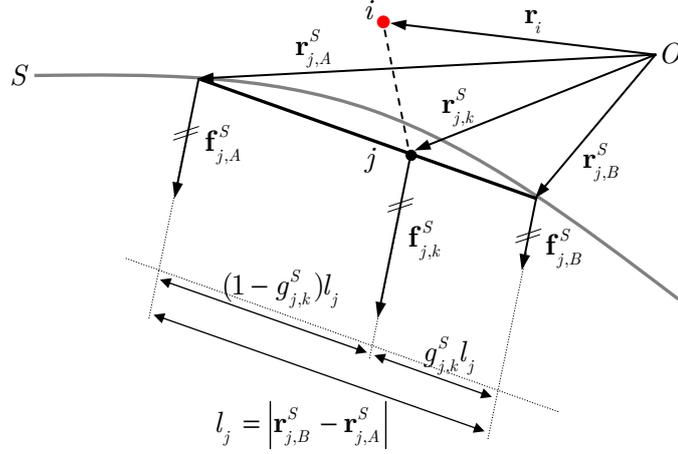
As a result from the surface discretization, i.e. the surface mesh and the numerical integration (4.4), we obtain for every surface element a number of point-wise pair contributions from the interaction of any SPH fluid particle with the sampling points on the boundary. The forces on the fluid particles are directly given by (4.4), however, on the mechanical side every single counterforce contribution needs to be extrapolated back onto the corresponding vertices for each boundary element, since it is the vertices, and not the local sampling points, that define the surface elements, and thus, represent the respective component of the MBS.

Now, by the sketch in Figure 4.9 let us consider line segment  $j$ , subject to a force

$$\mathbf{f}_{j,k}^S = -\frac{1}{2}l_j w_k \left( \mathbf{f}^{rep}(\mathbf{r}_i, \mathbf{r}_{j,k}^S) + \mathbf{f}^{visc}(\mathbf{r}_i, \mathbf{v}_i, \mathbf{r}_{j,k}^S, \mathbf{v}_{j,k}^S) \right) \quad (4.8)$$

exerted by the  $i$ -th SPH fluid particle and acting in the local point  $\mathbf{r}_{j,k}^S$  given by  $g_{j,k}^S$ . Based on this force, we need to determine the “equivalent” forces  $\mathbf{f}_{j,A}^S$  and  $\mathbf{f}_{j,B}^S$  acting on the vertices; to that end, assuming that the line segment is rigid, the balance of forces,

$$\mathbf{f}_{j,A}^S + \mathbf{f}_{j,B}^S = \mathbf{f}_{j,k}^S, \quad (4.9)$$



**Figure 4.9.:** Sketch of surface line segment  $j$  (length  $l_j$ ) and the force  $\mathbf{f}_{j,k}^S$  acting on the local sampling point  $\mathbf{r}_{j,k}^S$  due to the interaction with the SPH particle  $i$ , with the corresponding extrapolated forces  $\mathbf{f}_{j,A}^S$  and  $\mathbf{f}_{j,B}^S$  acting on the vertices, obtained from linear weighting using  $g_{j,k}^S$ .

as well as the balance of moments with respect to any reference point  $\mathbf{r}_{ref}$ ,

$$\left(\mathbf{r}_{j,A}^S - \mathbf{r}_{ref}\right) \times \mathbf{f}_{j,A}^S + \left(\mathbf{r}_{j,B}^S - \mathbf{r}_{ref}\right) \times \mathbf{f}_{j,B}^S = \left(\mathbf{r}_{j,k}^S - \mathbf{r}_{ref}\right) \times \mathbf{f}_{j,k}^S \quad (4.10)$$

must be satisfied; note that the latter does not depend on  $\mathbf{r}_{ref}$ , if the balance of forces is satisfied. However, this system of equations has no unique solution – it is underdetermined – which is clear since we have assumed that the body associated with the line segment under consideration is rigid, and hence there are infinitely many possibilities to choose  $\mathbf{f}_{j,A}^S$  and  $\mathbf{f}_{j,B}^S$  such that the resulting force and moment is equal to the original induced by  $\mathbf{f}_{j,k}^S$ . For further clarification of this fact just think of any additional force-counterforce pair acting in opposite directions parallel to the line element on its two vertices; this changes  $\mathbf{f}_{j,A}^S$  and  $\mathbf{f}_{j,B}^S$ , but leaves the total force and moment unchanged. Thus, a reasonable extrapolation (or interpolation) scheme satisfying equations (4.9) and (4.10) is necessary to establish a relation between the forces in the vertices and the force acting on the local sampling point on the line element.

Here, we use linear weighting by the distance between the sampling point and the respective vertex, which is given by the factor  $g_{j,k}^S$  from our linear spatial interpolation (4.5) by

$$\begin{aligned} \mathbf{f}_{j,A}^S &= (1 - g_{j,k}^S) \mathbf{f}_{j,k}^S \\ \mathbf{f}_{j,B}^S &= g_{j,k}^S \mathbf{f}_{j,k}^S. \end{aligned} \quad (4.11)$$

It is easy to show that this scheme of force redistribution satisfies the balance of forces and moments; it should be noted that exactly the same result is obtained from a direct solution of (4.9) and (4.10) without any assumptions if only the force components orthogonal to the

surface element are considered, in which case the balance equations have a unique solution (in 2D, as well as 3D). Thus, the counterforces corresponding to  $\mathbf{F}_i^{rep}$  and  $\mathbf{F}_i^{visc}$ , acting on an element  $j$  due to interaction with the SPH fluid particle  $i$ , are given by

$$\begin{aligned}\mathbf{f}_{j,A}^S &= \sum_{k=1}^{M_j} (1 - g_{j,k}^S) \mathbf{f}_{j,k}^S \\ \mathbf{f}_{j,B}^S &= \sum_{k=1}^{M_j} g_{j,k}^S \mathbf{f}_{j,k}^S.\end{aligned}\tag{4.12}$$

As a side note, in 3D the procedure for integration is analogous, except that now 3D surface elements (triangles and/or quadrilaterals), defined by 3 or 4 vertices and their velocities, have to be considered. The numerical surface integration then once more is just a point-based procedure (e.g. 7-point Gauss integration over a triangular area), and interpolation for velocities is again done linearly. Using the linear weighting scheme for the force redistribution of the contributions of the sampling points to the actual surface element vertices, the balance equations for force and momentum are also satisfied. In case of triangular elements, both the numerical integration, linear interpolation, as well as the force redistribution is best done by the parametrization of the triangle areas in barycentric coordinates, which take on the role of above factors  $(1 - g_{j,k}^S)$  and  $g_{j,k}^S$ .

Further information on the implementation of the contact formalism is given in Subsection 5.5.2. Refer to Subsection 5.4.2 for some details about how the resulting forces (4.12) are actually applied on the components of the multibody system.

#### 4.4. Interface and synchronization

In the previous sections the main idea of the coupling concept as well as the model and numerical computation of the SPH-wall interaction have been discussed. Based on the latter, we can now proceed to develop an interface design. As already mentioned, an interface comprises a specific set of data as well as communication routines, and is used on both sides of the coupled system. The crucial task of the interface is the establishment and definition of a link between the two sides, based on mutually used and exchanged data.

As to the latter, on the LIGGGHTS/SPH side the only data necessary for the contact formalism is the geometrical and kinematical information about any surface in contact with the fluid, which is, firstly, given by the surface mesh – attached to and possibly defined by the associated body in the MBS – consisting of a set of vertices based on which the surface elements are defined, and secondly, the respective velocities. On the other hand, the only information needed HOTINT/MBD-sided are the forces acting on those vertices, or, more precisely, on the corresponding MBS component at those vertices, due to the interaction with the fluid.

Thus, the minimal data set necessary for the coupling approach at hand consists of

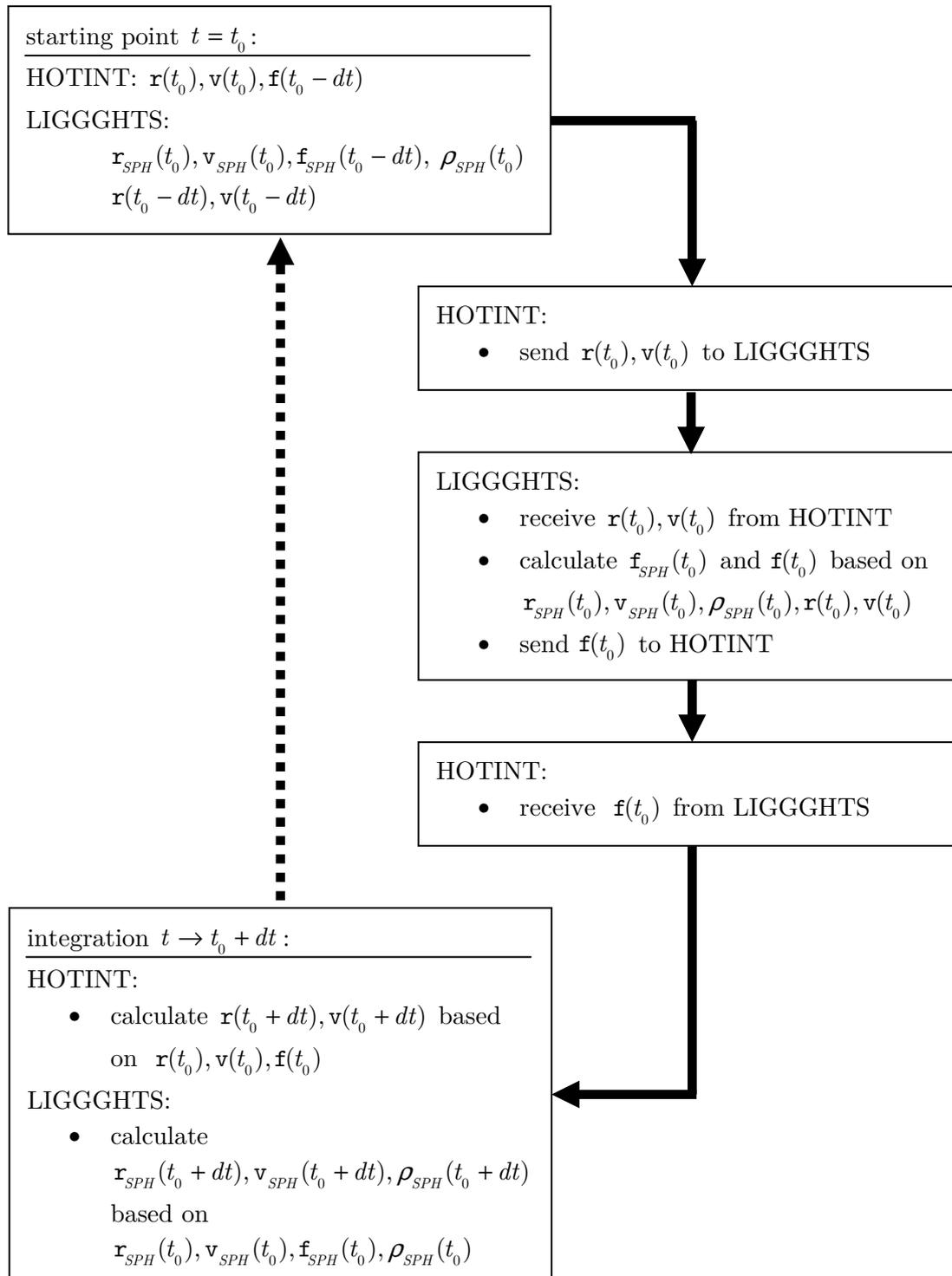
- three arrays containing the positions and velocities of, and the forces on boundary vertices, which shall be denoted as  $\mathbf{r}$ ,  $\mathbf{v}$  and  $\mathbf{f}$ , respectively,
- and an array containing pairs of numbers (in 2D) referencing the points in  $\mathbf{r}$ , defining the surface (line) elements via their vertices (3D-case: sets of 3 or 4 numbers, for triangular and quadrilateral elements), in the following referred to as  $\mathbf{e1}$ .

Additionally, for reasons of the problem set-up, data processing and evaluation as well as visualization, fluid data – the position, velocity, density and pressure of the SPH particles – can be added to the data set of the interface.

Concerning the communication, consequently, routines for the exchange of  $\mathbf{r}$  and  $\mathbf{v}$  from HOTINT to LIGGGHTS, and  $\mathbf{f}$  the other way around, as well as the transfer of SPH data from LIGGGHTS to HOTINT must be included; as already mentioned in Subsection 4.1.2, the data transfer shall be done via TCP/IP. More details on this and a discussion of the actually implemented interface are given in Section 5.2.

At this point the flexibility of the developed strategy should be pointed out: On the fluid side, it not only does not matter at all which kind of structural component lies under a surface mesh, but any surface element can be treated also completely independently from the underlying bodies. In other words, the sequence in which the elements are stored in  $\mathbf{e1}$ , as well as the order in which they are referenced and evaluated in the force computation, can be chosen freely. Furthermore, it should be noted that there is no need to define “inside” and “outside” of a given geometry – any kind of configuration, e.g. open or closed geometry, empty or filled with fluid, moving around in empty space or in another reservoir filled with fluid, is treated in the same universal way. On the other hand, the only thing that “remains” from the fluid on the side of the multibody system are additional forces on the components in certain points. Because of the above and the fact that the contact computation effectively is a pair evaluation between particles – any pair consisting of one SPH particle and a local sampling point on a boundary element – the approach is fully compatible with (adaptive) mesh-refinement and arbitrarily moving and/or deforming, even topologically changing geometry.

Now, we have made all necessary preparations – one could say, put together a “theoretical tool box” – to contemplate the implementation of the coupling of HOTINT and LIGGGHTS, except for one final question: How can the two simulators actually be connected and synchronized? The crucial point to this end is, as it has already been mentioned in Subsection 4.1.2, that the coupling must be done within the time-stepping. It should be pointed out that both sides use very different integration schemes – HOTINT a high-order implicit algorithm with adaptive time steps (cf. Subsection 2.2.3 and 2.1.3.3), LIGGGHTS an explicit two-stage second-order integrator (cf. Subsection 3.2.3). Again, the HOTINT-side – as server – is the one to determine the size of the time steps, which then



**Figure 4.10.:** Scheme of the program flow of one time step, starting at a time  $t = t_0$  and integrating to  $t = t_0 + dt$ . Here,  $\mathbf{r}(t), \mathbf{v}(t), \mathbf{f}(t)$  designate the arrays containing positions and velocities of, and forces on the vertices (boundary points) defining the boundary meshes of the multibody system components; correspondingly,  $\mathbf{r}_{SPH}(t), \mathbf{v}_{SPH}(t), \mathbf{f}_{SPH}(t), \rho_{SPH}(t)$  denote the arrays for positions, velocities, forces and densities associated with the SPH fluid particles.  $\mathbf{f}(t)$  is assumed to be constant over the stages (during the “substeps”) of the implicit integration scheme on the multibody side.

are communicated with the LIGGGHTS-side; thus, the time steps for the fluid simulation are in-sync with the respective step size in the integration routines of HOTINT, which, for the fluid side of course has nothing to do with real adaptive step size control. Note that the range of the size of the time step must be defined appropriately by the user, effectively meaning that it has to be adjusted according to the requirements of the explicit integration schemes on the fluid side; cf. Section 6.2 for more details. While HOTINT computes one time step of size  $dt$  via several intermediate substeps according to the stages of the implicit integration scheme (cf. Section 2.1.3.3), LIGGGHTS performs one integration step of equal size  $dt$  explicitly. At that, we assume a constant force due to the contact with the fluid for all stages of the implicit integrator on the structural side. With a flow diagram in Figure 4.10, illustrating how the coupling of the two simulators within the time-stepping based on the interface outlined above works, this discussion shall be concluded.

It should be noted that effectively only  $\mathbf{r}$ ,  $\mathbf{v}$  and  $\mathbf{f}$  – in fact, only the dynamic parts of those arrays (cf. Section 5.2) – must be exchanged in every single time step, given that  $\mathbf{e1}$  does not change during time. The latter stays constant as long as the surface mesh itself, or, more precisely, the association of the surface elements with certain corresponding boundary points (vertices) do not change; because that is usually true and the fact that the mesh-refinement is only done locally on the LIGGGHTS-side,  $\mathbf{e1}$  only needs to be initialized (i.e. transferred from the HOTINT- to the LIGGGHTS-side before the first time step). This can be significant, especially in case of large static boundary geometries; the transfer of everything else, SPH particle data, for example, is optional. Confer also Section 5.3.2 for issues concerning the performance of the TCP/IP data transfer.

As an important remark, the synchronization of the coupled program flow comes more or less automatically, not to say, inherently with the TCP/IP routines for data exchange and communication, because the `recv()`-routines – those functions responsible to receive and process TCP/IP packets over some specified network IP address and port – in the TCP/IP socket APIs used in the implementation are so-called “blocking calls” by default. This means, if the program execution on either side reaches a point where data from the respective other side is needed, it just stops there and waits until that information is received (unless specified otherwise), thus accounting for synchroniation; see the following section for more details on the implementation.

# 5. Coupling of HOTINT MBD and LIGGGHTS SPH – Implementation

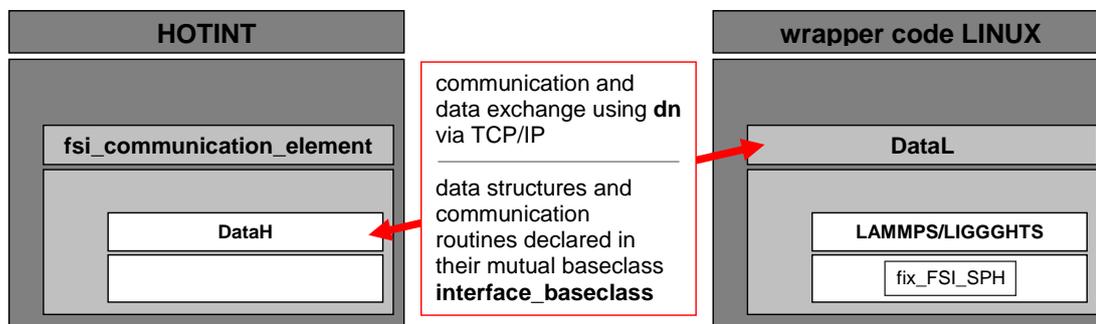
## 5.1. Introduction and overview

The following sections shall give an insight into the implementation of the coupling of HOTINT MBD and LIGGGHTS SPH, the theoretical basis of which has been discussed in the previous chapter. Since both sides are based on C++, the implementation naturally was also written in C++; it should be pointed out that every fragment of code of this implementation was written and developed from scratch by myself, using [45], as well as [46] and [47] as C++ standard references, along with [53] and [54] concerning all network programming issues.

HOTINT is a Microsoft Visual Studio project and was compiled and run in the 32-bit version on the platform MS Windows 7 (64 bit); thus, the code development on this side was done in Visual Studio 2005 Professional, obtained from the Johannes Kepler University Linz via the MSDN Academic Alliance [48]. The program package LIGGGHTS, on the other hand – as an open source project – was retrieved from [22], and compiled and run in 64 bit on the platform LINUX Fedora 15, for the simulations in Chapter 6 in fact on the same machine as HOTINT, with Windows as host platform, and LINUX running via the multi-core supporting emulator Oracle VM VirtualBox (version 4.1.4; cf. [49], as LIGGGHTS freely distributed under the terms of the GNU General Public License (GPL)). In that case, the TCP/IP communication was effectively done via a 1-GBit/s virtual network adapter (VirtualBox Host-Only Ethernet Adapter) simulated by the virtual machine. The code and program development on the LINUX side was performed using the editor “gedit” and command-line based compilation via makefiles, utilizing a “g++” (or “gcc”) compiler version invoked by the MPI wrapper compiler “mpic++” for MPI-parallelized applications. The use of the latter is convenient, since it automatically includes all linking dependencies and header files necessary to compile an MPI application.

In the following, the implementation of the coupling formalism, in total consisting of seven components the file names of which (headers and source code) are listed below, shall be outlined; a sketch of the (very) basic structure of the coupled application, along with a brief description about how the whole system works in principle, is given in Figure 5.1. For detailed discussion the reader is referred to the following sections.

- **dn** (cf. Section 5.3): contains routines for platform and architecture-independent exchange of floating point (double precision) numbers
- **interface\_baseclass** (cf. Section 5.2): contains the base class `interface_baseclass` defining the interface (data structure and communication)
- **exchange\_class\_windows** (cf. Subsection 5.4.1): includes the central class `DataH` for TCP/IP server functionality (Windows), problem set-up, data initialization, communication, and some auxiliary classes
- **exchange\_class\_linux** (cf. Subsection 5.5.1): contains the central class `DataL` accounting for TCP/IP client functionality (LINUX), problem set-up and initialization of LAMMPS/LIGGGHTS instances (used as static libraries), communication
- **fsi\_communication\_element** (cf. Subsection 5.4.2): contains the class `fsi_communication_element` for the HOTINT-sided implementation, connection between `DataH` and the framework of HOTINT
- **fix\_FSI\_SPH** (cf. Subsection 5.5.2): contains the class `fix_FSI_SPH` for the LIGGGHTS implementation of the contact formalism, MPI parallel
- **wrapper code LINUX** (cf. Subsection 5.5.3): wrapper program for instantiation and initialization of a `DataL` object and the MPI environment, actual client application



**Figure 5.1.:** Sketch of the components and their role in the coupled application: HOTINT runs on the Windows side as main application, and includes the special element `fsi_communication_element` (cf. also Subsection 2.2.3) in the model files which, in turn, makes use of the functionality of the class `DataH` defined in “exchange\_class\_windows” (problem set-up, initialization, communication). On the LINUX side, the “wrapper code LINUX” runs as client and main application, utilizing the functionality of the class `DataL` defined in “exchange\_class\_linux” (instantiation and initialization of LAMMPS/LIGGGHTS, communication) time step-wise in a loop controlled by the server side. At that, LAMMPS/LIGGGHTS is included as a static library, and itself uses `fix_FSI_SPH` for the computation of the SPH-wall interaction. Communication and data transfer is done via TCP/IP, effectively between the classes `DataH` and `DataL`, based on the interface defined via their mutual base class `interface_baseclass` and using the routines in “dn” for the exchange of double precision numbers.

The core of the implementation is formed by the classes `DataH` and `DataL` defined in “exchange\_class\_windows” and “exchange\_class\_linux”, respectively, both derived from their mutual base class `interface_baseclass` which determines the interface (cf. also Section 4.4). Here, `DataH` offers the server functionality and accounts for the problem set-up on both sides and initialization of the interface data, as well as the exchange of information with the client, including command sequences for control of the client application; `DataL`, on the other hand, is the counterpiece to the latter, provides the corresponding client functionality, moreover, integrates and controls LIGGGHTS via the LAMMPS baseclass, and, of course, also accounts for transfer of interface data.

On the Windows / server side, HOTINT is the main application. As it has already been discussed in Section 2.2, every model set-up here is assembled on the basis of some kind of elements (cf. Figure 2.3), all of which have their own specific functionality invoked at certain points in the program execution, for example in the different stages of one timestep. Now, the additional “special” element `fsi_communication_element` serves as connector between the framework of HOTINT and `DataH`. Via this element, the interface data and communication functionality of `DataH` are accessed and used in every time step, for example for the actual initialization after the HOTINT problem set-up before the first time step, or the access to and application of the forces acting on the boundary points to the corresponding elements (cf. also Sections 4.3 and 4.4).

On the LINUX / client side, the “wrapper code LINUX” runs in a loop – after initialization of the MPI environment – communicating time step-wise with the server, i.e. effectively with a `DataH` object, and utilizing the functionality of `DataL` (e.g. exchange of interface data). LIGGGHTS can be built as an executable, as well as a static library, and it is implemented in the form of the latter here. `DataL`, instantiated by the wrapper code, in turn creates an instance of the LAMMPS top-level base class on each MPI process, and thus integrates the LIGGGHTS environment. In that base class, a pointer is included to the instance of `DataL` which the LAMMPS base class was instantiated by. Hence, via this pointer, any functionality of `DataL`, for example the access to the data defining the positions and velocities of the vertices of the boundary meshes, can be used within the framework of LIGGGHTS, which is done in the fix `fix_FSI_SPH` to calculate the SPH-wall interaction. Note: A “fix” in LAMMPS/LIGGGHTS is a routine which is called at some point in every time step; see Subsection 5.5.2 for more information.

Finally, a few notes concerning the parallelization shall be mentioned. Everything on the LINUX side, as LAMMPS/LIGGGHTS itself, is written fully parallel using the MPI environment. At that, MPI stands for “message-passing interface” and is a standardized and portable message-passing system developed for parallelization of applications on parallel computers. Initially, it was designed particularly for distributed memory machines, since the parallelization is based on different processes – each associated with one or a set of processors with an own (individual) memory space – and explicit mutual communication.

For more information on MPI, refer for example to [50, 51], which have also been used as standard references in this work. As to the coupled application here, of course only one – the first – process is used for communication; all others are just “listeners” to this first process. In the current implementation, however, in contrast to the data LIGGGHTS is working with, the complete interface data (except for the force arrays) exists as identical copies in each memory space and is kept synchronized during the whole runtime. A shared memory approach, which is based on a common memory space for all processes and is also supported in a certain way by MPI since the 2.0 standard, or a data distribution model with the distributed memory approach where any process locally only has that part of data which is actually needed could additionally increase efficiency here.

## 5.2. Interface

The interface, as it has been discussed in the Section 4.4, is defined by the data and the corresponding exchange routines necessary for the coupling of both sides, and implemented in form of the class `interface_baseclass.h`, used on both sides as base class for the core components `DataH` and `DataL` in “exchange\_class\_windows” and “exchange\_class\_linux”, respectively.

The data set is given by

- `unsigned short dim` specifying the dimensionality of the problem,
- the double arrays `double* _r, _v, _f` containing positions, velocities, and forces of the vertices of the surface meshes (in the following also referred to as boundary points),
- the total number of boundary points `unsigned int n`, the number of static boundary points `int nstat`,
- the integer array `unsigned int* _e1` containing a set of pairs of numbers referencing points in `_r` and defining vertices of the surface elements (line segments) in 2D,
- the number of surface elements `unsigned int nel`,
- the time step size `double dt`,
- the double arrays `double* _rSPH, _vSPH, _rohSPH` containing SPH particle data – positions, velocities, and densities, respectively, for a
- total number `unsigned int nSPH` of SPH particles, and finally,
- the parameters `int refinement_option` specifying how the adaptive refinement of the surface meshes should be performed (see Table 5.1), along with
- `double dr` defining the refinement resolution (cf. Sections 4.3 and 5.5.2 for more details).

refinement_option	2D case	3D case
0	-	no refinement
1	recursive refinement based on the original surface meshes in every time step	recursive refinement based on the original surface meshes in every time step
2	-	pre-refined mesh created before the time stepping, no additional refinement
3	-	as option 1, but based on the pre-refined mesh of option 2

**Table 5.1.:** Specifications of the parameter `refinement_option`. In 2D, the parameter is reset to 1 in any case.

Here, the static boundary points, which are listed right after the regular dynamic boundary points, are associated with the surface meshes of static multibody components the geometry of which neither moves nor changes with time in any way. Hence, that static part of the data, as well as `e1` only needs to be transferred once from the HOTINT- to the LIGGGHTS-side in the initialization procedure (cf. Section 4.4). See also Subsection 5.3.2 for further information concerning the performance of the data transfer. Note that in above data structure, only the standard one-dimensional C-arrays are used to store any multidimensional array-like quantity, for reasons of compatibility with the used MPI routines.

Besides the routines for data access, i.e. read and write routines for member variables / the interface data, the member functions declared in this interface class for communication are

- `virtual void getrv()`: receive the dynamic part of `_r` and `_v`
- `virtual void sendrv()`: send the dynamic part of `_r` and `_v`
- `virtual void getrvfull()`: receive the full `_r` and `_v` arrays
- `virtual void sendrvfull()`: send the full `_r` and `_v` arrays
- `virtual void getrSPH()`: receive `_rSPH`
- `virtual void getvSPH()`: receive `_vSPH`
- `virtual void sendrSPH()`: send `_rSPH`
- `virtual void sendvSPH()`: send `_vSPH`
- `virtual void sendforce()`: send `_f` (only for dynamic boundary points;  $\mathbf{f} \equiv \mathbf{0}$  for all static points by default)
- `virtual void getforce()`: receive `_f` (only for dynamic boundary points)
- `virtual void getrohSPH()`: receive `_rohSPH`
- `virtual void sendrohSPH()`: send `_rohSPH`

This declares the functionality of the interface; the actual implementation of the routines is done in the derived classes `DataH` and `DataL` according to the respective requirements. See Subsections 5.4.1 and 5.5.1 for more details; for the complete C++ code of the interface base class, refer to `interface_baseclass.h` in the appendix.

## 5.3. Exchange of double-precision numbers

### 5.3.1. Conversion and platform-/architecture independent exchange

For the problem at hand, most of the data which should be exchanged via TCP/IP between a 32-bit application running on a Windows host, and a 64-bit application on a LINUX platform consists of double-precision numbers. The transfer of double-precision numbers between different machines with different platforms and/or architectures, in contrast to what might be supposed concerning this issue, is not a straight-forward matter, even though there is, of course, an IEEE standard universally covering the binary representation and arithmetics of floating-point numbers [52]. According to the latter, the standard binary representation of a double-precision number  $d$  is given by 64 bits (8 bytes) via

$$d = (-1)^s \cdot m \cdot 2^e, \quad (5.1)$$

where  $s$  is the sign bit (0 or 1),  $m$  is the mantissa, a positive number defining the precision represented by 53 bits corresponding to 16 digit decimal number (hence 16 digit decimal precision), and  $e$  is the exponent represented by 11 bits, thus ranging from -1023 to 1023 ( $2^{\pm 1023} \approx 10^{\pm 308}$ ).

At first lets have a look at the representation of an integer. For 32-bit applications an integer consists of 4 bytes, or 32 bits, which define its binary representation and are stored sequentially in the memory - the question here is, however, in which order? The two commonly used possibilities, depending on the hardware architecture, are the so called “Big-Endian”, also known as “Network Byte Order”, and “Little-Endian” byte order, according to whether the most significant bit or the least significant bit is stored first, i.e. at the lowest memory address. Similarly, the format of the binary representation of a double is universally defined according to equation (5.1), however, it is not specified how the bit sequences are actually stored in the memory.

This is the reason why the simple approach of sending a double-precision number just by transferring a 1:1 copy of the corresponding memory on one machine to another machine might work, but also might fail completely. Therefore, universal portable transfer routines have been written allowing for the platform- and architecture-independent exchange of double-precision numbers; the corresponding header and source code `dn.h` and `dn.cpp` are given in the appendix.

All of these routines are based on the standard functions

- `htonl(unsigned int a)`: converts an unsigned integer (long) `a` from Host Byte Order (the byte order of the host system) to Network Byte Order (Big Endian)
- `htons(unsigned short a)`: converts an unsigned short `a` from Host Byte Order to Network Byte Order
- `ntohl(unsigned int a)`: converts an unsigned integer `a` from host Network to Host Byte Order
- `ntohs(unsigned short a)`: converts an unsigned short `a` from host Network to Host Byte Order

included in the `winsock2`-library on Windows (header `winsock2.h`) and in `netinet/in.h` on LINUX. With those routines, any unsigned short or integer can be transformed to Network Byte Order on one machine, then sent via TCP/IP to the other machine, and finally converted back again from Network to Host Byte Order. Note that this is only implemented in a 32-bit version, where an (unsigned) integer is assumed to be represented by 4 bytes, and an (unsigned) short by 2 bytes. In case of 64 bit applications, the size of (unsigned) integers is 8 bytes, and for the present problem, we want to exchange data between with a 32-bit and a 64-bit system. Luckily, unless the range of a 4-byte unsigned integer (i.e. 0...4294967295) is exceeded, the routines work correctly even for this mixed case or transfers between two 64 bit systems, since only the effectively first 32 bits (starting with the least significant bit) are accessed here in the memory.

Now, any double precision number  $d = (-1)^s \cdot m \cdot 2^e$  – considering the decimal precision of 16 digits – is transformed into two unsigned 4-byte integers  $a_1$ ,  $a_2$  and one unsigned short  $exp$  based on

$$\begin{aligned} a_1 &= \text{ipart}(10^8 \cdot m) \\ a_2 &= \text{ipart}(10^8 \cdot (10^8 \cdot m - a_1)) \\ exp &= |e|, \end{aligned} \tag{5.2}$$

where `ipart(x)` denotes the integer part of a real number  $x$ , which in the implementation is simply accomplished by explicit type conversion to `unsigned int`, and  $m$  and  $e$  are obtained from the function `frexp(double d, int* ex)` in `math.h`, which returns  $(-1)^s m$  and stores  $e$  to `ex`, in both cases including the sign. In above approach, those signs are stored separately in the last bit of  $a_1$  and  $a_2$ ; note that for the representation of an 8-digit integer number, which is the case with  $a_1$  and  $a_2$ , only 30 bits are needed.

Thus,  $d$  is converted to two unsigned integers and one unsigned short, which, in turn, can be transferred platform- and architecture- independently using above listed Network-to-Host and Host-to-Network routines; the re-conversion of  $a_1$ ,  $a_2$ ,  $exp$  back into a double precision number then works in an analogous way. Of course, the price to pay here is a data overhead, since in this integer representation 10 bytes instead of 8 bytes are required for each double.

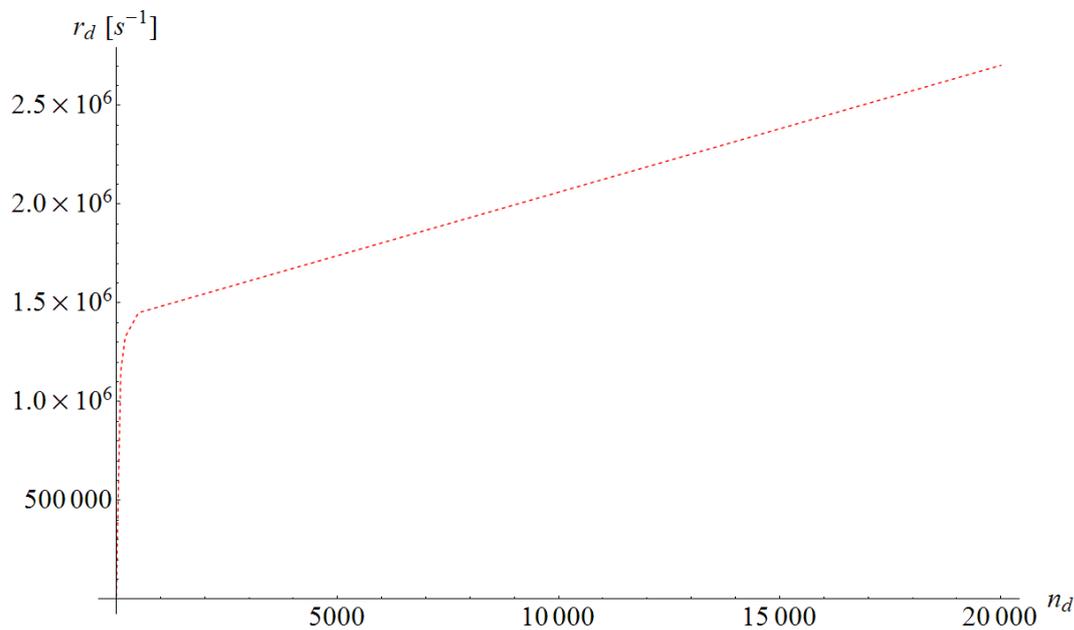
For details on the implementation, see the files `dn.h` and `dn.cpp` in the appendix; more information on the actual performance of the double-precision exchange routines is given in the following subsection.

### 5.3.2. Notes on the performance

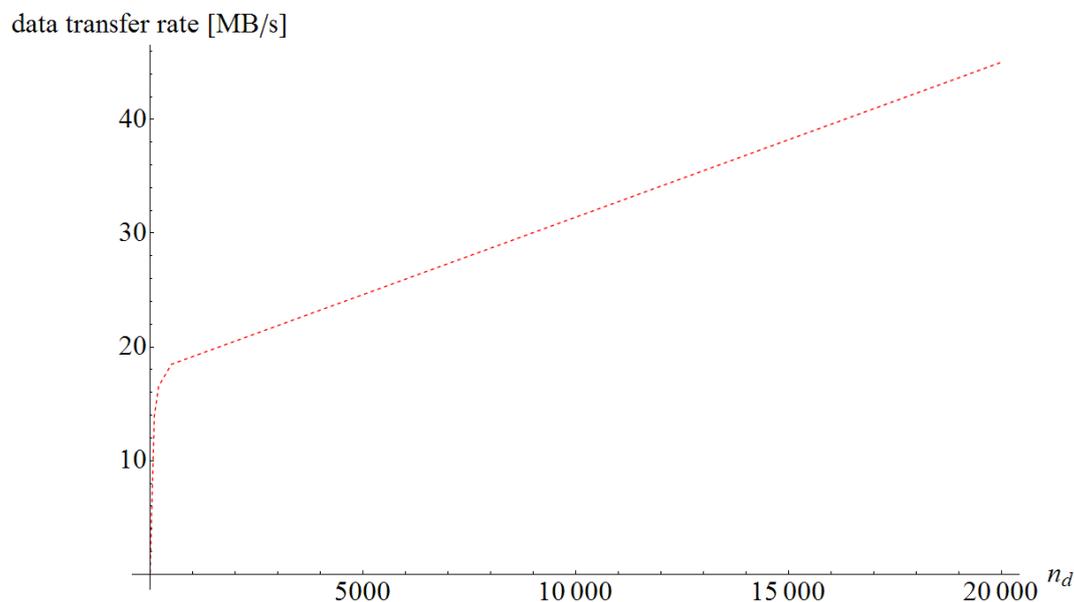
Of course, the overall TCP/IP performance depends on the used hardware (the whole system, in particular the network adapters) as well as software (drivers, firewalls,...); also the computational effort for the conversion process has to be taken into account. For today's most common bandwidths, 100 MBit/s and 1 GBit/s, one can assume an actual maximum data rate around 10 MB/s and 100 MB/s, respectively, which here theoretically corresponds to the platform- and architecture-independent transfer of  $10^6$  and  $10^7$  double precision numbers per second. However, for smaller amounts of data, or large amounts of data split into individually transferred small parts, performance decreases significantly probably due to time delays for the preparation and send/receive requests of that data packages; note that the data overhead of the TCP/IP packet headers is negligible in almost any case (except e.g. for the exchange of single double precision numbers), since it is only 40 bytes in total at a typical packet size of 1500 bytes via Ethernet connections.

Several tests for the “double exchange routines”, each of them run and averaged over 10000 cycles, were performed via a 1 GBit/s virtual network bridge between a Windows 7 host and a LINUX Fedora 15 running on a virtual machine (cf. Section 5.1), with an Intel i7 2720QM CPU:

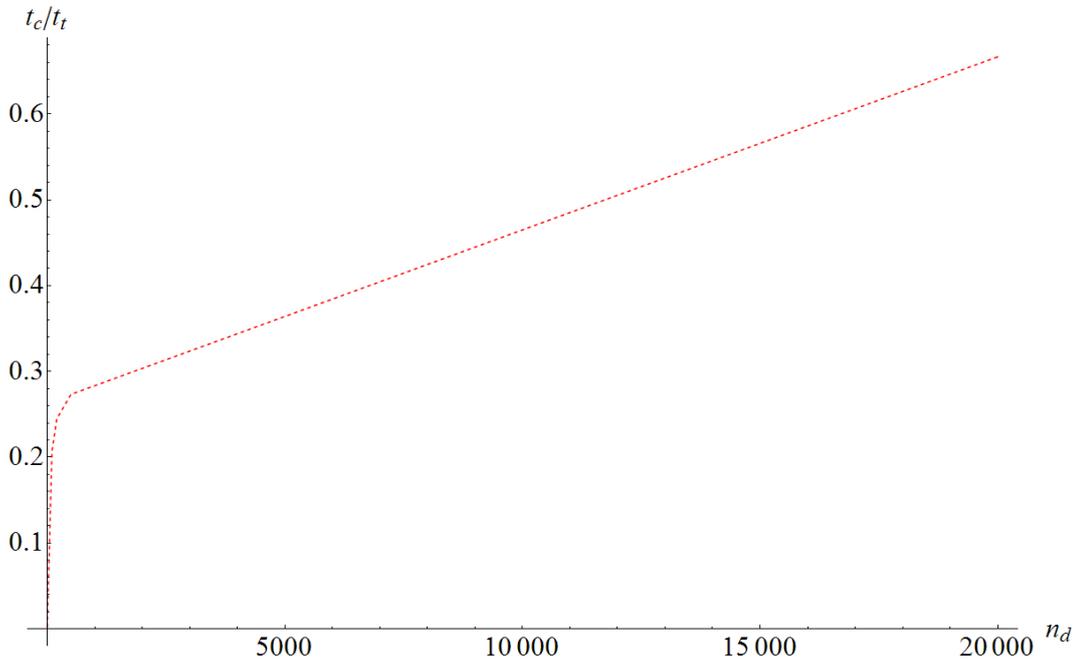
- Average time for one conversion cycle (conversion and back-conversion; Windows-sided, i.e. on the host system); it was determined as approximately  $t_c \approx 0.15 \mu\text{s}$ .
- Average effective transfer rate of double precision numbers in dependency of the number of doubles which are sent per call to a TCP/IP send/receive routine. This was done including the conversion and back-conversion, as well as a consistency test by, firstly, converting and transferring the double arrays from the Windows to the LINUX side, re-converting there, and then doing the same thing in the other direction, where finally the error between the original and the actual array is calculated. At that, the consistency test required the average error to be in the range of  $10^{-16}$ , i.e. the precision limit of double precision numbers. The results are given in Figure 5.2.
- Actual data transfer rate in dependency of the number of doubles which are sent per call to a TCP/IP send/receive routine. See Figure 5.3 for illustration.
- Relative overhead due to the conversion, which is defined by  $(t_c + t_t)/t_t - 1 = t_c/t_t$ , where  $t_c$  designates the time for one conversion cycle (see the first point), and  $t_t$  the time for the single transfer in one direction of one double precision number, in dependency of the number of doubles which are sent per call to a TCP/IP send/receive routine. The data is shown in Figure 5.4.



**Figure 5.2.:** Transfer rate of double precision numbers  $r_d$  vs. the array size  $n_d$  (i.e. the number of doubles sent per call to a TCP/IP send/receive routine); the benchmark included conversion and back-conversion, as well as a consistency test.



**Figure 5.3.:** Effective TCP/IP data transfer rate vs. the array size  $n_d$  (i.e. the number of doubles sent per call to a TCP/IP send/receive routine); the test was performed using a 1 GBit/s virtual network connection.



**Figure 5.4.:** Relative conversion overhead  $t_c/t_t$  vs. the array size  $n_d$  (i.e. the number of doubles sent per call to a TCP/IP send/receive routine), based on  $t_c \approx 0.15 \mu\text{s}$  determined before.

Assuming 100 MB/s effective data bandwidth, corresponding to  $10^{-7}$  s transfer time per double, and above determined time for a conversion cycle, the maximum transfer rate  $r_{d,max}$  on the test system would be

$$r_{d,max} \approx \frac{1}{t_c + 10^{-7}} = 4 \cdot 10^6 \text{ s}^{-1}. \quad (5.3)$$

Summarizing, in a real-world application the estimated maximum transfer rate of double precision numbers probably lies somewhere in the range of  $1 \dots 6 \cdot 10^6 \text{ s}^{-1}$  using a 1 GBit/s ethernet connection, depending on the hardware set-up. Even in the 3D case, this would be enough to exchange the minimum data set (i.e. positions, velocities of and forces on the discrete boundary points, cf. Section 4.4) of  $\approx 50000$  individual surface elements (with 3 vertices each) up to several times per second; thus, in this example the TCP/IP data transfer is not a bottle-neck as long as a full time step (without the communication) on both sides can be performed as fast or even faster, which is most probably not the case if one is working on standard desktop machines. Typically, the main computational effort lies on the fluid side, however, a problem with 50000 surface elements on the structural side, which – with classical finite elements – probably corresponds to several 100000 volume elements, is also very large. Furthermore keep in mind that a refinement of the surface meshes, if necessary, is done anyways locally on the fluid side, which allows for using coarse surface meshes for the components of the MBS; in particular, large (rigid) plane surfaces may be represented by one single surface element only. Hence, the bottleneck of the TCP/IP

communication becomes significant only in cases of relatively small and/or “fast” multibody systems with high-resolution surface meshes and a massively parallel computed fluid side.

Anyways, in none of the simulated test problems (neither in the 2D nor the 3D case) using standard desktop computers, the TCP/IP communication was limiting the overall performance (cf. also Chapter 6). For some further notes on the issue of performance and optimization potential, see the conclusions in Chapter 7; the role of the (optional) SPH data, and the reason for its negligence in above considerations is discussed with the member function `IncomingDataCommunication` in Subsection 5.4.2.

## 5.4. HOTINT / Windows - sided implementation

### 5.4.1. `exchange_class_windows`

The most important classes contained in the files `exchange_class_windows.h` and `exchange_class_windows.cpp` (see the appendix for the full C++ code) are `DataInit` and `DataH`. At that, `DataInit` is a class with dynamic data structure (based on linked lists) which is used for dynamically adding SPH particles, vertices of the surfaces meshes, as well as the surface elements. As soon as this process is finished, the `DataInit` object then is used to pre-initialize a `DataH` instance, or more precisely, the interface data (cf. Section 5.2) associated with that `DataH` instance.

The quantities represented by member variables of the `DataInit` class are

- `unsigned short dim`: dimensionality of the problem
- `unsigned int n`: number of boundary points (vertices of the surface mesh)
- `unsigned int nstat`: number of static boundary points, i.e. points associated with the surface mesh of static components of the multibody system
- `unsigned nel`: number of surface elements
- `unsigned nSPH`: number of SPH particles
- `vec3D* r, v`: dynamic data structure containing position and velocity vectors of the boundary points
- `elemlist2D(3D)* el2D (el3D)`: dynamic data structures containing the numbers of pairs of points (referencing `r` and `v`) which define the vertices of the surface line segments in 2D (in 3D: sets of three vertices corresponding to triangular surface elements)
- `vec3D* rSPH, vSPH`: dynamic data structures containing position and velocity vectors of the SPH particles.

For the dynamic data management and access, the following member functions apply:

- `void add_elem(int*)` and inline `void add_elem(int a, int b, int c=0){ int temp[3]; temp[0]=a; temp[1]=b; temp[2]=c; add_elem(temp);}`: adds a surface element with vertices of numbers `a,b,(c)` in the data structure `r` to `e12D (e13D)`
- `void add_point(double* r0,double* v0)`: adds a boundary point with position `r0` and velocity `v0` to `r` and `v`
- `void add_pointSPH(double* r0,double* v0)`: adds an SPH particle with position `r0` and velocity `v0` to `rSPH, vSPH`
- `unsigned int get_el(int i, int j) const`: returns the `j`-th point of element `i` (counting starts from 0)
- `double get_r(int i, int j) const`: returns the `j`-th coordinate of `i`-th boundary point position (counting starts from 0)
- `double get_v(int i, int j) const`: returns the `j`-th coordinate of `i`-th boundary point velocity (counting starts from 0)
- `double get_rSPH(int i, int j) const`: returns the `j`-th coordinate of `i`-th SPH particle position (counting starts from 0)
- `double get_vSPH(int i, int j) const`: returns the `j`-th coordinate of `i`-th SPH particle velocity (counting starts from 0)

With this, an arbitrary initial system consisting of surface elements (arbitrary single, separate elements or elements associated with any kind of surface mesh) and SPH particles can be set up. The corresponding `DataInit` object then is used for initialization of an instance of the `DataH` class, which is discussed in the following.

The only additional member variables to the inherited data set of `interface_baseclass` of the `DataH` class are two socket descriptors `int s` and `int c` identifying the TCP/IP sockets used for communication which can be thought of playing a similar role as file pointers or descriptors for file I/O operations, and a flag `bool isinitialized` specifying the initialization status of the object.

Based on the data of the `DataInit` object and some additional specifications as constructor arguments (e.g. the mass of the SPH particles), `DataH` accounts for

- the initialization of the interface data,
- the set-up of the TCP/IP server and the connection to the client (`DataL` object) as well as the corresponding shut-down and clean-up (based on the socket libraries `winsock2.h` on the Windows side, and the corresponding socket libraries for LINUX platforms; see the appendix with the full C++ source or [53] and [54] for more (theoretical) information on this issue)
- the initial transfer of the interface data,

- the set-up of LIGGGHTS based on a LIGGGHTS input script and additional commands partly based on specifications made in the HOTINT parameter files,
- any communication/synchronization tasks between the HOTINT- and LIGGGHTS-side during the time stepping, and
- any methods (read and write) necessary for the access to the interface data set.

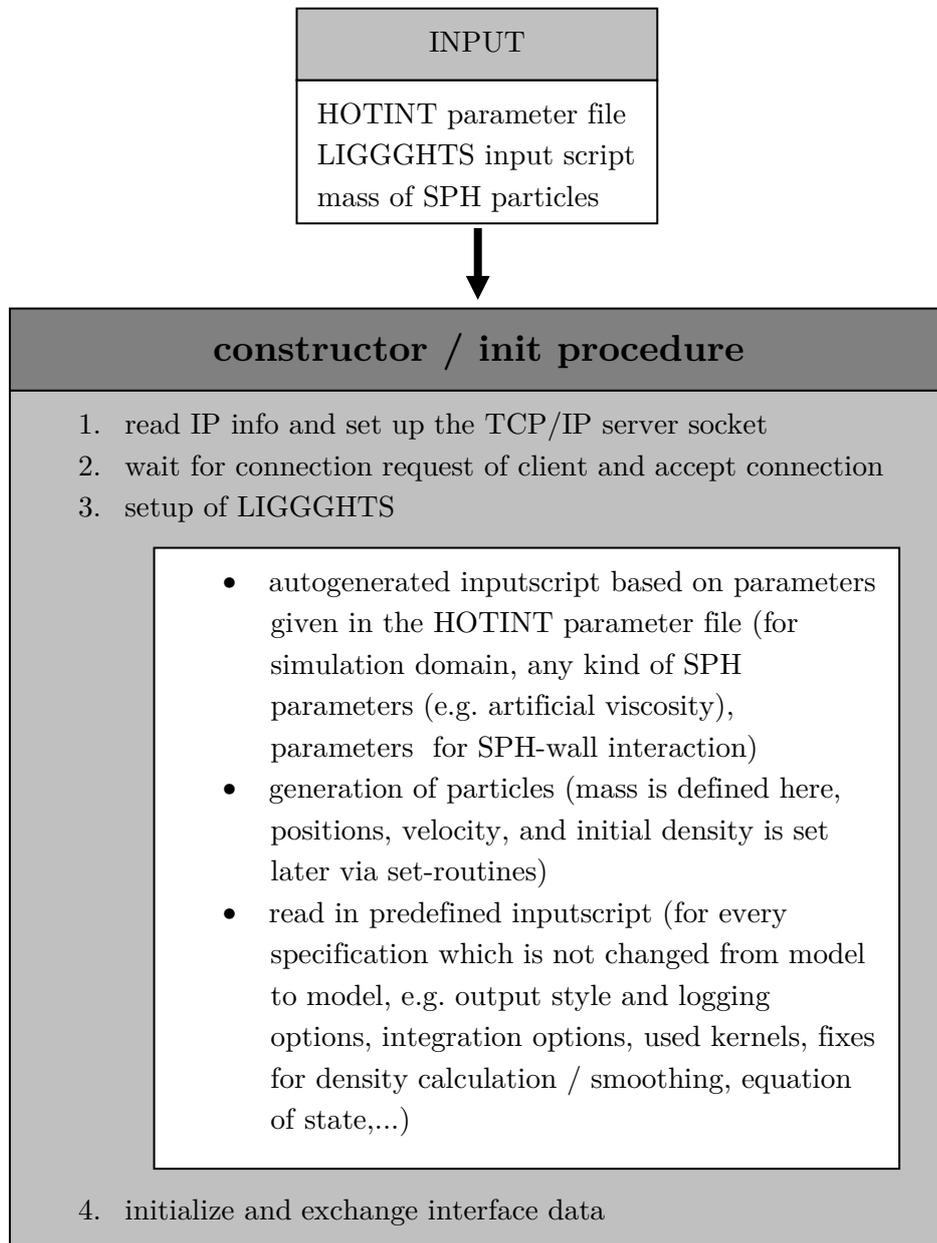
For reasons of clarity and, in particular, brevity, the functionality of `DataH` shall be illustrated only in the schematic diagrams shown in Figure 5.5 covering the functionality of the constructor and initialization procedure, and 5.6 for a short description of the member functions. Again, for detailed information, the reader is referred to the source code given in the appendix.

#### 5.4.2. `fsi_communication_element`

As a “special” element, the class `fsi_communication_element` is derived from the `element`-class in HOTINT; thus, access to the whole multibody system defined and assembled in HOTINT is given via a pointer of type `MBS*`. At that, the “top-level class” `MBS` contains references to all components defined and created in the HOTINT model file. For the representation of the SPH particles, additionally, new element classes were derived - `SPHParticle2D` and `SPHParticle3D` - amongst other things containing SPH-specific data (position, velocity, and density) about the corresponding particle. Surface meshes are managed using HOTINTs `GeomElement` classes, which originally were designed for drawing purposes only: A `GeomElement`, for example a line element in 2D or a triangle in 3D, can be created and linked to an “actual” component of the multibody system, and then (optionally) be drawn instead of the corresponding element; if that element undergoes any kind of motion or deformation, the `GeomElement` moves and deforms accordingly, because the link between the two element instances is established in terms of local coordinates fixed on the bodies (Lagrangian perspective). However, it is exactly this and no additional functionality necessary and sufficient for the implementation of the contact formalism on the structural side - for each body interacting with the fluid, an appropriate mesh consisting of `GeomElements` is created, which, at all times, represents the discretized surface of the arbitrarily moving and/or deforming body. Thus, they define the surface mesh vertices – local boundary points on the multibody components surfaces – which represent the surface geometry and kinematics of the body on the fluid side, and which, in turn, the forces due to the interaction with the fluid act on (cf. Sections 4.2 and 4.3). `GeomElements` additionally can be connected to the ground of the MBS, i.e. fixed in the inertial frame, representing arbitrary static geometric objects or static boundaries; in that case, any forces acting on those components are set equal to zero.

The most important member variables of the communication element are

- `int dim`: dimensionality of the problem



**Figure 5.5.:** Schematic diagram of the tasks of the DataH constructor and initialization procedure.

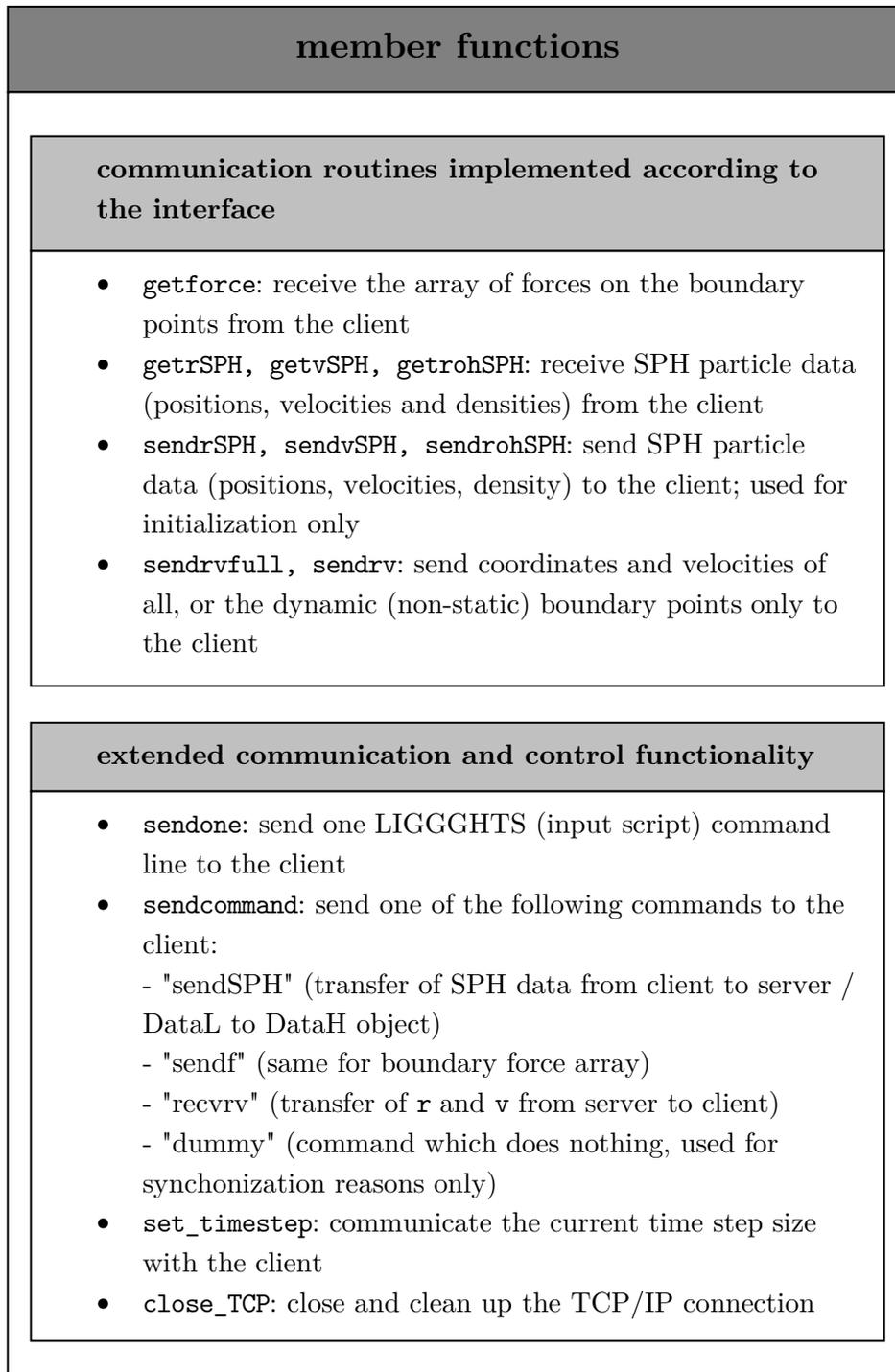


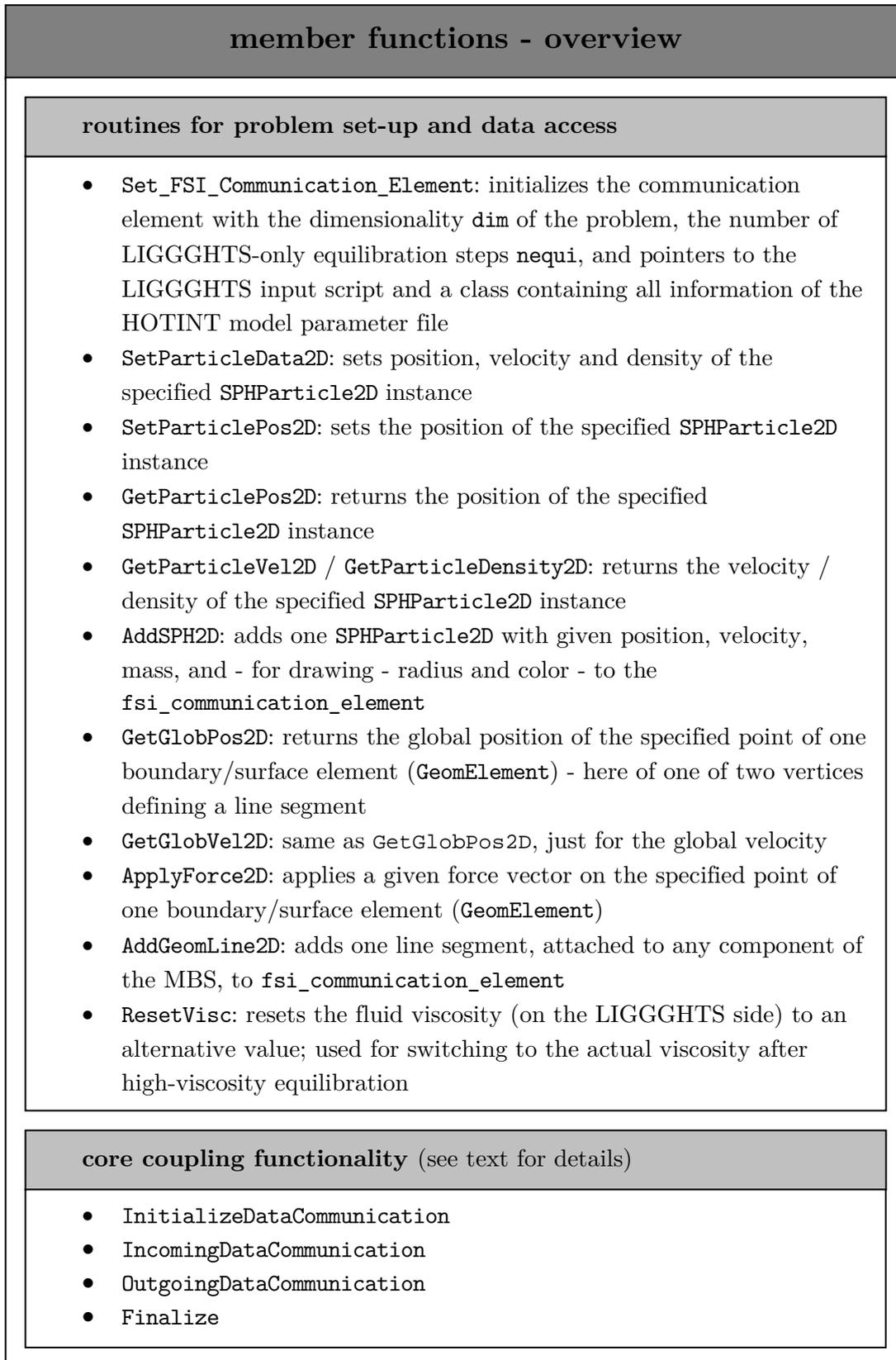
Figure 5.6.: Outline of the DataH member functions.

- `int nparticles`: number of SPH particles
- `int nequi`: number of LIGGGHTS-only equilibration steps
- `TArray<int> sph`: dynamic array containing the numbers of the `SPHParticle2D(3D)` instances in MBS
- `DataH dataobj`: instance of the actual coupling and server class (cf. Subsection 5.4.1)
- `TArray<int> boundary`: dynamic array containing the element numbers of all surface elements (integrated in HOTINT as `GeomElements`, as discussed above) in MBS
- `TArray<int> boundary_flags`: dynamic array with flags corresponding to `boundary`, defining whether the respective element is a static (no motion or deformation during the whole runtime) or a dynamic element; note that only the dynamic part needs to be exchanged between the two sides in every time step,

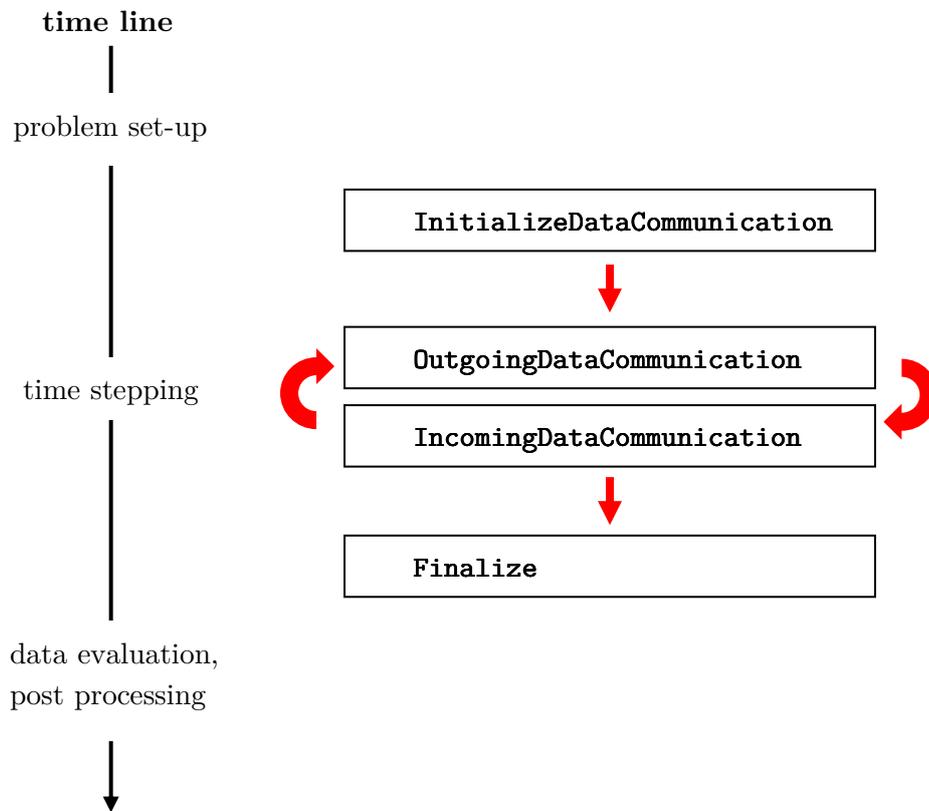
whereas the essential member functions (for 2D) are outlined in Figure 5.7.

The core functionality, based on the routines for “problem set-up and data access” in Figure 5.7, is implemented in the functions `InitializeDataCommunication` (called before the time-stepping), `IncomingDataCommunication` and `OutgoingDataCommunication` (called in every time step), and `Finalize` (called when the whole computation is finished), which shall be discussed in the following in more detail; a flow chart is given in Figure 5.8.

- `InitializeDataCommunication`:
  - Boundary elements are rearranged into a dynamic and a static part, i.e. the dynamic arrays `boundary` and `boundary_flags` are sorted accordingly.
  - In that order, `r`, `v`, `e12D` are dynamically created and added via a local instance of `DataInit`; the number of static points `nstat` is defined.
  - `rSPH` and `vSPH` are created via this `DataInit` instance.
  - `dataobj` is initialized based on above local `DataInit` instance, as well as the LIGGGHTS input script and a given mass of the SPH particles which is calculated from the (approximate) average volume per particle in the initial configuration and the nominal initial SPH mass density. After some numerical experiments using a regular particle distribution according to some kind of lattice this has proven to be the most useful approach, since for a given volume to be filled with fluid this yields a configuration relatively close to equilibrium; however, some sort of equilibration procedure still is necessary (cf. a few points below). Theoretically - if it is for any reason not important to fill a given volume, i.e. that the effective volume occupied by the particles in dynamic equilibrium is approximately equal to the volume they were distributed over initially - it should be possible to start with any “unphysical” configuration of mass, density



**Figure 5.7.:** Outline of the important member functions of `fsi_communication_element` (for 2D).



**Figure 5.8.:** Program flow chart with the core functionality of `fsi_communication_element`.

- and particle distribution, and the SPH dynamics should lead to a dynamic equilibrium. Nevertheless, initial configurations too far off of equilibrium may result in artefacts (e.g. formation of drops with very low density inside and high local density close to the surface, in case of an initial configuration with significantly lower density than the nominal density) or even instabilities.
- Initialization of the density array with the nominal density (defined in the HOTINT parameter file) in the interface via `dataobj`.
  - Definition and communication of the initial time step `dt` over the interface (specified by the HOTINT solver options).
  - Using the initial configuration of the MBS as fully static configuration, the LIGGGHTS side now computes `nequi` "LIGGGHTS-only" equilibration steps with a high viscosity. This has to be done if a quasi-equilibrium situation (or something close to that) is desired on the fluid side before the actual dynamic coupled simulation, since the particle configuration - for reasons of simplicity - typically is created according to some kind of regular lattice with constant mass and density, which clearly can not be an equilibrium configuration. It is difficult to obtain an SPH equilibrium configuration in a different way. One possibility - theoretically - would be to assume a nearly incompressible case and assign constant density to all SPH particles, then initialize their positions in such a way, that all forces - the internal forces due to pressure and viscosity as well as the contact forces due to the SPH-wall interaction - vanish. One could also think of some kind of density and/or mass variation procedure with a given particle distribution in order to reach (local) equilibrium. However, since all of these approaches either yield large, in general non-linear problems which would have to be solved iteratively anyways, or are iterative procedures themselves to begin with, there are no remaining benefits over the equilibration process with the already implemented (iterative) process of time integration using the initial multibody configuration as static boundaries and sufficiently high viscosity to ensure a high dissipation rate of the kinetic energy. As a final remark, it should be noted that the existence of an external force field of some kind (e.g. gravitational force) or the generation of a slight external pressure (e.g. using a piston) is not a requirement, but certainly conducive to the equilibration process, especially if it is important to completely fill up a geometry with fluid.
- **IncomingDataCommunication:**
    - In predefined time intervals – depending on the present problem, approximately once every hundred to a few thousand time steps – the full SPH data set (position, velocity, density) is transferred from the LIGGGHTS to the HOTINT side via `dataobj`; it should be noted that this information is not essential for

the calculation, but only used for visualization and post-processing. The reason why this is not done in every time step is to minimize the average amount of TCP/IP data transfer per time step and avoid a possible bottle-neck. For illustration, consider the following estimate: For each particle, above SPH data set is given by 9 double-precision numbers of the size of 10 bytes each in the exchange procedures (cf. Section 5.3), i.e. 90 bytes in total. For  $10^5$  particles this would result in 9 MB data size, and, assuming an ethernet connection with 100MBit/s bandwidth, limit the number of time steps per second already to a value near to or smaller than 1. Note that - especially for 3D problems -  $10^5$  particles is not that much, and with parallel computation of the fluid simulation on several CPUs the problem of that TCP/IP bottleneck can be significant. In contrast to that, if the exchange of the SPH data is done only once in every thousand time steps, which still is sufficient for visualization and post-processing in most cases due to the very small time step (typically in the range of  $10^{-5}$  s) required by the explicit integration on the fluid side, the time overhead due to that data transfer is negligible.

- The array of forces on the boundary (surface) elements is transferred from the LIGGGHTS to the HOTINT side in every time step.
- Application of those forces to the corresponding boundary points, which are local points on the components of the MBS defined by the associated `GeomElements`.
- **OutgoingDataCommunication:**
  - Setting and exchange of the current time step size `dt` via `dataobj`.
  - Positions and velocities of the boundary points are read from the `GeomElements` in HOTINT, and updated in the interface via `dataobj`.
  - That position and velocity data is transferred from the HOTINT to the LIGGGHTS side via TCP/IP.
  - A command is sent to the LIGGGHTS side to run one time step without recalculation of neighbor lists (which are still valid from the previous time step) or the previous force array (cf.  $\mathbf{F}(t)$  in equation (3.55)).
- **Finalize:**
  - A termination command is sent to the LIGGGHTS-side; there, the time-step loop is exited, the current TCP/IP connection is closed, everything is cleaned up, and the `wrapper code LINUX` (cf. the following section) falls back into a waiting loop, awaiting a new connection and problem set-up from the server side.

- HOTINT-sided, the server socket establishing the TCP/IP connection is closed; the coupled simulation has finished, and the data now can be used for post-processing.

## 5.5. LIGGGHTS / LINUX - sided implementation

### 5.5.1. exchange\_class\_linux

The class `DataL`, implemented in the files `exchange_class_linux.h` and `exchange_class_linux.cpp` (see the appendix for the full C++ code), is the counterpiece to the `DataH` class discussed in Subsection 5.4.1. Thus, it includes the corresponding TCP/IP client functionality and provides a response to any communication task initiated by the `DataH` object on the server side, accounting for

- the set-up of the TCP/IP client and the connection to the server (`DataH` object) as well as the corresponding shut-down and clean-up (based on implementations for network sockets declared in `unistd.h`, `errno.h`, `netdb.h`, `sys/types.h`, `netinet/in.h`, `sys/socket.h`, and `arpa/inet.h`; see [53] for detailed information on this issue)
- the set-up and initialization of LAMMPS/LIGGGHTS (via the instantiation of LAMMPS objects), based on the input script data provided by the server and additional command line arguments
- the initialization of the interface data (received from the server)
- any communication/synchronization tasks between the HOTINT- and LIGGGHTS-side during the time stepping, and
- any methods (read and write) necessary for the access to the interface data set.

It should be noted that this class is, as any other part on the LINUX side, written in parallel C++ code based on the message-passing interface (MPI, cf. also Section 5.1 and [50, 51]); hence, one instance of a `DataL` object is created on each process – which, in this implementation, corresponds to exactly one CPU core with an associated memory space – where the total number of those processes is specified when the whole application is started in the MPI environment via the wrapper program discussed in Subsection 5.5.3. In any case, the actual communication with the server is done by one (the first) process only, whereas all other processes play a passive role concerning the TCP/IP exchange, and synchronize or update their data set via local communication with that first process. Apart from that, any other task, including the actual simulation of the fluid, is done in parallel based on a spatial decomposition of the simulation domain which is created during the construction procedure of the LAMMPS objects. At that, if the application is started on  $n$  cores (processes), the box-shaped simulation domain – previously specified via the LIGGGHTS input script – is subdivided into  $n$  smaller boxes, each of them associated with one core. Consequently,

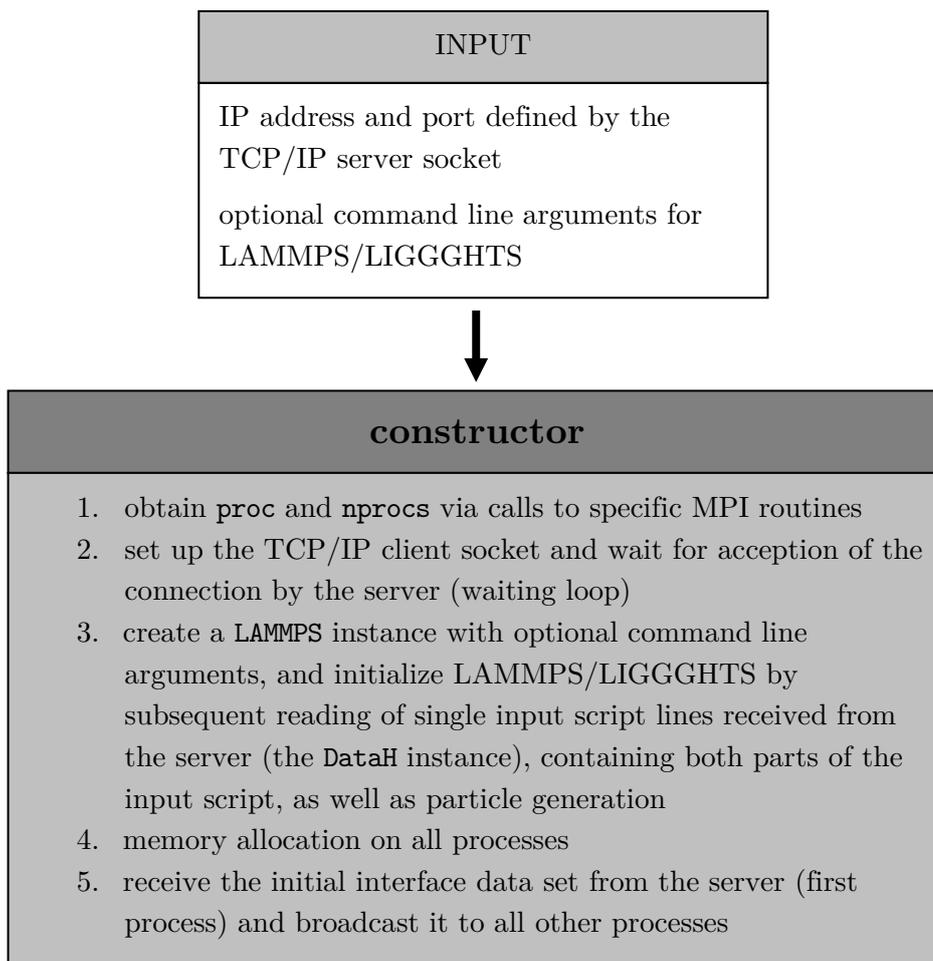
every process deals with a local set of particles (corresponding to its subdomain), and additionally accounts for contributions from the other subdomains which is accomplished via explicit communication (message-passing) with the other processes. Note that in case of short-range interactions, which is given for the pairwise contributions in the SPH as well as the developed contact formalism (cf. Sections 3.1.2 and 4.2), only a fraction of that non-local particles has to be considered, namely those within interaction range (i.e. the kernel support radius) to the boundaries of the respective subdomain (“ghost particles”, cf. Subsection 3.2.3). For a 3D system with a total number of  $N$  particles distributed over a volume  $V$ , the average distance between two particles can be estimated by  $r_{av} = (V/N)^{1/3}$ ; now, let  $V_{sub} \approx V/n$  be the volume and  $A_{sub} \approx V_{sub}^{2/3}$  the surface area of a local subdomain, based on which the number of local particles approximately is  $N_{loc} \approx (V_{sub}^{1/3}/r_{av})^3$ , whereas the number of particles from other subdomains with influence on the local particles is a “surface contribution” on the order  $N_{nonloc} \approx (V_{sub}^{1/3}/r_{av})^2$ . In case of large systems with  $V_{sub}^{1/3} \gg r_{av}$ , corresponding to a large  $N$  or  $N_{loc}$ , we have  $N_{nonloc} \ll N_{loc}$ , and thus the overhead due to communication between the processes becomes negligible. Due to the efficient methods implemented for the computation of all pair interactions, which is the only type of interaction here – either between pairs of two SPH particles or between an SPH particle and a boundary point – the overall computational costs and thus, the CPU time then scale roughly linear with  $N$  and, for sufficiently large  $N_{loc}$ , linear with  $1/n$ :

$$t_{CPU} \propto \frac{N}{n} \approx N_{loc}. \quad (5.4)$$

If  $N_{loc}$  falls below a certain limit, performance with respect to the scaling by  $1/n$  decreases due to above mentioned inter-process communication overhead; as a rough estimate, in the performed test simulations that limit lay in the range of  $N_{loc} \approx 1000 - 3000$ . Note that for the 2D case, above considerations are completely analogous.

Getting back to the implementation, as it has already been pointed out `DataL` is, such as `DataH`, derived from the interface base class (cf. Section 5.2), and moreover contains the following additional member variables:

- `int proc`: number of the current process in the MPI environment
- `int nprocs`: total number of processes started in the MPI environment
- `int c`: TCP/IP socket descriptor of the client socket
- `LAMMPS* lp`: pointer to the LAMMPS/LIGGGHTS base class which is instantiated by each `DataL` object during construction and provides complete access to the whole functionality of LIGGGHTS; It should be noted that the LAMMPS base class was slightly adapted by the addition of a pointer to exactly that `DataL` object which created the respective instance of LAMMPS; via that pointer, the functionality of `DataL` can be accessed from any point within LIGGGHTS, in turn.



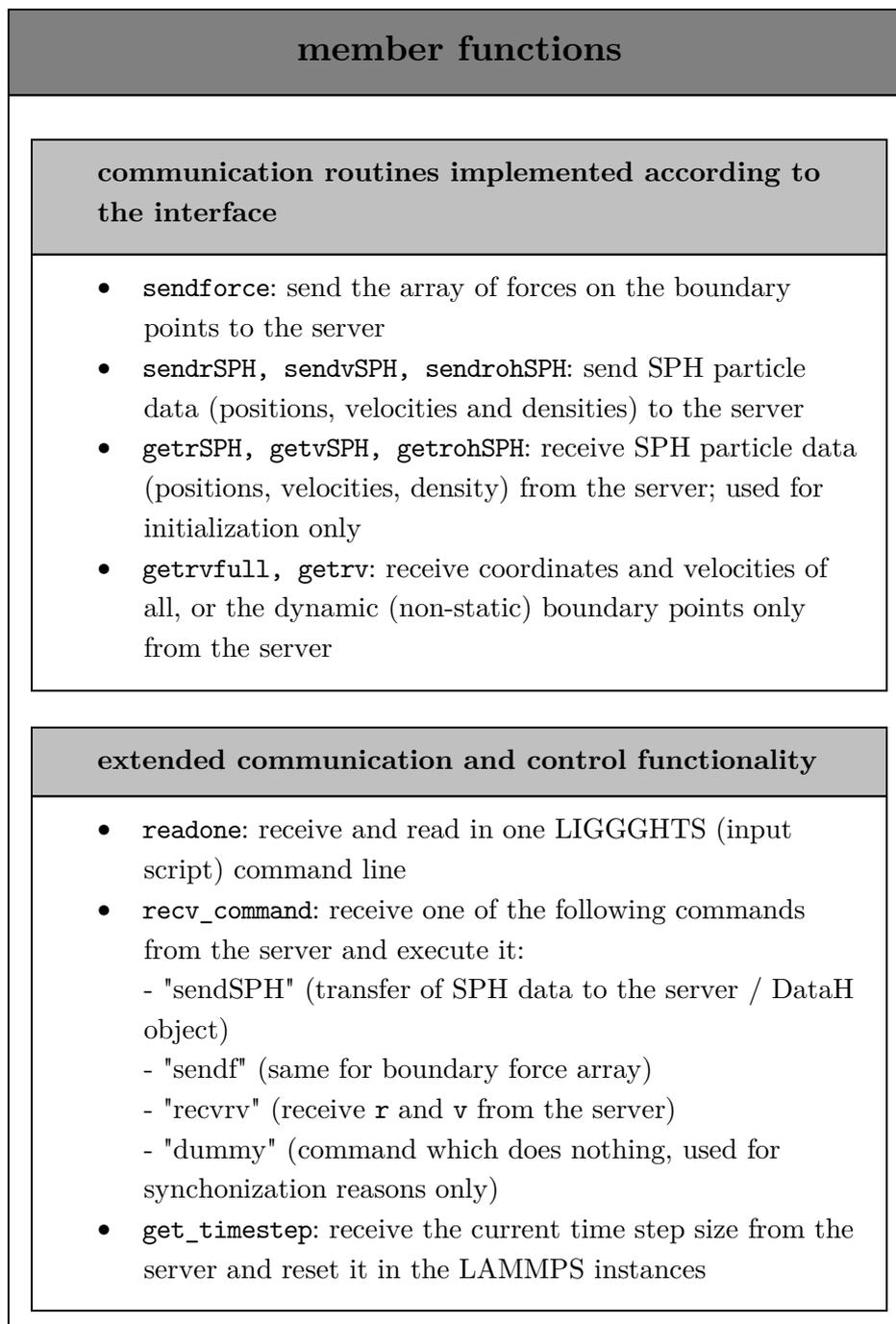
**Figure 5.9.:** Schematic diagram of the tasks of the `DataL` constructor and initialization procedure.

Once again, the construction procedure and most important member functions are outlined in the Figures 5.9 and 5.10; for details refer to the source code in the appendix.

### 5.5.2. `fix_FSI_SPH`

The files `fix_FSI_SPH.h` and `fix_FSI_SPH.cpp` contain the class `fix_FSI_SPH` which was developed and implemented as a so-called “fix” in the framework of LIGGGHTS, and which is responsible for the computation of the fluid-structure interaction forces (cf. Sections 4.2 and 4.3). At that, a “fix” in LIGGGHTS is a class derived from the baseclass `Fix` implementing member functions which are called at one or several stages of every time step (cf. Table 5.2) after inclusion of the corresponding command along with its parameters in the input script.

Here, the computation of the contact forces between fluid – in fact, the SPH particles – and the components of the MBS is performed in stage 7 (`post_force()`) after the evaluation of the (pair) interactions between the SPH particles, based on the parameters



**Figure 5.10.:** Outline of the DataL member functions.

stage	member function	base class	notes
1	<code>initial_integrate()</code>	Fix	first Velocity-Verlet substep ( $\mathbf{R}(t)$ and $\tilde{\mathbf{V}}(t)$ , cf. equation (3.55))
2	<code>post_integrate()</code>	Fix	neighbor list build (cf. Subsection 3.2.3) and other parallel operations
3	<code>pre_force()</code>	Fix	operations immediately before the main force calculation
4	<code>force→pair→compute()</code>	Pair	computation of short-range pair interactions
5	<code>force→bond→compute()</code>	Bond	inclusion of bonds between atoms
6	<code>force→kspace→compute()</code>	Kspace	k-space solver for long-range interactions
7	<code>post_force()</code>	Fix	operations immediately after the main force calculation
8	<code>final_integrate()</code>	Fix	second Velocity-Verlet substep ( $\mathbf{V}(t)$ , cf. equation (3.55))
9	<code>end_of_step()</code>	Fix	operations after the completion of one full integration step
10	<code>output→write()</code>	Output	output of simulation data, log file, ...

**Table 5.2.:** The stages of one time step in LIGGGHTS, with the corresponding member functions implemented in classes derived from the given base classes (e.g. `Fix`) which are called at the respective stage in every time step.

- $k$ : scaling factor for the repulsive force
- $t$ : scaling factor for the viscous force
- $r_0$ : equilibrium distance, i.e. the relative distance between an SPH particle and a boundary point at which the repulsive force vanishes
- $r_c = 2h$ : cut-off radius / range of the interaction potential between boundaries (boundary points) and SPH particles; assumed to have the same value for both the repulsive as well as the viscous terms,

which need to be specified for the fix `fix_FSI_SPH` in the input script, and the force densities given in equation (4.2) using the Spiky kernel (cf. equation (3.24)), renormalized by  $\alpha$  such that

$$\frac{1}{\alpha} \Delta_{\mathbf{r}_i} W(\mathbf{r}', \mathbf{r}_i, h) = 2 - |\mathbf{r} - \mathbf{r}'| / h. \quad (5.5)$$

As it has already been mentioned in the previous section, access to the data set of the interface is given via a pointer added to the LAMMPS base class on that `DataL` object which instantiates the LAMMPS object. Now, the evaluation of the interaction forces using the discretized equations (4.4) is performed on each MPI process (here, equivalent to one CPU core and its associated memory space) – operating on the interface data of the discretized boundaries as well as the local SPH data (position, velocities and forces) the LAMMPS instances are working with – according to the following scheme:

1. **Generation of a cell grid:** The rectangular (2D) or box-shaped (3D) simulation subdomain is split into regular cells with edge lengths greater than, but as close as possible to the interaction range of the contact forces  $r_c$ .
2. **Hashing of all local SPH particles on that cell grid:** Each cell of the cell grid is associated with a dynamic array of particle numbers (“cell list”). In every time step, for any local SPH particle the cell which contains that particle is determined and the particle number is added to the corresponding cell list. This, of course, is a linear operation on the order  $O(N_{loc})$  concerning the computational effort.
3. **Iteration through all surface elements, adaptive refinement and force calculation (2D):** Now, an iteration through all surface elements (line segments defined by their end points) is done, and the following steps are performed sequentially (element after element):
  - a) The length  $l$  of the line segment is determined. If  $l > \mathbf{dr}$ , i.e. the refinement resolution defined in the interface (cf. Section 5.2) and specified in the HOTINT parameter file, iterative bisections of the segment are performed, until the length of the hereby created sub-elements  $l_{sub}$  is smaller than  $\mathbf{dr}$ . For numerical surface integration, a 3-point Legendre-Gauss quadrature is used, and thus,  $\mathbf{dr}$  should be in the range of  $r_c = 2h$ , as it has already been discussed in detail in Section 4.3. Note also that in the 2D case the adaptive refinement of each line segment

is done in every time step, regardless of the parameter `refinement_option` (cf. Section 5.2). As discussed in Subsection 5.3.2, for optimal performance and a minimum of TCP/IP transfer data, the mesh refinement is performed locally on the LIGGGHTS side.

- b) Based on the refined sub-elements of each surface element, the positions of the three local Gauss points are computed (cf. equation (4.5)) which are then hashed on the cell grid. If any of those points lies inside a cell of the local cell grid, or one of the “ghost” cells (within other subdomains) adjacent to the boundaries of the current subdomain, the corresponding velocity of the Gauss point is calculated (cf. again equation (4.5)), and the contact force is computed between that point and all SPH particles which lie in the same or in adjacent cells of the cell grid. Note that, with above definition of the cell grid, this covers all possible interactions between, since any SPH particle in any other than the considered cells just is outside the interaction range. This procedure, known as “cell method” or “cell-linked lists”, is a well known method for the efficient determination of contact pairs (contact search) in many-particle problems, especially in the field of molecular dynamics: While the “brute force” approach would just go through all  $N_{pairs}$  possible pairs of  $N_b$  boundary points and  $N_{loc}$  SPH particles,

$$N_{pairs} = N_b N_{loc}, \quad (5.6)$$

here, with a number of  $n_c$  cells in the cell grid of the local subdomain, merely a fraction on the order

$$N_{pairs} \approx N_b \frac{N_{loc}}{n_c} \quad (5.7)$$

needs to be considered, which is significant especially for large systems. Keep in mind that for a typical problem configuration the SPH smoothing length  $h$  is chosen such that  $N_{support} \approx 30 - 100$  particles lie within the kernel support domain. A reasonable choice of  $r_c$  would be somewhere between  $h$  and  $2h$ , which means that any cell in the cell grid roughly also contains  $N_{support}$  particles, yielding

$$n_c \approx \frac{N_{loc}}{N_{support}}, \quad (5.8)$$

and thus with equation (5.7)

$$N_{pairs} \approx N_b N_{support} \ll N_b N_{loc}. \quad (5.9)$$

Importantly, for a given system configuration on a given number of processes the computational effort of this procedure scales linearly with  $N_b$ , since  $N_{support}$  stays approximately constant independent from spatial resolution. Thus, the CPU effort of the whole fluid side – the computation of the SPH dynamics

as well as the contact evaluation, which are performed sequentially as separate (decoupled) tasks in every time step – scales linearly with the number of particles and/or sampling (boundary) points (cf. also Subsections 3.1.3 and 3.2.3).

- c) **Update of force arrays:** Finally, the calculated forces are added appropriately to the LIGGGHTS force array of the SPH particles, as well as the force array corresponding to the boundary points in the interface after appropriate redistribution of the local forces to the endpoints of the respective line segment (cf. equation (4.12)).

Refer to Section 4.3 for a detailed discussion of the theoretical and mathematical background of the discrete calculation of the contact forces, the numerical integration, as well as the force redistribution.

### 5.5.3. wrapper code LINUX

The wrapper code on the LINUX side provides the environment which the `DataL` instances, and thus, LIGGGHTS, are running in; cf. to Figure 5.1 for an overview of the coupled program structure.

Essentially, the wrapper program is a small top-level application making use of the functionality of both `DataL` and LIGGGHTS – more or less – as static libraries. It is responsible for

- the set-up of the MPI environment
- the instantiation and initialization of the `DataL` instance, and
- the calls to the communication routines via `DataL`, synchronized with the server side (cf. Subsection 5.4.2) in the time stepping.

At that, a typical call to start the wrapper program via the command line would be

```
mpirun -np 4 main 192.168.56.1 12345 -log none,
```

which starts the executable `main` (compiled `wrapper code LINUX`) in the MPI environment on 4 processes, with the command line arguments `192.168.56.1` (IP address of the server socket), `12345` (port which the server socket is bound to), and one additional LIGGGHTS command line argument, `-log none` (specifying that no log file should be written to the hard drive), which is passed later to the LAMMPS instance.

Concludingly, an outline of the structure of the wrapper code is shown in form of a block diagram in Figure 5.11; for further details, refer to the C++ source code given in the appendix.

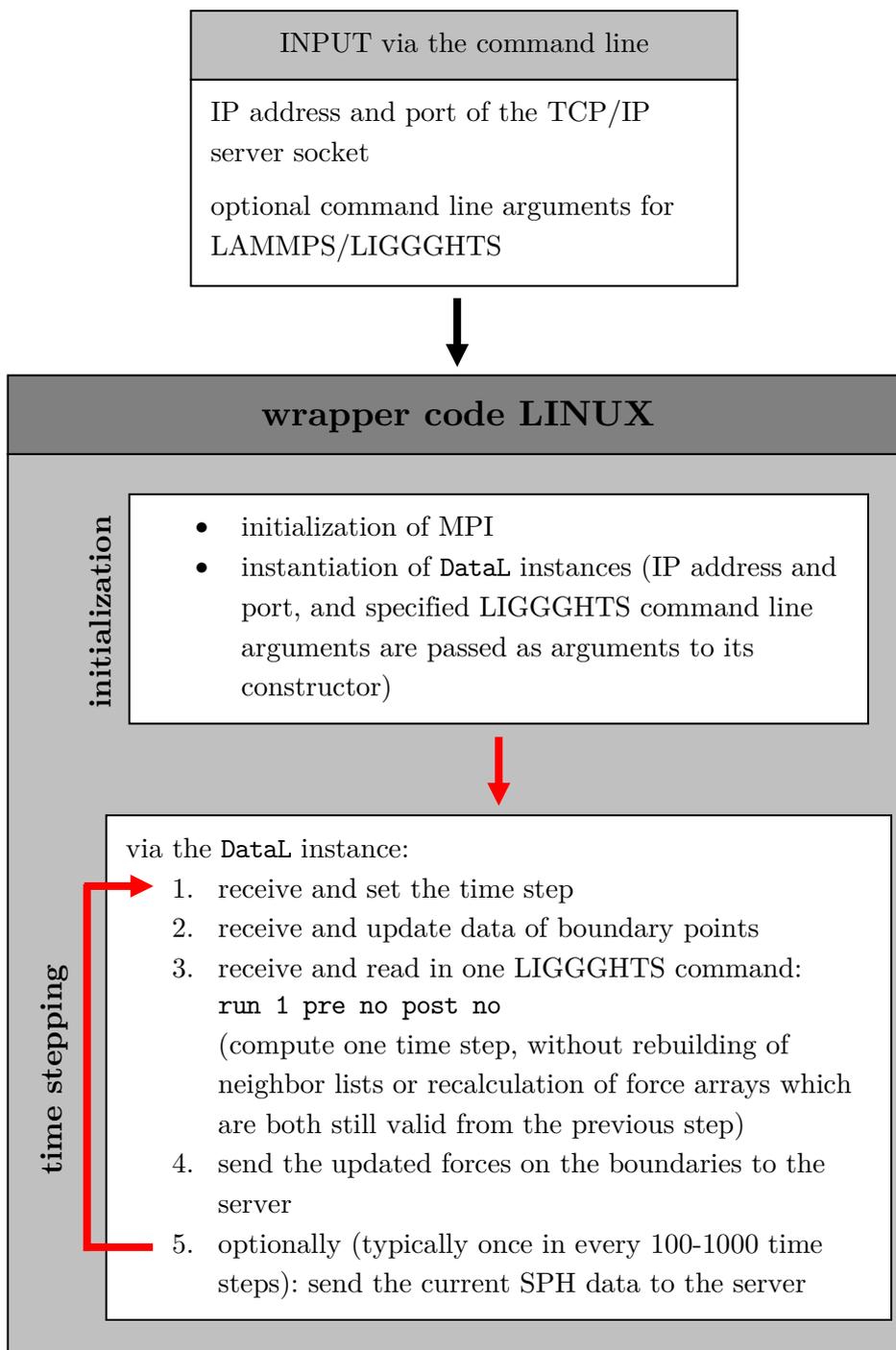
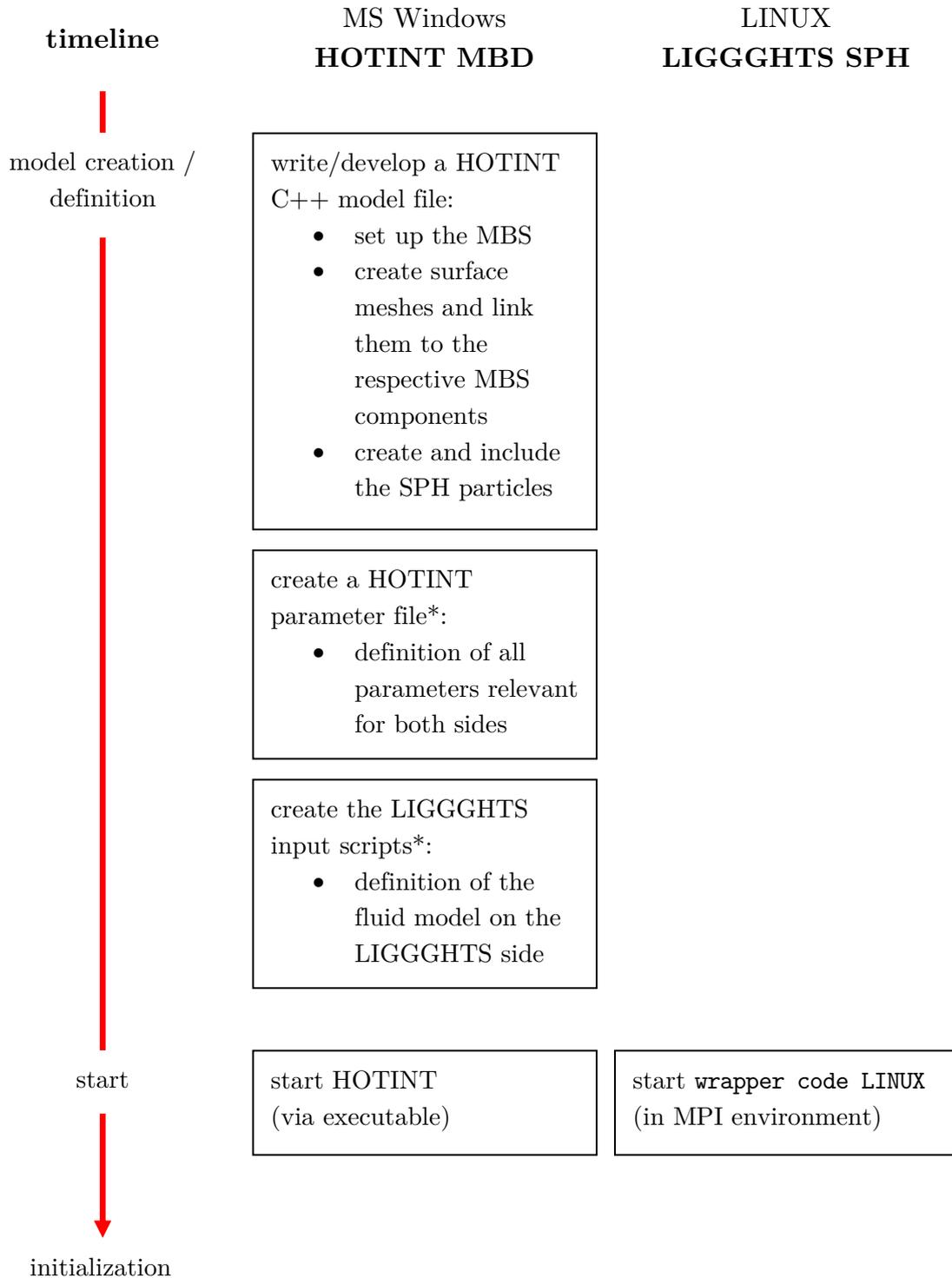


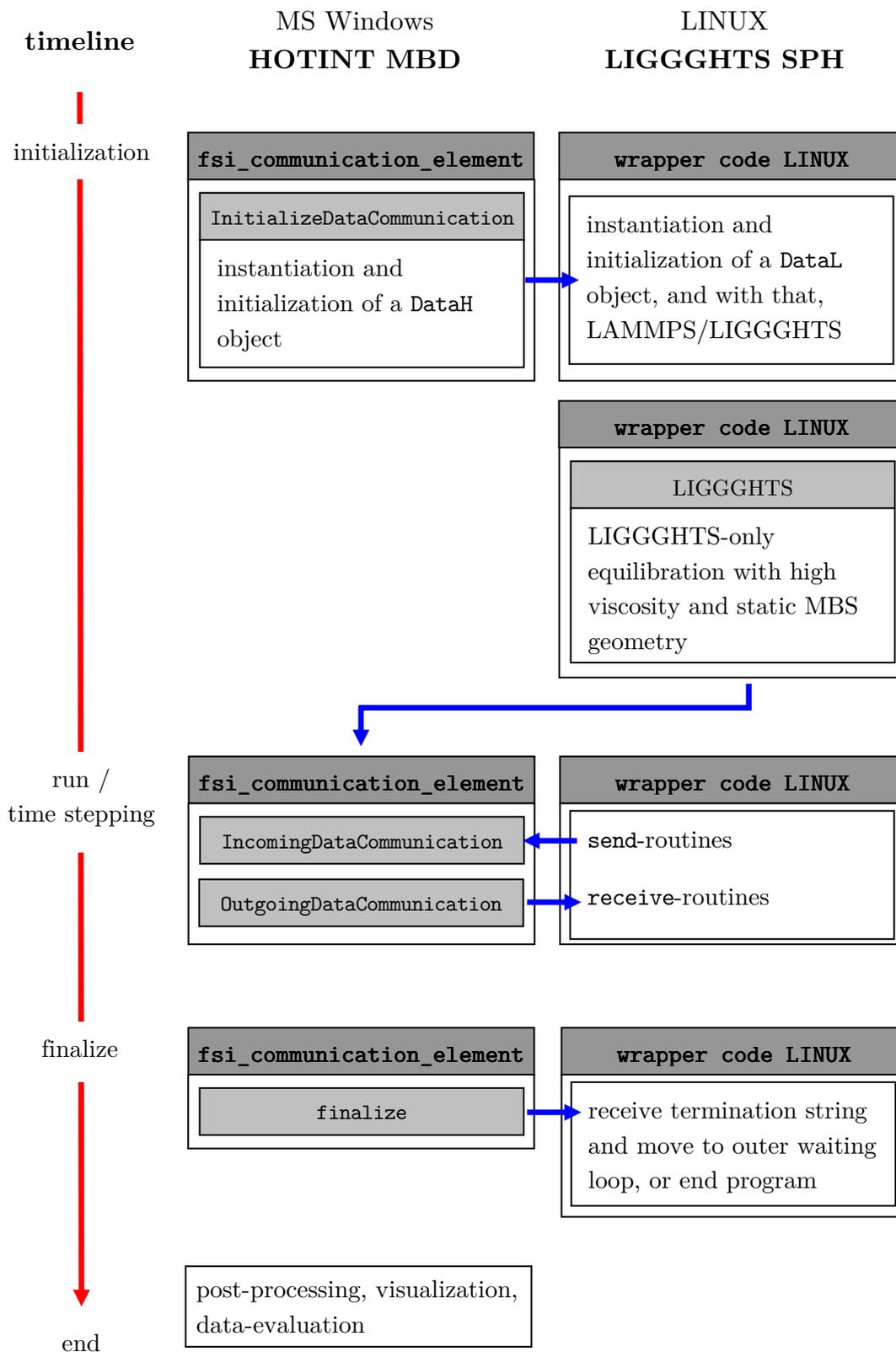
Figure 5.11.: Block diagram of the LINUX-sided wrapper program.

## 5.6. Initialization and the coupled program flow

Concluding above detailed discussion of the implementation, the whole process of setting up and defining a problem, followed by the initialization procedure and the coupled program flow is sketched in the Figures 5.12 and 5.13.



**Figure 5.12.:** Block diagram and time line of the process of problem set-up, initialization and the coupled application, from set-up to application start-up; \* for examples for the HOTINT parameter file and LIGGGHTS input scripts refer to the text.



**Figure 5.13.:** Block diagram and time line of the process of problem set-up, initialization and the coupled application, from initialization and equilibration to the end of the simulation and data evaluation/ post-processing; the blue arrows indicate the TCP/IP data transfer.

An example of a HOTINT parameter file and LIGGGHTS input scripts, corresponding to the simulation example of Section 6.3, is shown below, with short notes to the various quantities via comments which are displayed in green and indicated by “`%//`” HOTINT-sided, and by “`#//`” LIGGGHTS-sided. Note that first the HOTINT parameter file is read, then, on that basis, an autogenerated LIGGGHTS inputscript is generated and read in, followed by the first LIGGGHTS inputscript (“LIGGGHTS inputscript 1”), the LIGGGHTS-only equilibration phase with the static initial configuration of the MBS, and finally, the second LIGGGHTS inputscript (“LIGGGHTS inputscript 2”).

### HOTINT parameter file

```

1  %// model data for static consistency test simulation
2
3  SolverOptions
4  {
5    end_time = 0.7 %// total simulation time
6    Timeint
7    {
8      max_step_size = 0.5*1e-5 %// maximum time step size
9      min_step_size = max_step_size %// minimum time step size
10     tableau_name = "LobattoIIIA" %// integration scheme (LobattoIIIA)
11     max_stages = 2 %// maximum number of stages of the integration scheme
12     do_implicit_integration = 1 %// use implicit integration (default)
13   }
14 }
15 Solution
16 {
17   write_solution_every_x_step = 10
18   store_data_every = 0.001 %// interval for saving solution data and full
19     exchange of SPH data
20 }
21 Linalg
22 {
23   use_sparse_solver= 0 %// 1|(0) ... Sparse Jacobian and sparse solver is (
24     not) activated
25 }
26
27 Newton.max_modified_newton_steps= 40 %// maximum modified Newton steps.
28
29 LoggingOptions.output_level = 6 %// defines a level of output in the log (how
30   much information on the solution procedure is written)

```

```

31 width = 0.05    ///< height of surrounding box
32 height = 0.1   ///< width of surrounding box
33 fill_level = 0.4*height ///< filling level
34 piston_thickness = 0.005 ///< thickness of piston
35 piston_y = 0.75*height-0.5*piston_thickness ///< height of piston
36 piston_v0 = 0.5*height          ///< initial, constant piston velocity in
    -y-direction
37
38 offset_x = 0.0 ///< offset of coordinate system
39 offset_y = 0.0
40 }
41
42 LIGGGHTS_SPH_parameters
43 {
44   SPHdensity = 1000.    ///< nominal SPH density
45
46   smoothinglength = 0.0015 ///< SPH smoothing length; kernel support radius
    is 2*smoothinglength
47
48   particlespacing = smoothinglength/1.9 ///< initial distance between SPH
    particles on regular square lattice
49     ///< (set such that ~30-70 particles lie within the kernel support
    domain)
50
51   SPHparticlemass = SPHdensity*particlespacing*particlespacing ///< mass (here
    (2D): mass per depth unit) of SPH particles
52
53   cAB = 20.0          ///< (artificial) speed of sound in artificial viscosity;
    approx 10 times the actual maximum flow velocity
54   viscosity = 1e-3    ///< kinematic viscosity
55   highviscosity = 100.0*viscosity ///< high viscosity for LIGGGHTS-only
    equilibration
56
57   xmin = 0. + Geometry.offset_x -0.01 ///< definition of the LIGGGHTS
    simulation domain
58   xmax = 0.05 + Geometry.offset_x_
59   ymin = 0.0 + Geometry.offset_y
60   ymax = 0.1 + Geometry.offset_y
61   zmin = -10.0        ///< in 2D case zmin and zmax are ignored
62   zmax = 10.0
63
64   SPH_wall_cutoff = 2.0*smoothinglength ///< cut-off radius for SPH-wall
    contact
65   SPH_wall_equilibriumdist = 1.0*SPH_wall_cutoff ///< equilibrium distance for
    interaction force (0...r0 repulsion, r0...rc attraction (adhesion))

```

```

66 SPH_wall_rep = 17500.    ///< scaling parameter for repulsive force
67 SPH_wall_visc = SPHdensity*viscosity/smoothinglength ///< scaling parameter
    for wall friction / viscous contact force
68
69 equilibration_steps = 100000 ///< number of equilibration steps
70
71 refinement_option = 2 ///< specification of the refinement option:
72     ///< for 3D: 0...no refinement, 1... recursive refinement based on
    original triangles in every time step,
73     ///<     2... pre-refined mesh, no additional refinement in time-
    stepping, 3... as 1, but based on pre-refined mesh;
74     ///< for 2D: only option 1
75     ///< default is 1
76
77 refinement_resolution = 4.0*smoothinglength ///< specification of the
    refinement depth / resolution
78     ///< the mesh is refined in 2D (3D) until any line element (edge
    of a triangle) is shorter than dr
79
80 }
81
82 TCP_data
83 {
84     port = 12345 ///< port of the server socket
85     ip1 = 192 ///< TCP/IP v4 adress of the server socket, given by ip1.ip2.ip3.
        ip4
86     ip2 = 168
87     ip3 = 56
88     ip4 = 1
89 }

```

### LIGGGHTS inputscript 1

```

1  ///< variable definitions
2  ///< any variables not defined here have to be defined in the HOTINT parameter
    file, and via that included in the auto-generated part of the LIGGGHTS
    input script
3
4  variable    skin equal ${smoothinglength}*0.25          ///< parameter for
    neighbor-list builds
5  variable    eta equal 0.01*${smoothinglength}*${smoothinglength} ///< parameter
    for artificial viscosity
6  variable    aux equal ${cAB}*${smoothinglength} ///< parameter for artificial
    viscosity
7  variable    alpha equal ${viscosity}/${aux}
8  variable    alphatemp equal ${highviscosity}/${aux} ///< parameter of

```

```

        artificial viscosity in the high-viscosity LIGGGHTS-only equilibration
        phase
9  variable    zdim equal 0.25*${smoothinglength}    ///< pseudo-depth for the 2D
        case (used for the cell-grid generation only)
10
11 ///< parameters for Tait's equation (see below)
12 variable    gamma equal 7.0
13 variable    b0 equal ${SPHdensity}*${cAB}*${cAB}/${gamma}
14
15 dimension  2    ///< set problem dimensionality to 2D
16 atom_style sph ///< use SPH formalism for the particle simulation
17 atom_modify map array sort 0 0
18 communicate single vel yes
19
20 boundary    f f p ///

```

```
viscosity
38
39 /// temporary viscosity in alphatemp >> alpha (mutemp >> mu) for fast
    equilibration
40 /// pair_style is reset in the second input script using the actual viscosity
41
42 pair_coeff * * /// use the same interaction between all types of particles (
    here only one type, anyways)
43
44 /// definition of SPH fixes
45 /// density
46 fix          density all sph/density/continuity /// specification to use the
    continuity-approach-based density calculation
47 /// fix          density all sph/density/summation /// this would be the
    summation density approach
48 fix          corr all sph/density/corr shepard every 20 /// apply the Shepard
    filter on the density field every 20 time steps
49
50 /// pressure / equation of state
51
52 /// fix          id group style type [if Tait: B rho0 gamma] (according to
    Monaghan 1994)
53 /// B = c^2*rho0/gamma
54 fix          pressure all sph/pressure Tait  $\rho_0$   $\rho_0$   $\gamma$ 
55
56 /// FSI fix
57 fix          fsi all wall/sph_fsi  $\rho_0$   $\rho_0$   $\rho_0$   $\rho_0$ 
     $\rho_0$   $\rho_0$ 
58
59 /// time integration
60 fix          integr all nve /// use an NVE integrator (constant total energy,
    volume and number of particles (microcanonical ensemble))
61
62 /// gravity
63 fix          gravi all gravity 9.81 vector 0.0 -1.0 0.0 /// set the vector for
    gravitational acceleration in -y-direction
64
65 /// enforce 2D
66 fix          enf all enforce2d ///forces the z-component of positions,
    velocities, and forces to be 0
67
68 /// output settings, include total thermal energy
69 thermo_style custom step #atoms ke vol cpu /// defines the output style of
    simulation data during the running simulation
70 thermo          1          /// output frequency (interval defined in time steps)
```

```
71
72 thermo_modify lost warn #// releases a warning when particles are lost, i.e.
    when particles move outside the simulation domain
73
74 #// dump      dmp all custom 250 dump2D.sph id type x y z ix iy iz vx vy vz fx
    fy fz q density #// specification of the simulation data file output
```

## LIGGGHTS inputscript 2

```
1 #// this file is read in after equilibration
2 #//every input line here either is additional to LIGGGHTS_input_script_2d or
    overwrites/changes commands specified there
3
4 pair_style sph spiky2D ${smoothinglength} artVisc ${alpha} 0. ${cAB} ${eta} #
    // overwrite temporary pair style (high viscosity for equilibration) with
    pair style with actual viscosity
```

## 6. Example problems and simulations

### 6.1. Introduction

After the implementation of any numerical approach it is essential to investigate its properties and performance by means of test simulations and the investigation of simple problems with existing reference solutions (possibly even analytical solutions), with the focus on stability and consistency, as well as computational performance. In our case, furthermore, a number of unknown parameters have been introduced which need to be defined reasonably in each test example, or ideally, via a standard choice or an approximate analytical expression (depending on other parameters) applicable in any case. Additionally, the sensitivity of the system and the resulting numerical solution with respect to those parameters should be analyzed. It is the iterative process of testing, debugging, the identification of arising problems – or even, of generally problematic system configurations in the representation of the respective approach – and the corresponding improvements, which piece by piece eventually (may) lead to a reliable, stable and consistent formulation and implementation.

To this end, the next subsections contain an outline about the choice of various parameters and the issue of stability, concludingly followed by three numerical 2D-examples for testing and verification. The latter simulations were performed on one of two different hard- and software configurations: The first one has already been discussed in Section 5.1, with an Intel i7 2720QM CPU (4 real physical cores + 4 cores with hyperthreading) and 8GB DDR3 RAM; the other consists of actually two machines directly coupled via a real 1 Gbit/s network connection, one with MS Windows XP as operating system for HOTINT, with an Intel Q6600 CPU, and the other one with a LINUX Ubuntu distribution, powered by an Intel i7 2600k CPU (as above, 4 physical cores + 4 hyperthreaded cores).

As a side note concerning CPU time and an upper limit of the problem size, with above setup it is possible to perform simulations of systems roughly consisting of up to  $\approx 100000$  SPH particles and  $\approx 10000$  surface elements associated with a multibody system represented by hundreds of degrees of freedom, for a simulated physical time in the range of few seconds (corresponding to several 100000 time steps) within reasonable time, i.e. several hours up to several days. Of course, the computational costs actually depend on the very specifics of the problem itself, such as the size and number of time steps, which components the MBS consists of, how fast the solver can handle the resulting system of equations of the MBS, or furthermore, on the convergence or accuracy goals predefined by the user. Hence, above

example should only be considered as a rough estimate – a reference value for the order of magnitude.

## 6.2. Details on the configuration - parameters and stability

As already mentioned in the introduction above, a range of parameters and unknowns come with this approach – in particular, with the method of SPH, such as the scaling factors of the fluid-structure interaction forces, the parameters of the artificial viscosity, the kernel along with the smoothing length, the mass of the SPH particles, or the scalar equation of state with corresponding parameters. For the configuration of the test examples discussed here (and all others) the following choices and estimates apply:

- **SPH smoothing kernel:** Based on the originally implemented cubic spline kernel in 3D (cf. equation (3.22)), its 2D-version, as well as the spiky kernel for 2D and 3D were additionally implemented in LIGGGHTS. However, after several test simulations the cubic spline kernel showed to be problematic, especially in regions close to the boundaries in states close to equilibrium (almost static states), since there it lead to numerical divergences. The reason for those problems is the fact that its gradient  $\nabla_{\mathbf{r}_j} W(\mathbf{r}_i, \mathbf{r}_j, h)$  vanishes when the relative particle distance  $|\mathbf{r}_i - \mathbf{r}_j|$  approaches zero (cf. Figure 3.1), which results in vanishing forces due to the pressure (cf. the terms proportional to  $p_i$  and  $p_j$  in equation (3.36)). Because of that, in particular in vicinity to an equilibrium state with (relative) velocities approaching zero, and thus, vanishing viscous forces, when two SPH particles get too close to each other, their mutual repulsion approaches zero, and there is nothing left to keep those particles spatially separated. This can (and did) result in an “exact” overlap of particles (meaning that  $|\mathbf{r}_i - \mathbf{r}_j|$  becomes smaller than the resolution limit of double precision numbers,  $\approx 10^{-16}$ ), leading to divisions by zero in the LIGGGHTS SPH implementation and thus “NAN”s (“not a number”) in the respective field quantities. Therefore, from that point on only the spiky kernel was used, which does not produce such instabilities due to its continuously increasing absolute value of the gradient with decreasing inter-particle distance (cf. Figure 3.2).
- **SPH smoothing length:** The constant, universal smoothing length  $h$  was chosen such that the average number of particles within the kernel support domain (support radius  $2h$ ) lay in the range of 50 – 100. Given an average particle distance  $d_{av}$  (see the next point below),  $h \approx 1.6 \dots 2.6 \cdot d_{av}$  for 2D and  $h \approx 1.1 \dots 1.4 \cdot d_{av}$  for 3D.
- **Initial particle configuration – mass / density / average distance of SPH particles:** During several test runs, the following procedure performed best: Creation of an initial particle configuration according to a regular lattice with the lattice constant  $d$  and an additional (optional) random displacement from those reference positions. The initial average distance between the SPH particles approximately is

$d_{av} \approx d$ , thus the associated initial volume is  $d_{av}^2$  ( $d_{av}^3$ ) for 2D (3D), and the corresponding mass is set to  $m = \rho_0 d_{av}^2$  ( $\rho_0 d_{av}^3$ ), based on a constant initial nominal density  $\rho = \rho_0$  for all particles. For some further information on the initial particle configuration in the context of the equilibration procedure, see also the discussion of `InitializeDataCommunication` in Subsection 5.4.2.

- **Viscosity:** In LIGGGHTS SPH the artificial viscosity according to Monaghan and Gingold [18] is implemented (cf. equation (3.37)). Here,  $\beta = 0$ ,  $\eta = 0.1h$ , and  $\alpha = \frac{\nu}{hc}$  (according to equation (3.42)) were used, with the smoothing length  $h$ , the speed of sound  $c$  (see the next point below), and the kinematic viscosity  $\nu$ .
- **Equation of state:** As already discussed in Subsection 3.2.3, for modelling weakly compressible fluids, Tait’s equation of state was employed (cf. equation (3.53)), using  $\gamma = 7$ , the nominal fluid density  $\rho_0$  and the speed of sound  $c$  which was chosen approximately 10 times as large as the expected/estimated bulk flow velocity. The latter should result in maximum density variations in the range of a few percents (cf. equation (3.54)).
- **Density computation:** Starting with a given, initial density distribution, the integral approach based on the continuity equation (cf. equation (3.31)) was used for the calculation of the density field; additionally, in order to prevent oscillations, a smoothing filter (“Shepard filter”, see Subsection 3.2.3 for more information) was applied once every 20-30 time steps.
- **Size of the time step:** The maximum time step size (for both sides) was chosen in the range of  $10^{-5} \dots 10^{-6}$  s, depending on the bulk modulus (stiffness) of the fluid which is determined by the speed of sound (cf. the point “Equation of state” above). As already discussed, this very small step size is due to the explicit integration routines on the fluid side. As a side note: One useful indicator for a time step size chosen too large in this implementation are significant (unphysical) variations in the density field.
- **Parameters for the fluid-structure contact force field densities:** Here, the four parameters  $k$  (scaling factor of the repulsive force),  $t$  (scaling factor of the viscous force),  $r_0$  (equilibrium distance) and  $r_c = 2h$  (the cut-off or interaction range) – cf. equation (4.2) – have to be determined. In the test simulations, adhesive effects should not be considered, thus  $r_0 = r_c$ ; the cut-off radius  $r_c$ , apart from the scaling parameters, significantly determines the stiffness of the contact, since for a given system, there exists a lower limit for  $k$  in order to retain the “no-penetration” condition (equation (3.44)), i.e. to keep particles from moving through the boundaries. Hence, very small values for  $r_c$  result in very stiff fluid-structure contact, which must be accounted for on the fluid side by using sufficiently small time steps; on the other hand, the smaller  $r_c$  becomes, the higher the accuracy of the representation of the actual boundary shapes gets. Therefore, the choice of  $r_c$  always is some sort of compromise;

typical values lie in the range of the SPH smoothing length (or kernel support radius). Note that, after the definition of  $r_c$ , the refinement resolution  $\mathbf{dr}$  also is chosen in that range (cf. Subsections 5.5.2 and 4.3).

The scaling parameter  $k$  – corresponding to a pressure – must be defined individually in each case, since it heavily depends on the specific problem situation. If the gravitational force is considered, the maximum hydrostatic pressure in the static case, given by  $\rho_0 g h_{max}$  (with the maximum “filling height”  $h_{max}$  and nominal density  $\rho_0$  of the fluid, and the gravitational acceleration  $g \approx 9.81 \text{ m/s}^2$ ) can be used as an estimate for the lower limit of  $k$ . A reasonable choice for  $k$  in such cases is that hydrostatic pressure multiplied by a some factor in the range of  $2 - 20$ ,  $k \approx 2 \dots 20 \cdot \rho_0 g h_{max}$ . However, this is only a rule of thumb, and usually some test runs are necessary in order to find an appropriate value which ideally should be as small as possible (i.e., just large enough to prevent the particles from penetrating the boundaries).

Concerning the scaling parameter of the visous force terms, an estimate for a reasonable value can be calculated based on the assumption that the viscous forces between two SPH particles  $i$  and  $j$ , here designated as  $\mathbf{f}_{visc,SPH}$ , should be approximately equal to those between a local boundary point  $i$  (from the numerical surface integration of line segment  $m$ ) and one SPH particle  $j$ , denoted as  $\mathbf{f}_{visc,wall}$  (cf. equations (3.36) and (4.4), as well as (5.5)):

$$\left| \mathbf{f}_{visc,SPH} \right| = m_i m_j \Pi_{ij} \cdot |\nabla_{\mathbf{r}_i} W_{ij}| \approx m^2 \frac{\nu}{\rho h} v_{ij} |\nabla_{\mathbf{r}_i} W_{ij}| \quad (6.1)$$

$$\left| \mathbf{f}_{visc,wall} \right| = \frac{1}{2} l_m w_i \left| \mathbf{f}^{visc}(\mathbf{r}_j, \mathbf{v}_j, \mathbf{r}_{m,i}^S, \mathbf{v}_{m,i}^S) \right| \approx t v_{ij} \frac{1}{h} |\nabla_{\mathbf{r}_i} W_{ij}| h^5, \quad (6.2)$$

where again,  $|\Delta_{\mathbf{r}_i} W_{ij}| \approx |\nabla_{\mathbf{r}_i} W_{ij}|/h$  was used, and  $h$  for all occuring characteristic distances. Now, from  $\left| \mathbf{f}_{visc,SPH} \right| \approx \left| \mathbf{f}_{visc,wall} \right|$  we get

$$t \approx m^2 \frac{\nu}{\rho h^5}, \quad (6.3)$$

and with  $m \approx \rho h^2$  (in 2D) finally

$$t \approx \frac{\nu \rho}{h} \quad (6.4)$$

with the kinematic viscosity  $\nu$  and the nominal fluid density  $\rho \approx \rho_0$ . Note that above derivation was done for the 2D case, but can be done for 3D in analogy, yielding with equation (6.4) an identical final result.

### 6.3. Volume, density, and weight - a static consistency test

The first thing that was investigated quantitatively was the actual volume (per depth unit) of a fluid filled into a box in equilibrium. Additionally, the weight force exerted by the

fluid on the bottom of the box was determined, and the result then should be compared to the theoretical weight force which is defined exactly by the number of particles and their constant equal masses, as well as to the weight determined from the measured volume and the nominal fluid density.

At first, this might seem to be a trivial consideration, but keep in mind that in the formalism of the meshfree, particle-based method SPH the macroscopic volume, as well as the surface of the fluid, are not defined exactly. Hence, it should not be assumed a priori that the simple relation of mass, density, and volume is consistently represented here, even though it is a very basic requirement for the methods viability.

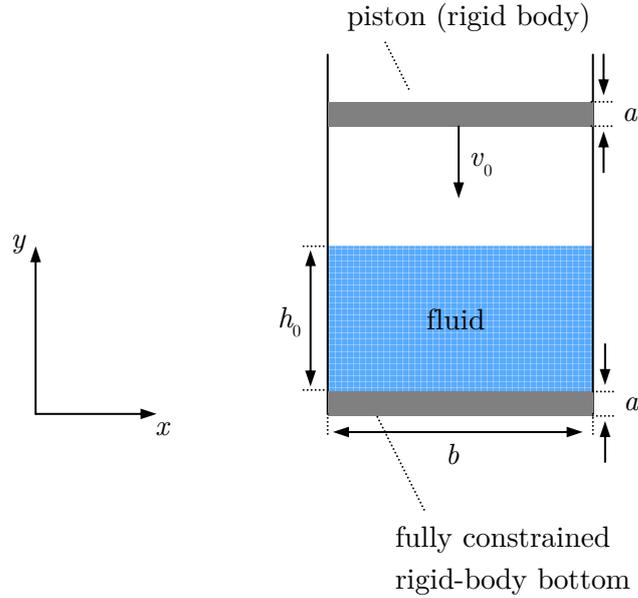
The test simulation is based on the set-up sketched in Figure 6.1, consisting of the components listed below – of course including gravitational force; a snapshot of the simulation for illustration of the “real” set-up is shown in Figure 6.2:

- Two impervious walls at the side,
- one rectangular rigid body forming the bottom of the box, with a position constraint for all three of its degrees of freedom (the position of its center of gravity (two coordinates  $x_b$  and  $y_b$ ), as well as the rotation angle  $\varphi_b$  about the center of gravity),
- a rigid body piston, moving towards the bottom with a constant velocity  $v_0$  in  $-y$ -direction, enforced by a coordinate constraint of its  $x$ -position  $x_p$  as well as the rotation  $\varphi_p$ , and a velocity constraint  $\dot{y}_p = v_0$ , with a position sensor to measure the piston displacement,
- and finally, the fluid (SPH particles) enclosed by those four components of the MBS.

Now, in this case without a LIGGGHTS-only equilibration phase, the piston moves towards the fluid, and until contact with the SPH particles the average total force on the rigid body at the bottom corresponds to the weight of the fluid. As soon as the piston starts to effectively compress the fluid, the measured forces increase rapidly, where the point of this transition can be used to identify the effective fluid volume actually represented by the SPH particles. It should be noted that in this example the measurement of the forces in HOTINT is performed using sensor elements evaluating the Lagrange multipliers corresponding to the respective kinematic constraints of the rigid bodies (constraint forces, cf. equation (2.2)).

The following parameters were used in the simulation:

- piston thickness  $a = 0.005$  m and width  $b = 0.05$  m (cf. Figure 6.1)
- initial  $y$ -coordinate of the pistons center of gravity:  $y_p(t = 0) = 0.0725$  m
- constant  $y$ -coordinate of the bottoms center of gravity:  $y_b = 0$  m
- constant piston velocity:  $v_0 = 0.05$  m/s
- SPH smoothing length:  $h \approx 0.000614$  m
- mass (per depth unit) of SPH particles:  $m \approx 0.0001053$  kg/m



**Figure 6.1.:** Sketch of the test example for the investigation of the relation of mass/volume/density and weight force of the fluid: The piston moves with constant velocity  $v_0$  in  $-y$ -direction,  $a$  denotes the thickness of the piston and the bottom,  $b$  the width of the box and  $h_0$  the effective equilibrium filling height of the fluid.

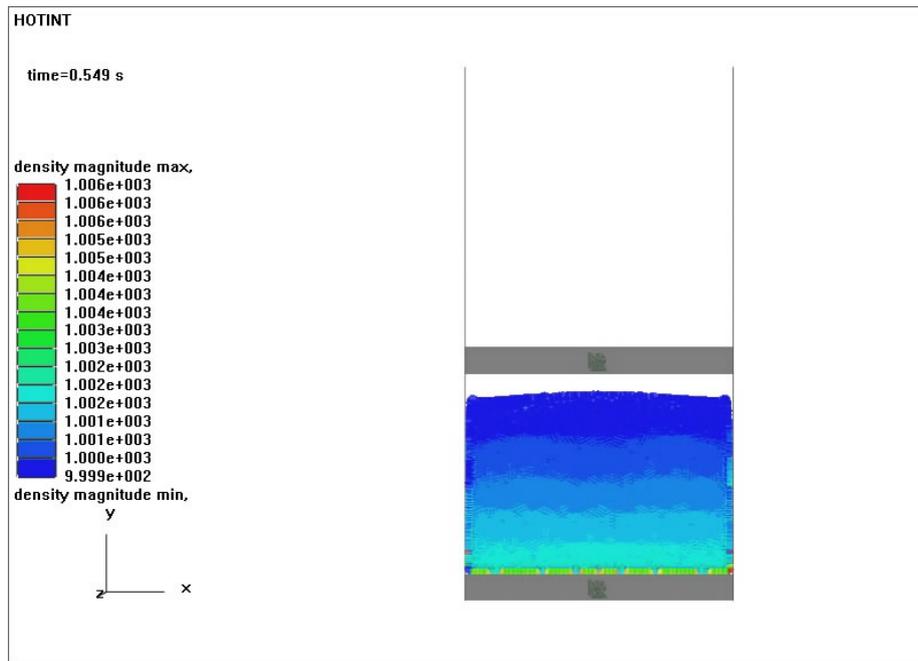
- number of SPH particles:  $N = 14298$
- nominal fluid density:  $\rho_0 = 1000 \text{ kg/m}^3$

The sensor data, i.e. the displacement of and total force in  $y$ -direction on the piston, as well as the force (in  $y$ -direction) on the bottom, are shown in the Figures 6.3 to 6.5. Without any LIGGGHTS-only equilibration, significant fluctuations in all quantities can be observed. This should demonstrate that, if an initial situation close to equilibrium of the fluid is required and/or important, a preceding equilibration process is essential. In this case, however, we can still use the average value of the weight force (Figure 6.5) as a good approximation for its equilibrium value  $F_0$ , as well as determine the point (simulation time  $t_c$ ) of the onset of effective compression of the fluid by the piston from Figure 6.4 (and then from Figure 6.3, or via  $v_0$ , the effective equilibrium filling height  $h_0$ ) with sufficient accuracy.

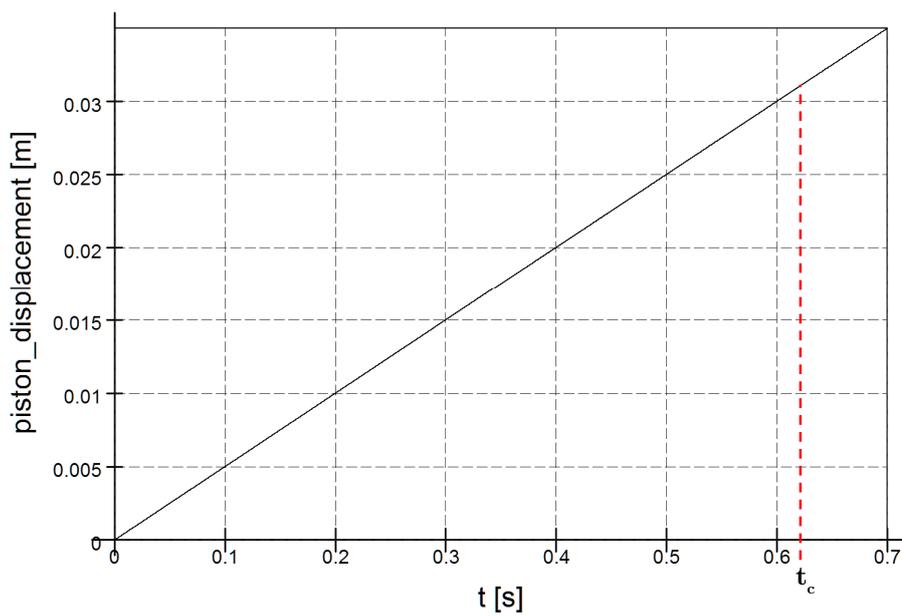
Based on the obtained simulation data, the equilibrium filling height  $h_0$  is given by

$$h_0 = y_p(0) - 1.5a - \Delta y_p \approx 0.034 \text{ m} \quad (6.5)$$

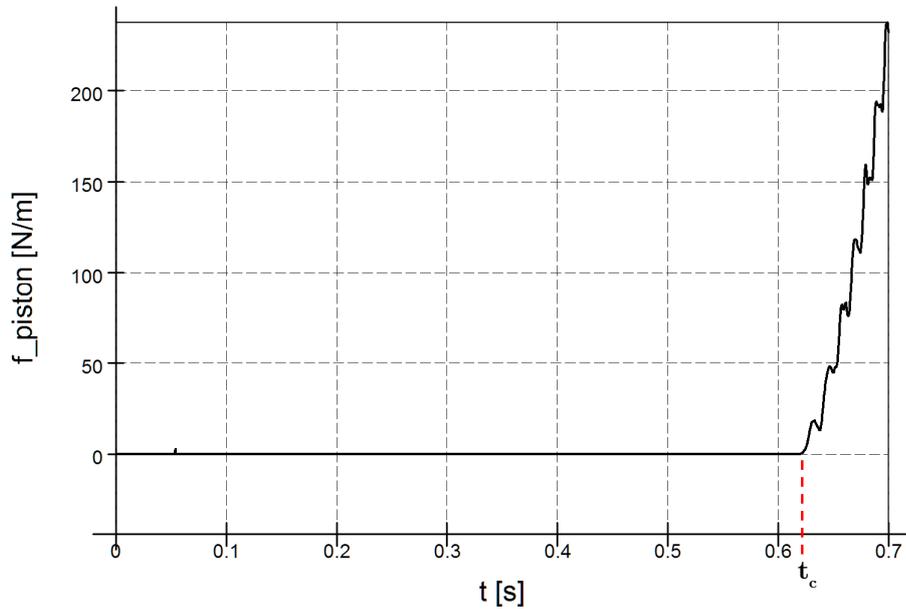
with the piston displacement  $\Delta y_p$  at the compression limit. Since the scaling parameter for the repulsive fluid-structure contact force was chosen very high in this example (allowing for test simulations with high compression of the fluid), thus resulting in a very high contact



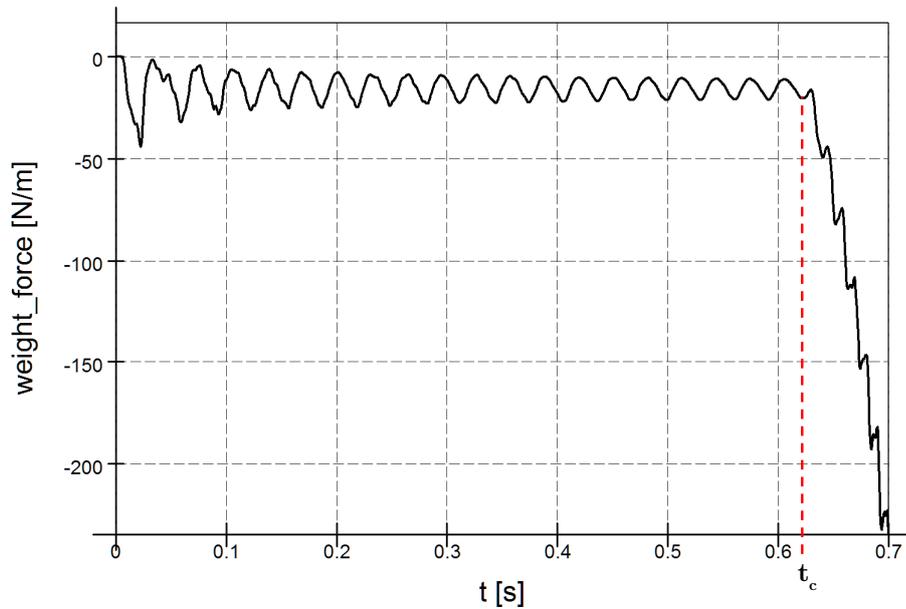
**Figure 6.2.:** A snapshot of the dynamic simulation of this experimental set-up at  $t = 0.549$  s for further illustration (cf. also the sketch in Figure 6.1), where the colors correspond to the density field of the fluid. As expected, the density slightly increases towards the bottom, however, in immediate vicinity of the boundaries some irregularities are observed originating from too stiff fluid-structure contact, i.e. too small interaction range and/or too large force scaling parameters.



**Figure 6.3.:** Piston displacement (absolute value) in  $-y$ -direction versus simulation time;  $t_c \approx 0.62$  s is the time at which the compression limit (effective compression of the fluid by the piston) is reached (cf. Figure 6.4), with a corresponding displacement of  $\Delta y_p = v_0 t_c = 0.031$  m. Moreover, sensor data like this can be conveniently used to double-check if the system behaves as it should, i.e., in this case, a linear relation between the displacement and simulation time.



**Figure 6.4.:** Total force on the piston due to the interaction with the fluid in  $y$ -direction versus simulation time; from this,  $t_c$  – the time at which the compression limit (effective compression of the fluid by the piston) is reached – was determined as  $t_c \approx 0.62$  s. From that point on, significant oscillations of the force due to oscillations of the fluid density field can be observed, which originate from the non-equilibrium initial state as well as from the impact of the piston on the fluid.



**Figure 6.5.:** Weight force ( $y$ -direction) of the fluid acting on the rigid body on the bottom versus simulation time;  $t_c \approx 0.62$  s is the time at which the compression limit (effective compression of the fluid by the piston) is reached (cf. Figure 6.4).

stiffness, for the calculation of the effective volume accessible for the SPH particles in a state of equilibrium we do not use  $h_0b$ , but a corrected value via the smoothing length,

$$V \approx (h_0 - 2h)(b - 2h) \approx 0.001598 \text{ m}^3/\text{m}, \quad (6.6)$$

yielding an absolute value of the total weight force (per depth unit) of

$$F_{weight}^V \approx \rho_0 V g \approx 15.68 \text{ N/m} \quad (6.7)$$

of the fluid, where  $g = 9.81 \text{ m/s}^2$  and  $\rho_0$  was used as average density considering the weak compressibility. The averaged value for this weight force determined from the sensor data (cf. Figure 6.5) via an average over all full oscillation periods within the interval  $0.25 \text{ s} \leq t \leq 0.6 \text{ s}$  was obtained as

$$F_{weight}^{sensor} \approx 15.42 \text{ N/m}, \quad (6.8)$$

where the identification of those full periods was performed using a the mean value over the whole time interval as preliminary average for  $F_{weight}^{sensor}$ . For comparison, the exact value is given by

$$F_{weight}^{exact} = Nmg \approx 15.43 \text{ N/m}, \quad (6.9)$$

which finally yields

$$F_{weight}^V / F_{weight}^{exact} \approx 1.0159 \quad (6.10)$$

$$F_{weight}^{sensor} / F_{weight}^{exact} \approx 0.999. \quad (6.11)$$

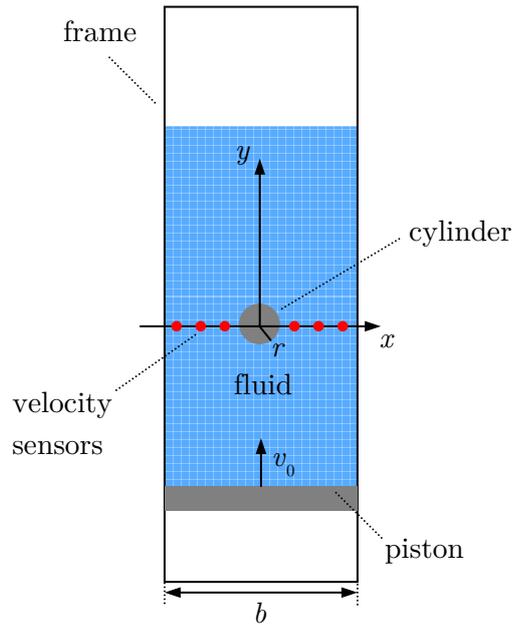
Thus, with relative errors in the low percent or sub-percent range, the method produced consistent results in this static test example for the effective fluid volume as well as the corresponding weight force.

## 6.4. Laminar flow around a cylinder

As one example of classic problems in fluid dynamics, the investigation of the laminar flow around a cylinder was chosen mainly to verify the fluid side of the developed approach, i.e. the results produced by LIGGGHTS SPH. To this end, a reference solution was generated with a classical finite volume simulation, using the solver ANSYS Fluent 6.3 with a 2D mesh generated in Gambit; some further details follow below.

The system set-up for HOTINT/LIGGGHTS, outlined and sketched in Figure 6.6, consists of

- a surrounding box of width  $b = 0.20 \text{ m}$  (effective width:  $\tilde{b} \approx 0.18 \text{ m}$ , cf. explanation below in the text) partly filled with fluid (9048 SPH particles, generated on a regular grid with small random displacements),



**Figure 6.6.:** Sketch of the system set-up: The piston smoothly is accelerated to a constant velocity  $v_0 = 0.1$  m/s in  $y$ -direction, and moves the fluid along the channel of width  $b = 0.2$  m (effective width  $\tilde{b} \approx 0.18$  m, see text for details) past the cylinder (radius  $r \approx 0.014$  m); in the symmetry plane, here given by the  $x$ -axis, 13 sensors for measurement of the fluid velocity – to be exact, of the spatial average over the velocities of all those particles which lie within a distance of  $2h$  ( $h$ ... SPH smoothing length) to the respective sensor – were placed equidistantly along the line from left to right. All components of the multibody system are rigid and, except for the piston, static (fixed to the ground); gravity was included ( $-y$ -direction).

- a cylinder with effective radius  $r \approx 0.014$  m in the center, and, for the generation of a flow past that cylinder against the gravity field,
- a piston which is accelerated smoothly from 0 to the afterwards constant velocity  $v_0 = 0.1$  m/s in  $y$ -direction during the first second of the simulation, initially positioned 0.64 m below the center of the cylinder, and
- 13 sensors at equidistant positions on the  $x$ -axis over the cross section which measure the fluid velocity in  $y$ -direction, spatially averaged over all particles within a distance of  $2h$  (with the SPH smoothing length  $h$ ) to the respective sensor.

Consequently, the geometry of the simulation domain for the finite volume reference solution was box-shaped, with  $\tilde{b} \approx 0.18$  m width, and 0.7 m height, with a cylinder of equal radius  $r \approx 0.014$  m in the center. Instead of the piston, a Dirichlet boundary condition of constant velocity  $v_0 = 0.1$  m/s in  $y$ -direction at the bottom of the box was specified, along with a constant pressure (ambient pressure) at the boundary on the opposite side. Concerning the solver configuration, second-order algorithms were used for both the upwind scheme for the convective terms as well as pressure interpolation, together with the SIMPLE algorithm

for the decoupled computation of the pressure and velocity field. The (static) finite volume mesh was generated in Gambit, using a total of 1808 quadrilateral cells

The fluid parameters on both sides – the coupled HOTINT/LIGGGHTS simulation as well as the reference finite volume simulation – are given by

- the nominal density  $\rho_0 = 1000 \text{ kg/m}^3$ , and
- a kinematic viscosity of  $\nu = 10^{-3} \text{ m}^2/\text{s}$ ,

including the gravitational force.

With the flow velocity at the inlet, corresponding to the stationary piston velocity, and the diameter of the cylinder the Reynolds number of the problem is given by

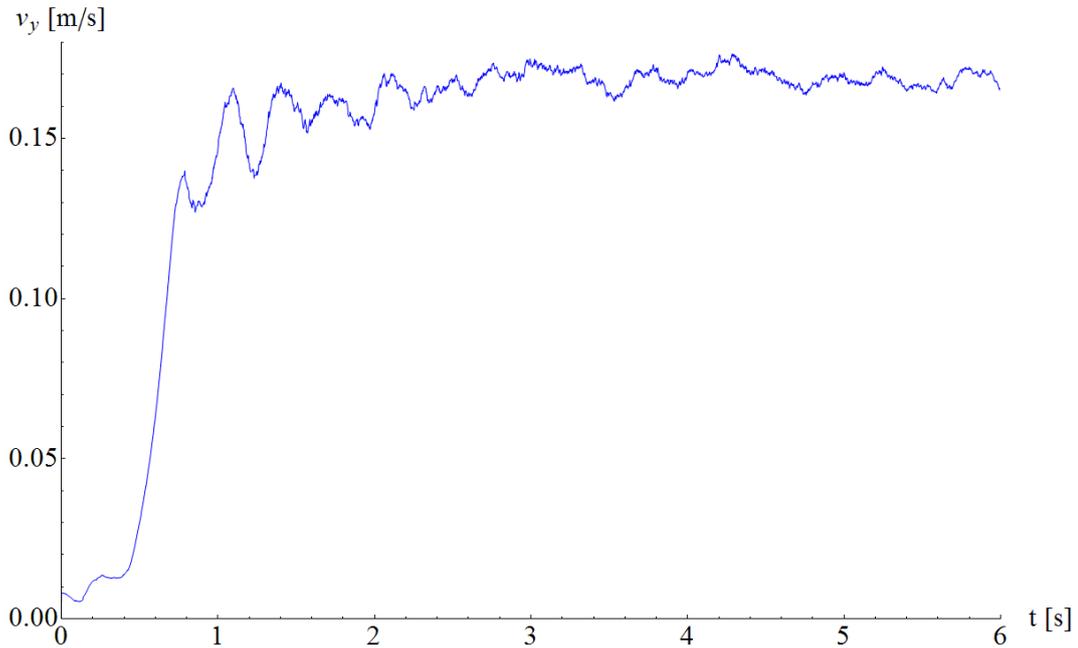
$$\text{Re} = \frac{2v_0r}{\nu} \approx 2.8, \quad (6.12)$$

which leaves us with a non-separated, laminar flow situation. For the latter,  $\text{Re} = 5$  would be the limit; in the range  $5 \leq \text{Re} \leq 40$  spatially fixed symmetrical vortices develop behind the cylinder, followed by the formation a laminar vortex street for  $40 \leq \text{Re} \leq 200$  and the transition to turbulence in the wake at  $200 \leq \text{Re} \leq 300$  [55].

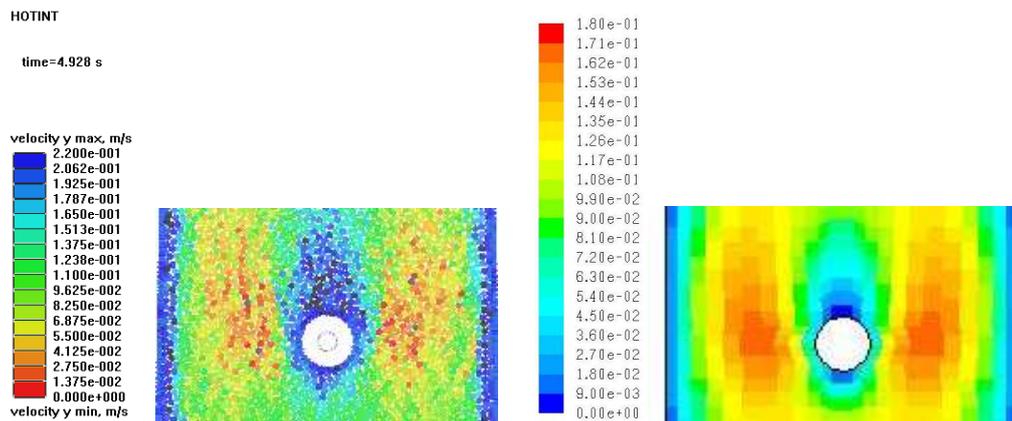
Note that above “effective” dimensions were introduced in order to account for the – in this case – relatively large interaction range of the repulsive fluid-structure contact force, given by  $r_c = 2.5h$  with the SPH smoothing length  $h \approx 0.0088 \text{ m}$ , which, together with the specified force scaling parameter, results in additional space of approximately  $0.1 \text{ m}$  ( $\approx h$ ) between the SPH particles and the boundaries in the quasi-stationary state. In other words, the boundaries effectively generated on the fluid side are expanded in all directions by that distance. The reason for the choice of a large interaction range was to produce a less stiff fluid-structure contact which turned out to be essential for consistent results, particularly in the vicinity of the boundaries. If the contact is too stiff, the SPH particles tend to form a dense line adjacent (parallel) to the side walls which then is forced to move with the same velocity as the piston, thus compromising the no-slip condition. Furthermore, this effect is the source of significant perturbations, generated each time one particle jumps out of such a line.

The results of the simulation as well as a comparison with the reference solution are shown and briefly discussed in the Figures 6.7 to 6.9.

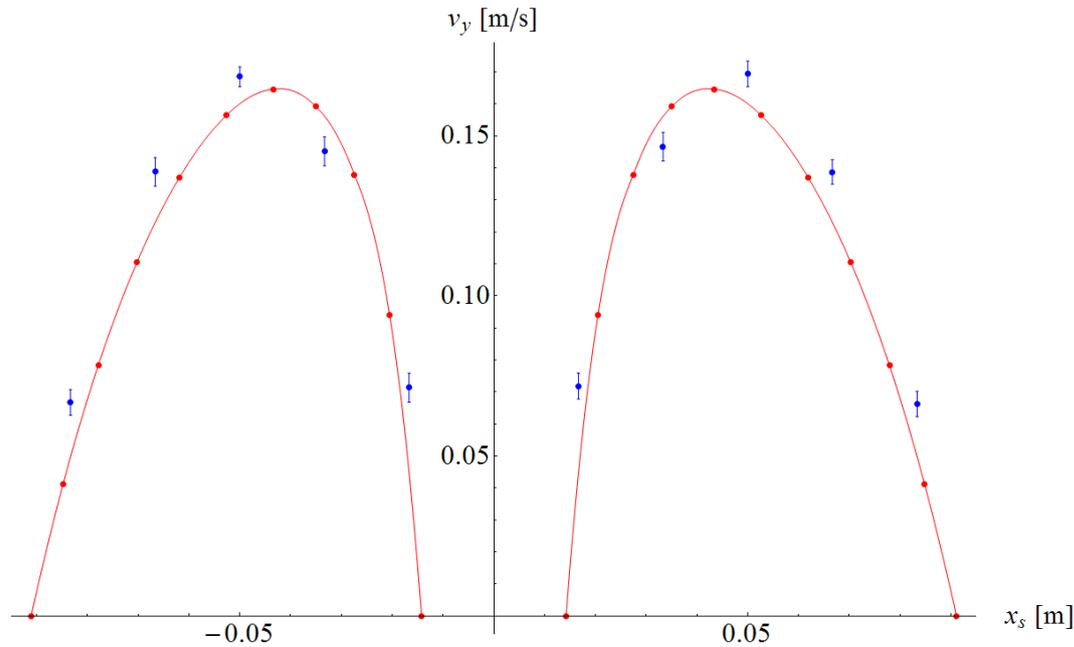
Even in the quasi-stationary state after the smooth acceleration phase of the piston – with a thorough preceding LIGGGHTS-only equilibration – the velocity field of the fluid in the dynamic HOTINT/LIGGGHTS SPH simulation exhibited significant fluctuations (cf. Figure 6.7). The direct comparison of the velocity profile with the result from the finite volume reference simulation (Figure 6.9) shows consistency within an error range of approximately 5 – 10% for the time-averaged sensor data.



**Figure 6.7.:** Fluid velocity in  $y$ -direction versus simulation time, measured at the point  $(-0.05 \text{ m}, 0 \text{ m})$  by the fourth sensor from the left, and spatially averaged over a circular domain of radius  $2h$  (with the SPH smoothing length  $h$ ). After the acceleration phase of the piston ( $t \leq 1 \text{ s}$ ), a quasi-stationary state is reached, with relative fluctuations in the range of 10%.



**Figure 6.8.:** Qualitative comparison of the velocity field in  $y$ -direction around the cylinder obtained from the coupled HOTINT/LIGGGHTS SPH simulation at  $t = 4.928 \text{ s}$  with in total 9048 SPH particles (left-hand side) and a standard finite volume simulation using the solver ANSYS Fluent 6.3, based on a 2D mesh with 1808 quadrilateral cells (right-hand side). More information on the latter is given in the text; for a quantitative comparison, cf. Figure 6.9.



**Figure 6.9.:** Profile of the fluid velocity in  $y$ -direction in the symmetry plane, i.e. on the  $x$ -axis (cf. Figure 6.6). The red line corresponds to the interpolated data points (also plotted red) obtained from a standard finite volume simulation using the solver ANSYS Fluent 6.3, based on a 2D mesh with 1808 quadrilateral cells (more information on that is given in the text), whereas the blue markers correspond to the time average over the time interval  $2 \dots 6$  s of the sensor data obtained from the coupled HOTINT/LIGGGHTS SPH simulation, with the respective error bars (cf. also Figure 6.7).

We can see an acceptable agreement of the results from the method developed in this work and the reference finite volume solution with deviations in the range of 5 – 10%, however, it should be noted, that this example is generally problematic – particularly in 2D (see text for details). Further improvements of the accuracy could be achieved by the calculation of the actual value of the velocity field at the sensor positions according to the SPH interpolation (cf. equation (3.12)) – which has not been implemented yet – instead of the simple averaging process over all particles within a distance of  $2h$  ( $h$ ... SPH smoothing length) to the respective sensor.

In general it can be stated that this specific example comes with several difficulties with the present method, in particular in two dimensions. As discussed above, the choice of appropriate values for the contact force scaling parameters and the corresponding interaction range is crucial and must also be considered in the original design of the geometry of the problem due to the possibly relatively large additional space between the fluid and the boundaries. In context with the latter, the problem of “stacking of particles” along the side walls and the resulting errors in the velocity field in vicinity to the respective boundaries is certainly more significant in 2D than it would be in 3D, since the freedom of movement of the SPH particles here is much more restricted.

Another thing worth mentioning concerns the equal size of all particles in the present implementation: Consider a set of hard spheres of perfectly equal size, and compare the dynamics those spheres exhibit when moving in a plane, for instance, with grains of sand of equal size in average. Due to the high symmetry of the spheres, in particular in cases of regular spatial distribution (e.g. according to some kind of lattice), some directions of motion are preferential, or, in other words, the “environment” for each particle is an anisotropic one. This anisotropy is realistic in case of hard spheres, but in the context of SPH particles – representing an isotropic fluid – clearly undesirable. Of course, the SPH particles are not exactly hard spheres, but nevertheless they have spherical symmetry and, with equal smoothing lengths, effectively equal size, which is the reason why the initial particle configuration was generated based on a regular grid with random displacements. Using a regular distribution at the beginning, the configuration may stay partly in that state even when the flow is fully developed, in particular in the region before the cylinder, bringing forward above, for the representation of a fluid unphysical anisotropic effects.

And last, but not least, it is also problematic to obtain a symmetric solution in this symmetric example, due to small fluctuations especially in the velocity field, which still remain to some extent even after a long equilibration procedure, and lead to (non-symmetric) perturbances. In such cases of very sensitive examples improvements of the equilibration process probably would be necessary, maybe by the use of smoothening techniques or the introduction of some kind of additional (artificial) dissipation.

All in all, with this example consistency of the method could be shown for the fluid side by comparison to a reference solution, even though several problems arised causing considerable drawbacks compared to standard methods (FVM). Actually, in this case there would be no need for a fully coupled flexible treatment – the problem at hand is, in fact, a classical example to be solved numerically by means of the finite volume method – but, as we have seen, it is always very informative from a developers perspective to investigate how a method performs in less compatible or suitable situations. Investigations of this kind not only serve as quantitative verification, but also provide insights which further improvements and extensions may be based on.

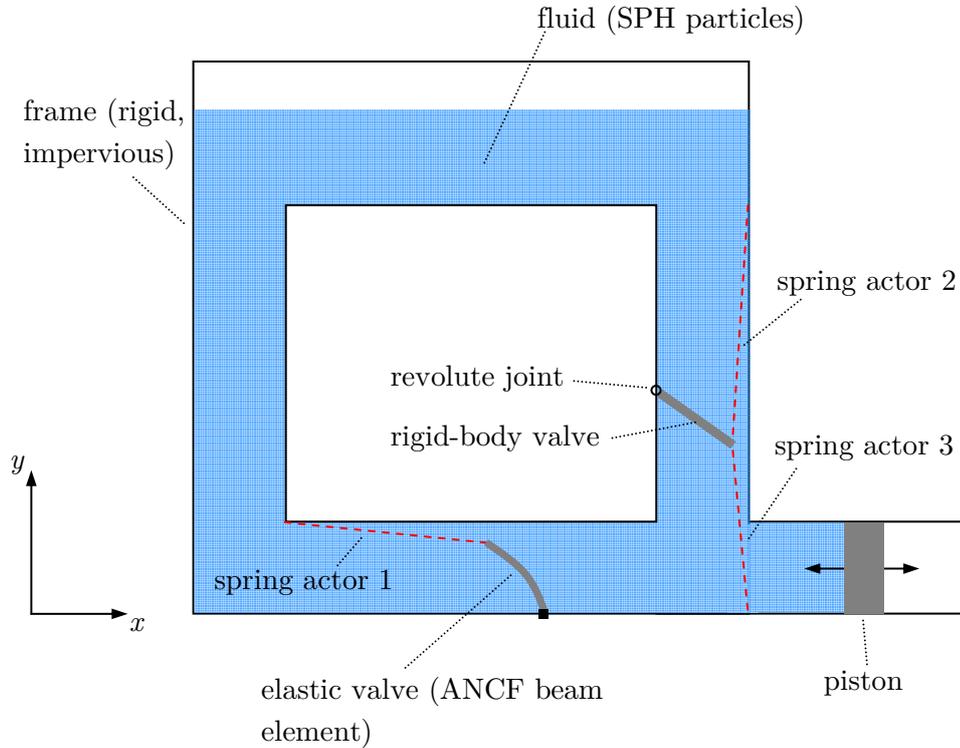
## 6.5. Pump-driven channel flow with two valves

Without any quantitative analysis, this test example just shall be an illustration of the potential of the developed approach in application to a slightly more complex multibody system consisting of

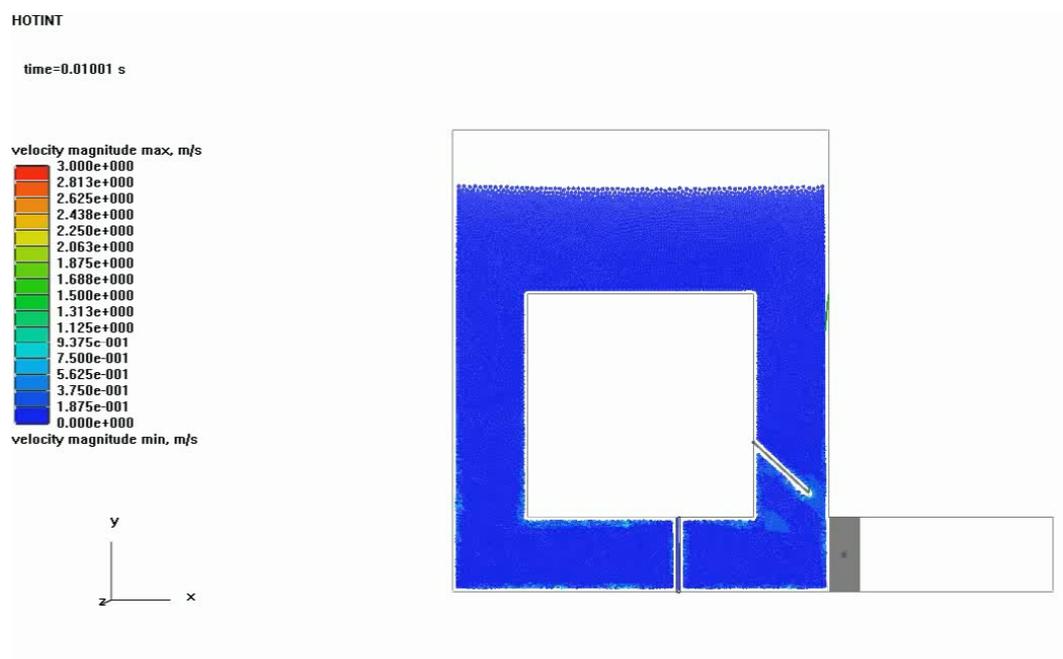
- a closed channel system guiding a circular flow, driven by a
- piston pump, and passively controlled by one
- rigid body valve connected with a non-linear spring to the ground, and one
- flexible valve modelled by a 2D ANCF beam element, also with an additional spring actor element.

Gravitational force was also included; the simulation was performed with 35000 SPH particles (fluid parameters: nominal density  $\rho_0 = 1000 \text{ kg/m}^3$ , kinematic viscosity  $\nu = 10^{-3} \text{ m}^2/\text{s}$ ). For a sketch of the system configuration with a short outline of its mechanics see Figure 6.10. The initial state after the equilibration procedure is shown in Figure 6.11, whereas the sequence of video frames in the Figures 6.12 and 6.13 should capture the dynamics of the system.

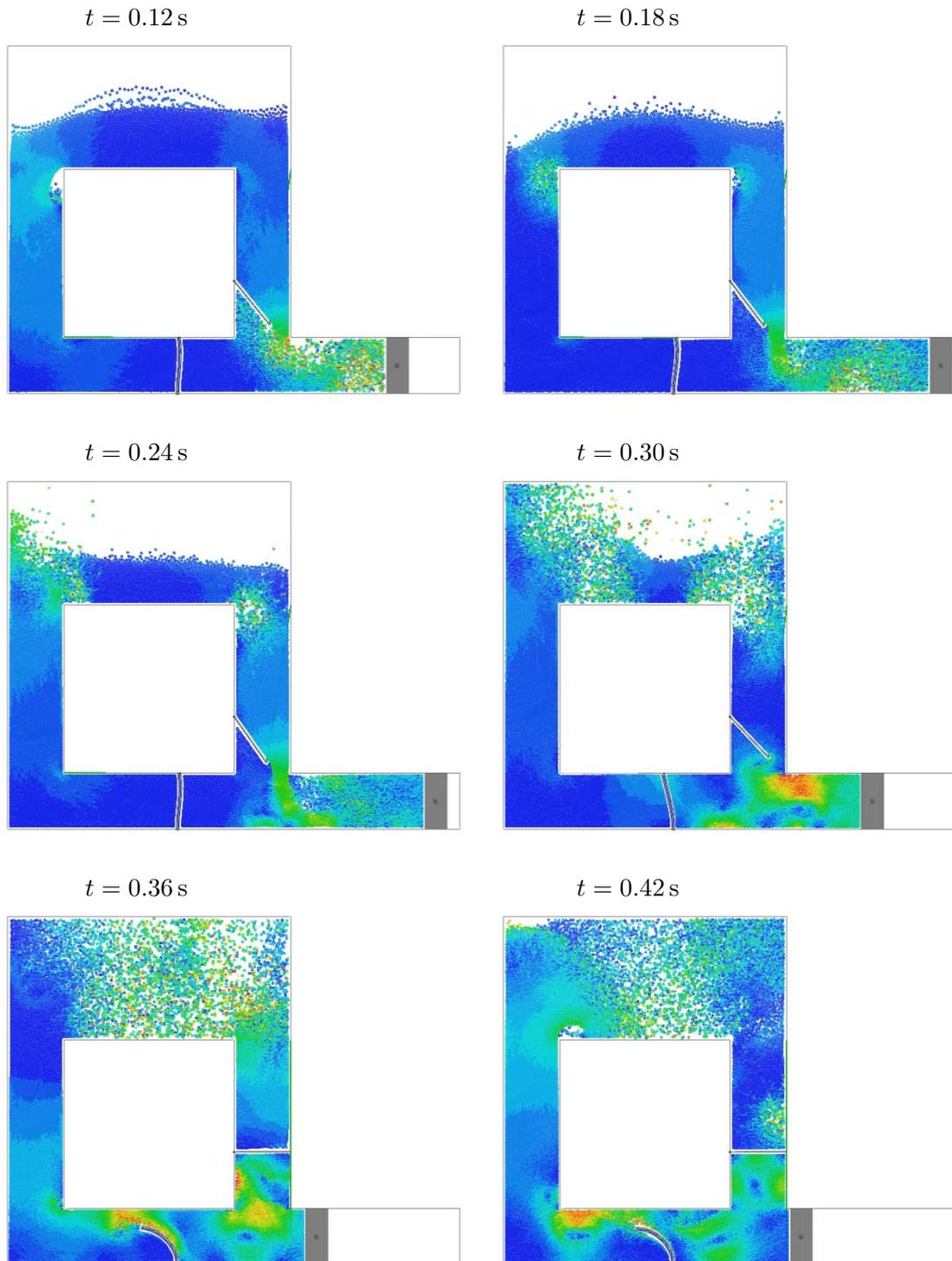
For some more advanced examples which were investigated and simulated based on a 3D implementation of the formalism – both outside the scope of this work – see the final remarks and outlook in the following, last chapter.



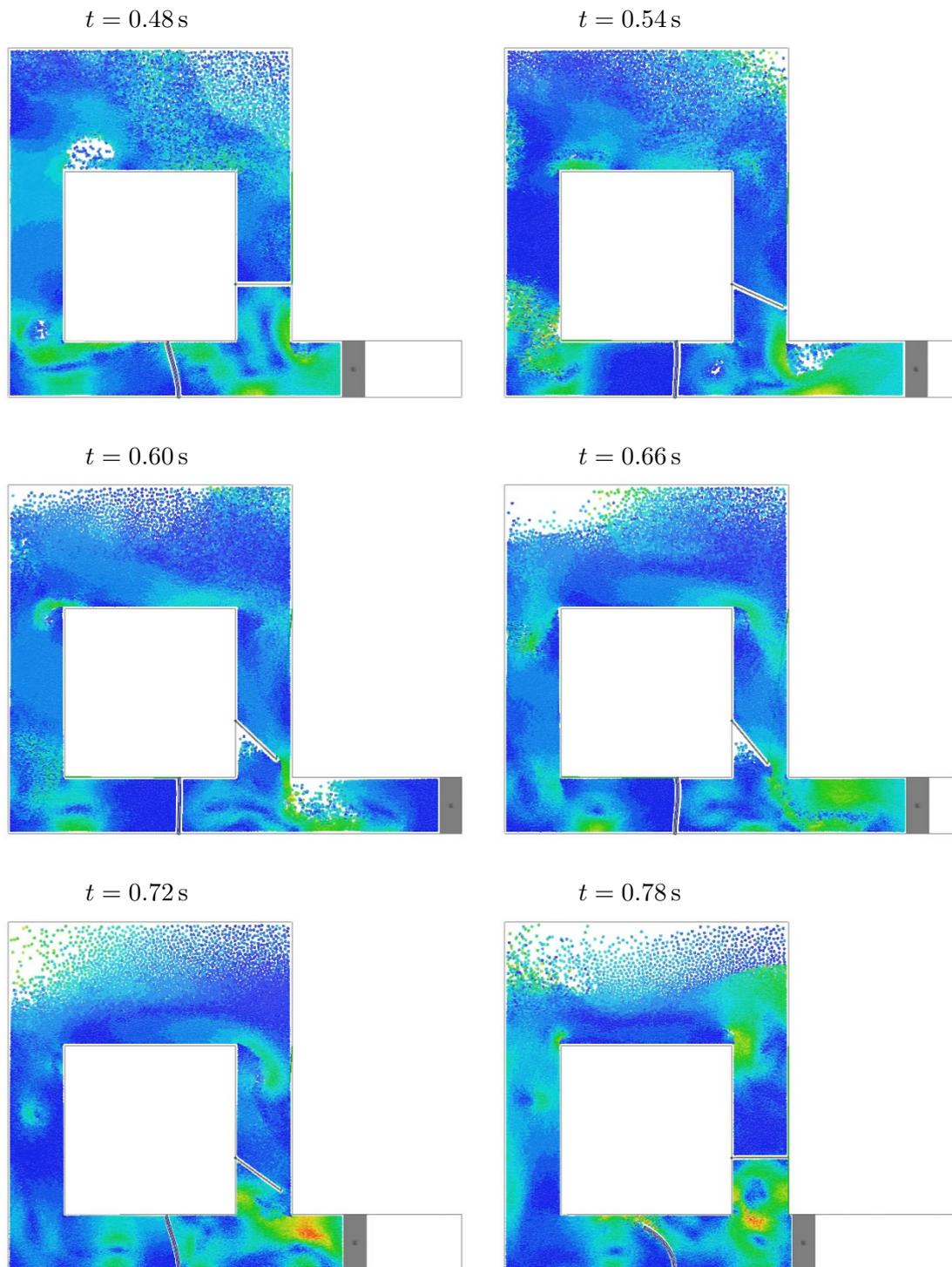
**Figure 6.10.:** Sketch of the system configuration: The piston is moving oscillatory in  $x$ -direction and pumping fluid in clockwise direction, where the rigid-body valve, connected to the frame (ground of the MBS) via a revolute joint, acts as an inlet valve, and the elastic valve, modelled by a 2D ANCF beam element, represents an outlet valve. The (non-linear) spring actor elements are used as passive control elements, with their equilibrium positions in the state where both valves are closed. Spring actor 1 is associated with a high tensile stiffness in  $+x$ -direction, thus ensures that no fluid can flow through the elastic valve from left to right, while spring actor 2 is exerts an additional force in  $-y$ -direction on the tip of the rigid body valve in case of compression, preventing any mass flux in the  $+y$ -direction in the compression phase (piston movement in  $-x$ -direction); the third actor element just supports the intake process of the fluid in the expansion phase (piston movement in  $+x$ -direction). Gravitational force is included ( $-y$ -direction).



**Figure 6.11.:** The state of the system at  $t = 0.01$  s after the equilibration procedure, which was performed with a partly opened rigid body valve in order to fill the whole geometry uniformly, including the piston chamber. The color of the fluid (SPH particles) corresponds to the absolute fluid velocity (cf. the legend at the left-hand side).



**Figure 6.12.:** Sequence of video frames of the simulation in equidistant time intervals of  $\Delta t = 0.06$  s, starting from  $t = 0.12$  s simulation time; for a legend of the fluid color corresponding to the absolute fluid velocity, cf. Figure 6.11. In the compression phase, the elastic outlet valve undergoes a large deformation due to the interaction with the fluid, corresponding to the pressure on the valves surface. The sequence of frames is continued in Figure 6.13.



**Figure 6.13.:** Sequence of video frames of the simulation in equidistant time intervals of  $\Delta t = 0.06\text{ s}$ , starting from  $t = 0.48\text{ s}$  simulation time; for a legend of the fluid color corresponding to the absolute fluid velocity, cf. Figure 6.11.

# 7. Outlook and conclusions

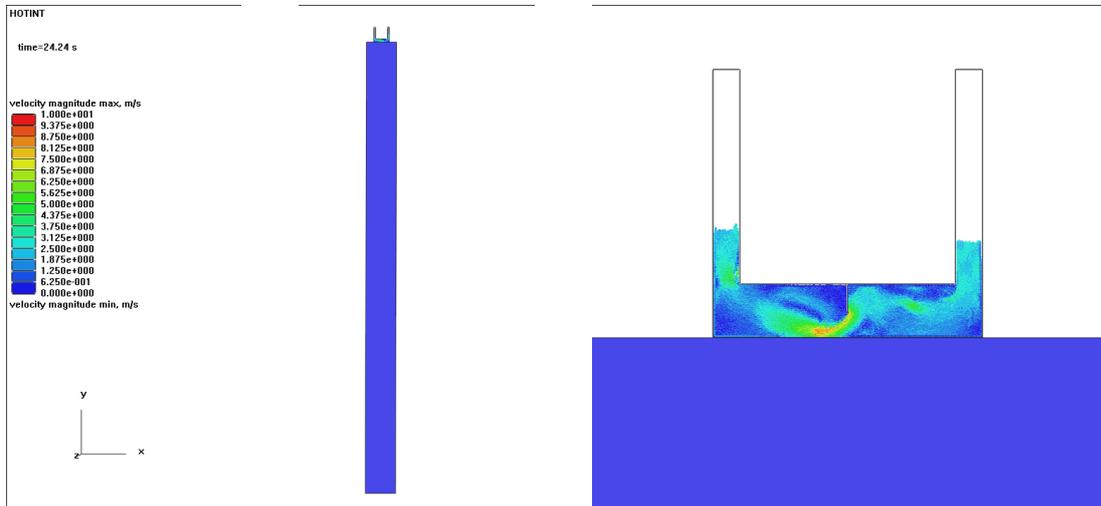
## 7.1. Outlook

Outside the scope of this master thesis, based on the discussed implementation in 2D, a full 3D implementation has been developed and still is being extended, refined, and tested on varying example problems (note: the C++ source code in the appendix also includes major parts of that additional implementation). The key points and challenges for the extension from two to three spatial dimensions lie, firstly, in the definition, creation, implementation and treatment of arbitrary complex 3-dimensional boundaries, including the surfaces of arbitrarily shaped, moving and deforming 3D objects – on the side of both HOTINT and LIGGGHTS – and, secondly, in data management and issues of optimization and efficiency, as well as visualization.

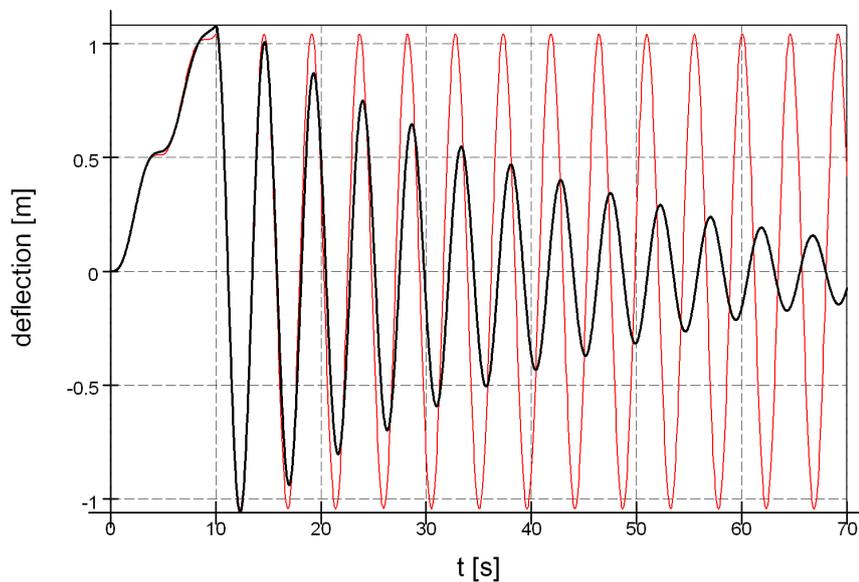
A short recap of the approach developed in this work – its benefits and problems – as well as a discussion on possible future extensions and improvements is given the next (and last) section. Concludingly, for illustration of the current state, simulation results of one advanced 2D and two 3D examples shall be presented here in short:

- 2D simulation of a tuned liquid column damper attached to a flexible structure modelled by an ANCF beam element. Some results are shown in the Figures 7.1 and 7.2.
- 3D simulation of a flexibly mounted axial pump with a rigid body 5-blade rotor geometry. Numerical data (sensor data) and a few snapshots are shown in the Figures 7.3 to 7.5.
- 3D simulation of fiber-fluid interaction with an array of thin, highly flexible fibers consisting of two large-deformation ANCF beam elements each. See the Figures 7.6 to 7.10 for simulation results.

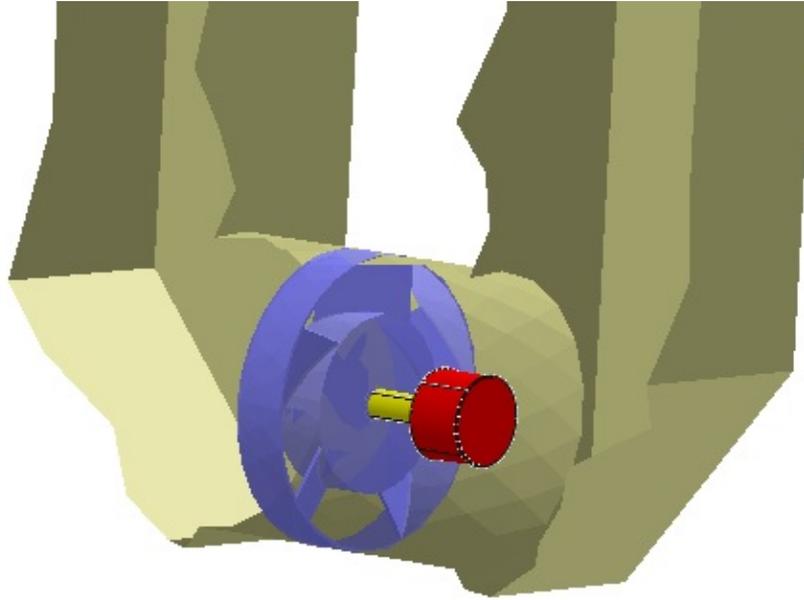
Note: In all three cases the gravitational force was included.



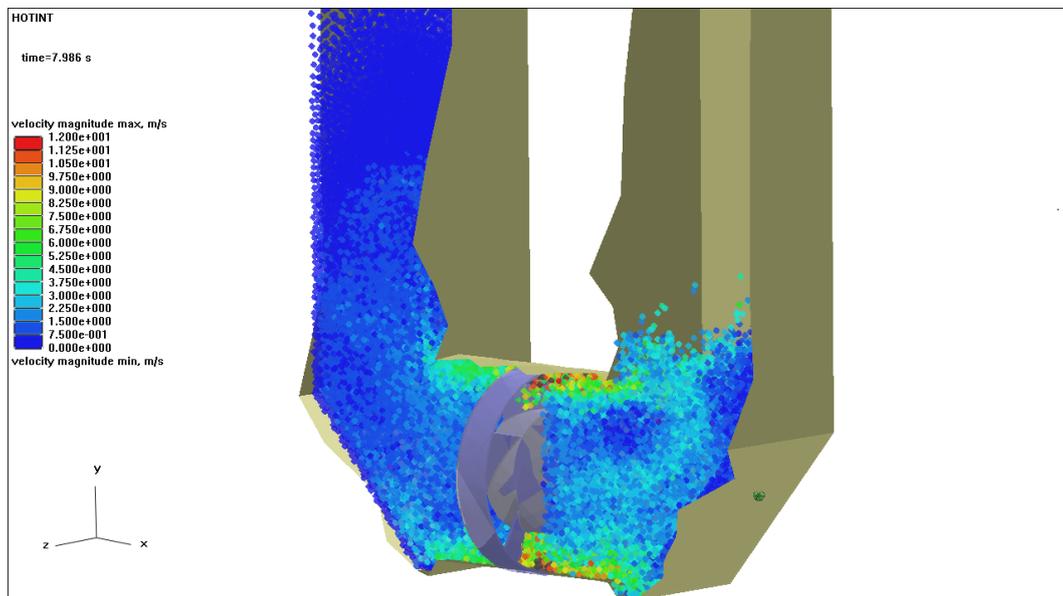
**Figure 7.1.:** A U-shaped tuned liquid column damper on the top of a building modelled by one 2D ANCF beam element (pictured in blue); the colors of the SPH particles correspond to the absolute fluid velocity.



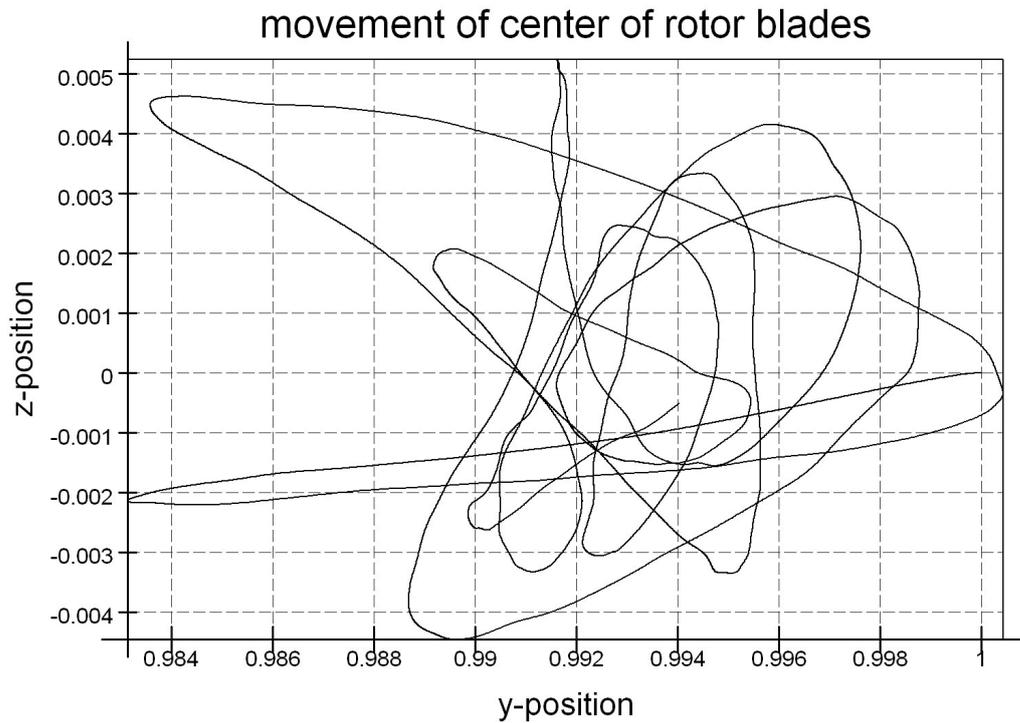
**Figure 7.2.:** Investigation of the damping behavior: The beam was deflected by a linear increasing initial force ( $0 \leq t \leq 10$ s), and then released. The red curve corresponds to the motion of the beam tip without fluid in the damper, the decaying black curve to the deflection with the filled tuned liquid column damper, versus simulation time.



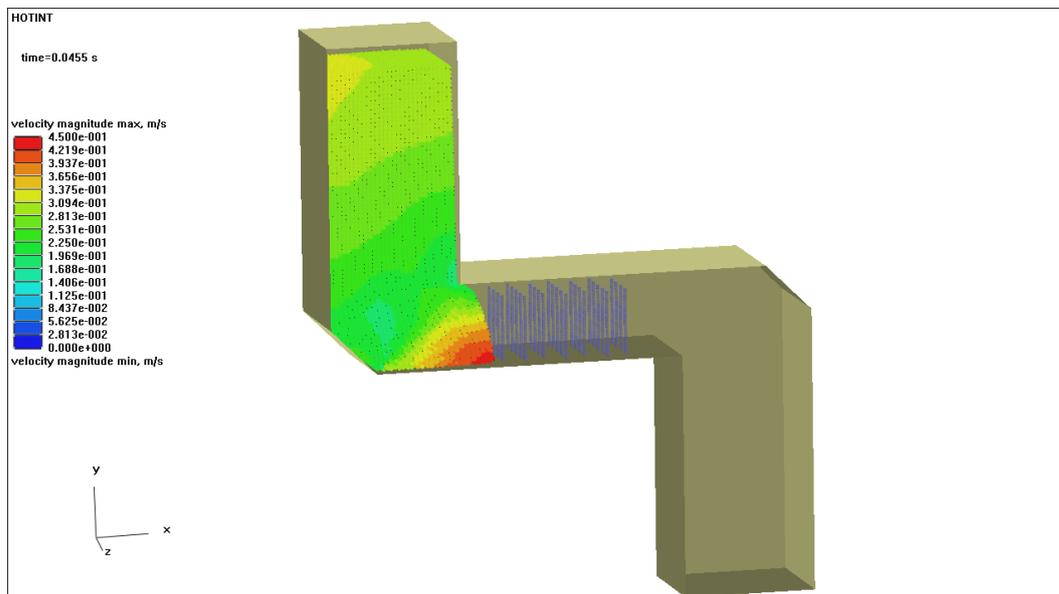
**Figure 7.3.:** Cut through the outer geometry without SPH particles, uncovering the flexibly-mounted rotor (blue) and the drive shaft/train (yellow/red, just for illustration).



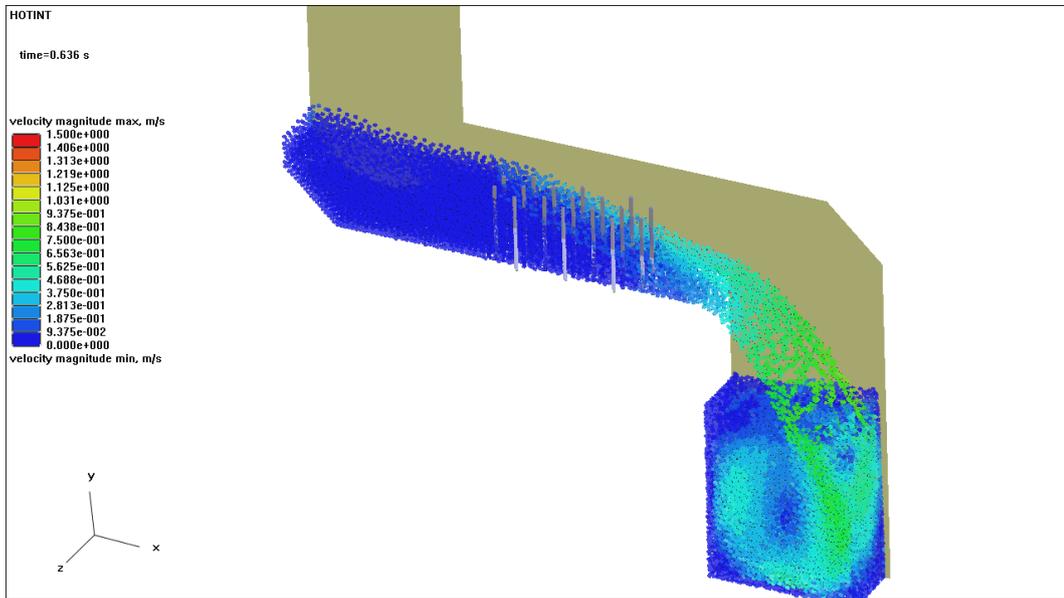
**Figure 7.4.:** Cut through the geometry during the pumping process, with an effective mass flux from right to left generated by a forced clockwise rotation of the rotor; the color corresponds to the absolute fluid velocity.



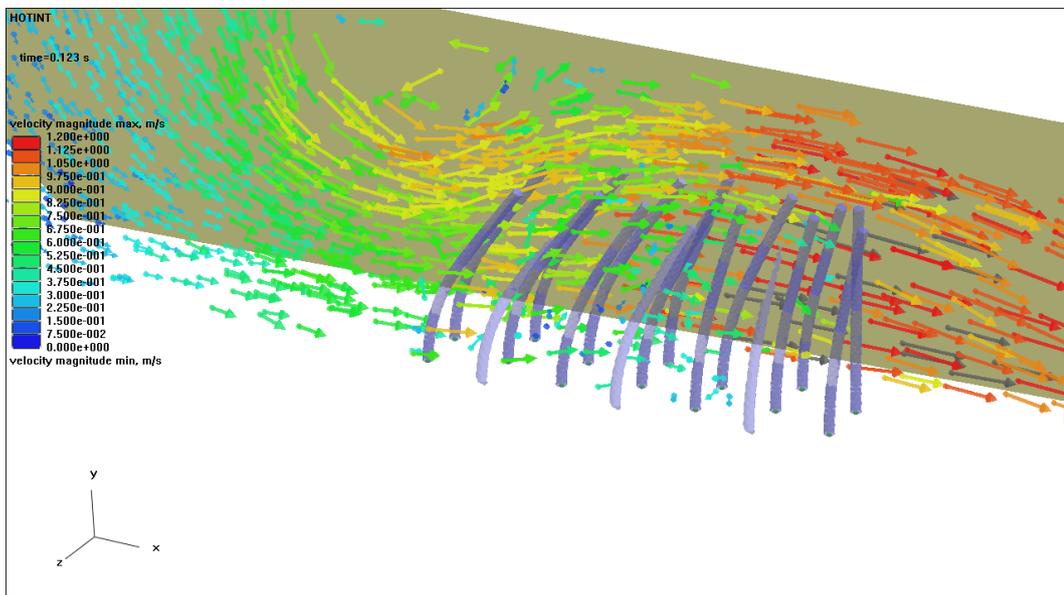
**Figure 7.5.:** Movement (trajectory) of the center of the rotor about its mounting point, i.e. its initial position  $\mathbf{r}_{center}(t = 0) = (0, 1, 0)$  in the  $yz$ -plane; the rotation axis as well as the main flow direction generated by the pump here is parallel to the  $x$ -axis.



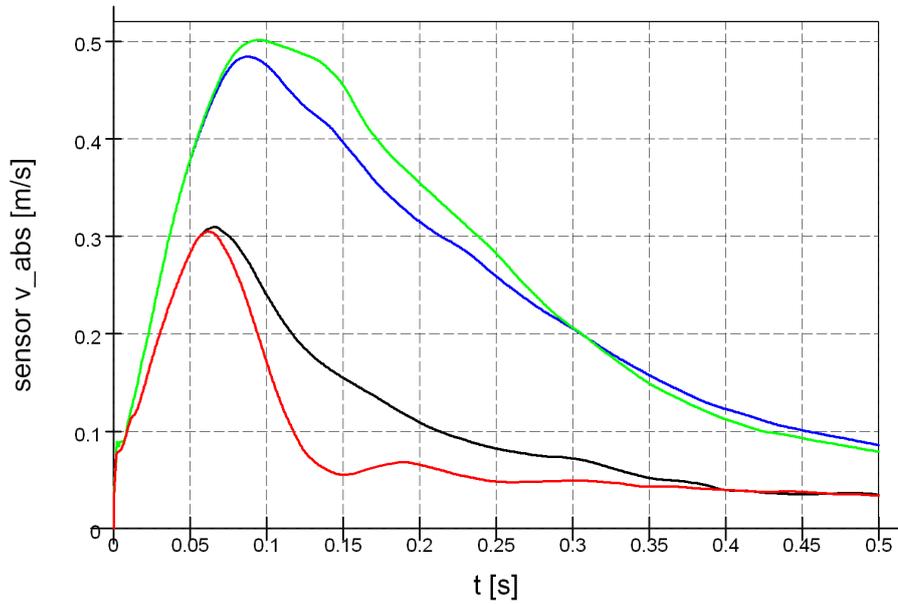
**Figure 7.6.:** Cut through the geometry shortly after the initial setting, with the fluid flowing from the upper left-hand side through the array of flexible fibers in the center (in blue, modelled by two ANCF beam elements per fiber) into the box on the lower right-hand side.



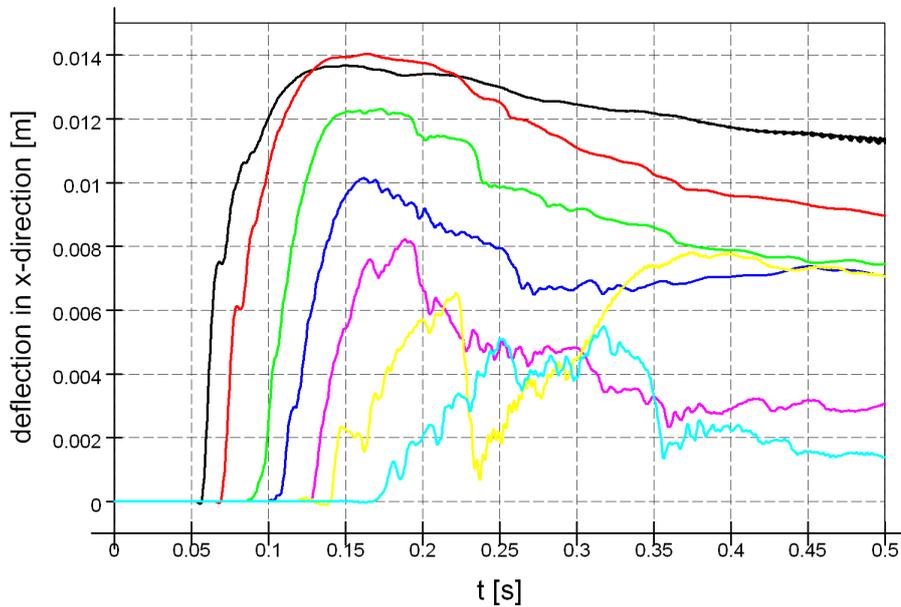
**Figure 7.7.:** Cut through the geometry with the fully developed flow, using a very high stiffness for the beam elements (almost rigid case).



**Figure 7.8.:** A closer look at one half of the fibers in interaction with the fluid, using highly flexible beam material; only a fraction of the SPH particles is shown with velocity vectors scaled and colored according to the absolute fluid velocity.



**Figure 7.9.:** Absolute fluid velocity measured with a sensor placed on a central position in front of the fibers versus simulation time, for different material parameters: Low (very high) viscosity  $\nu_{low}$  ( $\nu_{high}$ ), and low (very high) stiffness of the beam elements  $E_{low}$  ( $E_{high}$ ). Red curve:  $\nu_{high}$ ,  $E_{high}$ , black curve:  $\nu_{high}$ ,  $E_{low}$ , blue curve:  $\nu_{low}$ ,  $E_{high}$ , green curve:  $\nu_{low}$ ,  $E_{low}$ .



**Figure 7.10.:** Sensor data of the deflection of the fiber tips, for the first row of fibers parallel to the back wall, in  $x$ -direction (i.e. the flow direction) versus simulation time. The black line corresponds to the beam reached by the fluid first, then the others follow subsequently with increasing  $x$ -coordinate of their mounting position.

## 7.2. Conclusions

Starting from, and with focus on the mechanical side, the objective of this work was the development of a method to include a general viscous fluid in simulations of flexible multibody system dynamics. At that, the structural side – the wide range of possible configurations and problem situations in MBD – determined the requirements of the fluid side, demanding the following key capabilities:

- Handling of arbitrary geometries, or, in other words, applicability to any MBS, regardless of the state of motion and/or deformation of the components of the MBS,
- performance and efficiency, and
- stability as well as consistency of the fluid simulation.

Considering the first requirement in particular, the use of a meshfree model for the representation and simulation of the fluid is the most promising choice, as already mentioned in the introduction. With the direct coupling of flexible multibody dynamics and smoothed particle hydrodynamics, an unconventional approach to the problem of fluid-structure interaction was developed, the underlying theory as well as the implementation of which have been discussed in the previous chapters, concluded by the simulation and investigation of several 2D example problems. To that end, with HOTINT on the multibody side and LIGGGHTS for the fluid simulation, two powerful tools featuring extensive capabilities and functionality on their respective fields have been coupled in an efficient and flexible way based on the introduction of a server-client relation and a definition of an interface.

All in all, the main advantages of this approach are its applicability to arbitrary multibody systems consisting of arbitrarily moving and/or deforming bodies, and the very accurate modelling of the structural components. Moreover, high performance is given due to the advanced and efficient methods of MBD implemented in HOTINT on the one hand, and to the fast algorithms used in many-particle simulations and the field of molecular dynamics, as well as the possibility of massively-parallel computation of LIGGGHTS on the other hand. For sufficiently small time steps the coupled simulation is stable and produces consistent, accurate results, where the latter, however, can be difficult on the fluid side, even in case of certain simple problems which can be solved easily and very accurately by means of classical methods, for instance, the finite volume method (cf. the simulation of the laminar flow around a cylinder in Section 6.4). Hence, the main field of application for this fully coupled MBD/SPH approach are problems with a complex structural side and highly flexible components undergoing large motion and/or deformation, possibly including free surface flows.

Of course, it should be noted that there are still many difficulties, which have – with respect to the present coupling approach – either arisen during test runs of certain simulations, or are well-known anyways, particularly concerning the method of SPH itself. Amongst those problems are

- an accurate computation of the fluid properties in the vicinity of boundaries (especially in 2D), along with
- an appropriate choice for the (scaling) parameters of the fluid-structure contact forces (in particular, for the repulsive force) or the artificial viscosity terms,
- the process of equilibration,
- significant fluctuations/oscillations in certain fluid field quantities, especially in case of a (quasi-)stationary state,
- the implementation of other fluid boundary conditions (e.g. a Dirichlet condition for the velocity field at a velocity inlet), and
- stability issues, in particular with respect to the LIGGGHTS-sided non-adaptive, explicit time integration without convergence or consistency tests, which not only is problematic for the fluid simulation itself, but can also affect the mechanical side (e.g. a too large time step size for the fluid simulation can result in artificial, mechanical instabilities of interacting flexible structures).

Note that several of above points were also encountered in the basic “laminar flow around a cylinder” example discussed in Section 6.4. However, none of these problems are a priori insoluble, and after all, there is a lot of optimization potential. The following list outlines some possibilities for a further increase of efficiency and performance:

- A full distributed-memory implementation of the handling of the discretized surfaces. Currently, all processes operate on identical copies of the interface data set; clearly, it would be ideal to use a similar, efficient strategy as in the case of the SPH particles by considering and managing only that part of the boundaries which lies within the respective subdomain of each process; also, a shared-memory-like approach based on only one mutually accessed interface data set could be another option.
- The implementation of a more efficient surface definition. Currently – even though the interface itself is node-based – all surface elements, also connected ones with mutual vertices, are defined independently, i.e. the full number of vertices of each element is read in individually, resulting in redundant information (which is, roughly, by a factor of 2 in 2D and 3 in 3D larger than the essential data size).
- The use of different time step sizes on both sides, with corresponding interpolation schemes for the forces on the structural side, and, importantly, the positions and velocities of the boundary elements for the fluid simulation. In the current implementation, the time step size is controlled by HOTINT, but must be limited to very small values (typically in the range of  $10^{-5}$ - $10^{-6}$  s) due to the explicit integration routines LIGGGHTS-sided, which is by a factor of 10-1000 smaller than necessary for the integration of the multibody system. This would increase efficiency, and eliminate the possibility of a TCP/IP bottleneck, since then all data – not only the SPH

information – would only be exchanged in intervals of 100s or even 1000s of time steps.

Concerning the stability, some options for improvements would be

- an adaptive step-size control in LIGGGHTS, and/or
- communication and synchronization of information on the current state of convergence, or
- the implementation of other time integration schemes for the fluid simulation.

Additional enhancements could be possible by the use of an enhanced equilibration procedure, of other strategies for the numerical integration in the computation of the contact forces and the corresponding mesh refinement, modified smoothing kernels, and certainly by advanced implementations of the SPH formalism – with respect to both accuracy and stability on the fluid side. It should be noted that the SPH implementation in LIGGGHTS used for this work (retrieved in October, 2011) is a rather basic one, based on one type of identical particles with one smoothing kernel and constant parameters only. For more information about advanced methods, aiming for higher stability, smoother results, an efficient handling of multi-scale problems requiring different spatial resolutions (i.e. different “sizes” of particles – given by the mass and average distance of the particles, hence also connected to the smoothing lengths of the corresponding kernel functions), or more accurate results in boundary regions, refer to the respective literature and the current research in the field of SPH (e.g. the proceedings of the SPHERIC conference 2011 [42], and the references therein).

## **Declaration of authenticity**

I hereby declare under oath that the submitted Master's thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited. The submitted document here present is identical to the electronically submitted text document.

## Bibliography

- [1] Galdi, P. G. / Rannacher, R. (eds.) *Fundamental Trends in Fluid-Structure Interaction. Contemporary Challenges in Mathematical Fluid Dynamics and Its Applications*, Vol. 1. Singapore, 2010.
- [2] Bungartz, H. J. / Schäfer, M. (eds.) *Fluid-Structure Interaction: Modelling, Simulation, Optimisation*. Berlin Heidelberg New York, 2006.
- [3] Bungartz, H. J. / Mehl, M. / Schäfer, M. (eds.) *Fluid-Structure Interaction II: Modelling, Simulation, Optimization*. Heidelberg London Dordrecht New York, 2010.
- [4] Shabana, A. A. *Dynamics of Multibody Systems*. Third edition, New York, 2005.
- [5] Gerstmayr, J. / Sugiyama, H. / Mikkola, A. *An Overview on the Developments of the Absolute Nodal Coordinate Formulation*. In: Proceedings of the Second Joint International Conference on Multibody System Dynamics, Stuttgart, 2012.
- [6] Nachbagauer, K. / Pechstein, A.S. / Irschik, H. / Gerstmayr, J. *A new locking-free formulation for planar, shear deformable, linear and quadratic beam finite elements based on the absolute nodal coordinate formulation*. Journal of Multibody System Dynamics, 26 (3), pp. 245-263, DOI: 10.1007/s11044-011-9249-8, Open Access, 2011.
- [7] Stangl, M. / Gerstmayr, J. / Irschik, H. *A Large Deformation Planar Finite Element for Pipes Conveying Fluid Based on the Absolute Nodal Coordinate Formulation*. ASME Journal of Computational and Nonlinear Dynamics, Vol. 4 (3), pp. 031009-1 - 031009-8, 2009.
- [8] Gerstmayr, J. *Computer Methods in Mechanics: Lecture Notes*. Linz, 2011.
- [9] Gerstmayr, J. *Higher Computer Methods in Mechanics: Lecture Notes*. Linz, 2009.
- [10] Bronstein, I. N. et. al. *Taschenbuch der Mathematik*. 6. Auflage, Frankfurt am Main, 2006.
- [11] Ascher, U. M. / Petzold, L. R. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia, 1998.
- [12] Hairer, E. / Wanner, G. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Second edition, Berlin Heidelberg, 1996.
- [13] Liu, G. R. / Liu, M. B. *Smoothed Particle Hydrodynamics: A meshfree particle method*. Singapore, 2003.

- [14] Dalrymple, R. A. *Particle Methods and Waves, with Emphasis on SPH: Additional lecture material*. Baltimore, 2007.
- [15] Liu, G. R. *Mesh Free Methods: Moving beyond the Finite Element Method*. Boca Raton London New York Washington D.C., 2003.
- [16] Monaghan, J. J. *Simulating Free Surface Flows with SPH*. Journal of Computational Physics, Vol. 110, pp. 399-406, 1994.
- [17] Monaghan, J. J. *Smoothed Particle Hydrodynamics*. Annual Review of Astronomical and Astrophysics, Vol. 30, pp. 543-574, 1992.
- [18] Monaghan, J. J. / Gingold, R. A. *Shock simulation by the particle method of SPH*. Journal of Computational Physics, Vol. 52, pp. 374-381, 1983.
- [19] Greiner, W. / Stock, H. *Hydrodynamik*. 4. Auflage, Frankfurt am Main, 1991.
- [20] Plimpton, S. J. *Fast Parallel Algorithms for Short-Range Molecular Dynamics*. Journal of Computational Physics, Vol. 117, pp. 1-19, 1995.
- [21] Plimpton, S. J. / Pollock, R. / Stevens, M. *Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations*. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, 1997.
- [22] *CFDEM - Open Source CFD, DEM and CFD - DEM*. <http://www.liggghts.com>. Accessed 4/2012.
- [23] *LIGGGHTS Documentation*. Available via <http://www.liggghts.com>. Accessed 1/2012.
- [24] Sandia National Laboratories *LAMMPS Users Manual: Large-scale Atomic/Molecular Massively Parallel Simulator*. 2003. Available at <http://lammps.sandia.gov>.
- [25] Zulehner, W. *Numerische Mathematik: Eine Einführung anhand von Differentialgleichungsproblemen*. Band 2: *Instationäre Probleme*. Basel, 2011.
- [26] Frenkel, D. / Smit, B. *Understanding Molecular Simulation: From Algorithms to Applications*. San Diego San Francisco New York Boston London Sydney Tokyo, 2002.
- [27] Griebel, M. / Knapek, S. / Zumbusch, G. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Berlin Heidelberg, 2007.
- [28] Bathe, K. J. *Finite Element Procedures*. New Jersey, 1996.
- [29] Zienkiewicz, O. C. / Taylor, R. L. *The Finite Element Method*. Volume 1: *The Basis*. Fifth edition, Oxford, 2000.
- [30] Zienkiewicz, O. C. / Taylor, R. L. *The Finite Element Method*. Volume 2: *Solid Mechanics*. Fifth edition, Oxford, 2000.
- [31] Zienkiewicz, O. C. / Taylor, R. L. *The Finite Element Method*. Volume 3: *Fluid Dynamics*. Fifth edition, Oxford, 2000.

- 
- [32] Segerlind, L. J. *Applied Finite Element Analysis*. Second edition, New York Chichester Brisbane Toronto Singapore, 1984.
- [33] Greiner, W. *Klassische Mechanik II: Teilchensysteme, Lagrange-Hamiltonsche Dynamik, Nichtlineare Phänomene*. 7. Auflage, Frankfurt am Main, 2003.
- [34] Gerstmayr, J. *HOTINT Manual*. Linz, 2011.
- [35] Gerstmayr, J. / Stangl, M. *High-Order Implicit Runge-Kutta Methods for Discontinuous Multibody Systems*. In: Proceedings of the Summerschool on Actual Problems in Mechanics, Editor: D.A. Indeitsev, pp. 162-169, St. Petersburg, 2004.
- [36] Gerstmayr, J. / Irschik, H. *On the correct representation of bending and axial deformation in the absolute nodal coordinate formulation with an elastic line approach*. Journal of Sound and Vibration, Vol. 318, pp. 461-487, 2008.
- [37] George, A. / Liu, J. W. H. *The evolution of the minimum degree ordering algorithm*. SIAM Review, Vol. 31(1), pp. 1-19, 1989.
- [38] Amestoy, P. R. / Davis, T. A. / Duff, I. S. *An Approximate Minimum Degree Ordering Algorithm*. SIAM Journal on Matrix Analysis and Applications, Vol. 17, pp. 886-905, 1996.
- [39] Hanke-Bourgeois, M. *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*. 2. Auflage, Wiesbaden, 2006.
- [40] Gerstmayr, J. *HOTINT - A C++ Environment for the Simulation of Multibody Dynamics Systems and Finite Elements*. In: Multibody Dynamics 2009 ECCOMAS Thematic Conference, Editors: Arczewski, K. / Frączek, J. / Wojtyra, M., Warsaw, 2009.
- [41] González, L. M. et. al. *On the non-slip boundary condition enforcement in SPH methods*. In: 6-th International SPHERIC Workshop, Editors: Rung, T. / Ulrich, C., pp. 283-290, first edition, Hamburg, 2011.
- [42] Rung, T. / Ulrich, C. (eds.) *6-th International SPHERIC Workshop*. First edition, Hamburg, 2011.
- [43] Müller, M. et. al. *Interaction of fluids with deformable solids*. Computer Animation and Virtual Worlds, Vol. 15, pp. 159-171, 2004.
- [44] Weisstein, E. W. *Legendre-Gauss Quadrature*. From MathWorld – A Wolfram Web Resource. <http://mathworld.wolfram.com/Legendre-GaussQuadrature.html>. Accessed 4/2012.
- [45] Breymann, U. *C++: Einführung und professionelle Programmierung*. 8., erweiterte Auflage, München Wien, 2005.
- [46] *The C++ Resources Network*. <http://www.cplusplus.com/>. Retrieved 4/2012.
- [47] *Visual C++ Reference*. [http://msdn.microsoft.com/en-US/library/ty9hx077\(v=vs.80\).aspx](http://msdn.microsoft.com/en-US/library/ty9hx077(v=vs.80).aspx). Retrieved 4/2012.

- [48] *MSDN Academic Alliance*. [http://www.microsoft.com/austria/education/msdn\\_academic\\_alliance.aspx](http://www.microsoft.com/austria/education/msdn_academic_alliance.aspx). Retrieved 4/2012.
- [49] *Virtual Box*. <https://www.virtualbox.org/>. Retrieved 4/2012.
- [50] Gropp, W. / Lusk, E. / Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing-Interface*. Second edition, Cambridge London, 1999.
- [51] Gropp, W. / Lusk, E. / Thakur, R. *Using MPI-2: Advanced Features of the Message-Passing-Interface*. Cambridge London, 1999.
- [52] IEEE Computer Society *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008. New York, 2008.
- [53] Brian Hall *Beej's Guide to Network Programming: Using Internet Sockets*. 2009. Available at <http://beej.us/guide/bgnet/>.
- [54] *Winsock Tutorial von c-worker.ch*. [http://www.c-worker.ch/tuts/wstut\\_op.php](http://www.c-worker.ch/tuts/wstut_op.php). Accessed 10/2011.
- [55] Sumer, B. M. / Fredsøe, J. (eds.) *HYDRODYNAMICS AROUND CYLINDRICAL STRUCTURES*. Advanced Series on Ocean Engineering, Vol. 26. Revised edition, World Scientific Publishing, 2006.

## A. C++ source code

### A.1. interface\_baseclass

```
1  #ifndef INTERFACEBASECLASS
2  #define INTERFACEBASECLASS
3
4  class interface_baseclass{
5
6  protected:
7
8      //all counting starts from 0
9      double* _r; // global positions of boundary particles; coord j of particle i
           is given by r(i,j); set coord to d by setr(i,j,d)
10     double* _v; // global velocities of boundary particles; component j of
           particle velocity i is given by v(i,j); set vel to d by setv(i,j,d)
11     double* _f; //forces on boundary particles; f(i,j), setf(i,j,double)
12     unsigned int* _el; // el(m,j) contains number of point j (j=0..1 2D, 0...2 3
           D) of surface element m
13     //(line segment in 2D, triangle in 3D); setel(m,j,val);
14     double* _rSPH; //global positions of SPH particles; rSPH(i,j), setrSPH(i,j,
           double);
15     double* _vSPH; //global velocities of SPH particles; vSPH(i,j), setvSPH(i,j,
           double);
16     double* _rohSPH; //densities; rohSPH(i) returns density of particle i
17     double dt; //current size of timestep
18
19     int refinement_option; //for 3D: 0...no refinement, 1... recursive
           refinement based on original triangles in every time step, 2... pre-
           refined mesh, no additional refinement in time-stepping, 3... as 1, but
           based on pre-refined mesh
20     //default is 1; 2D is always 1
21     double dr; //refinement depth / resolution - mesh is refined in 2D (3D)
           until any line element (edge of a triangle) is shorter than dr
22
23 public:
24
25     unsigned int n; //number boundary points
```

```
26  unsigned int nstat; //number of boundary points which are assumed to be
    static points (boundaries fixed to ground)
27  //points with numers 0...(n-nstat-1) are treated as regular dynamic points,
    points with numbers n-nstat...n-1 are treated as static points, i.e.
    corresponding r and v is only exchanged in initialization; by default,
    nstat=0 (i.e., all points receive full data transfer)
28
29  unsigned int nSPH; //number of SPH particles
30  unsigned short dim; // 2 for 2D, 3 for 3D; used for el array, coordinates
    always 3D (in case of 2D: 0 z-component)
31  unsigned int nel; //number of surface elements
32
33  virtual void getrv(){}; //receive r and v (the dynamic part, cf. nstat)
34  virtual void sendrv(){}; //send r and v (the dynamic part, cf. nstat)
35  virtual void getrvfull(){}; //recv r and v (dynamic and static)
36  virtual void sendrvfull(){}; //send r and v (dynamic and static)
37  virtual void getrSPH(){}; //receive rSPH
38  virtual void getvSPH(){}; // receive vSPH
39  virtual void sendrSPH(){}; //send rSPH
40  virtual void sendvSPH(){}; //send vSPH
41  virtual void sendforce(){}; // send f (only for non-static elements)
42  virtual void getforce(){}; //receive f (only for non-static elements)
43  virtual void getrohSPH(){}; //receive rohSPH
44  virtual void sendrohSPH(){}; //send rohSPH
45
46  virtual ~interface_baseclass(){};
47
48  inline double r(int i, int j) {return _r[i*3+j];} //return coordinates /
    velocity components / force components
49  inline double v(int i, int j) {return _v[i*3+j];}
50  inline double f(int i, int j) {return _f[i*3+j];}
51  inline double* rp(int i) {return _r+3*i;} //return pointers to i-th
    coordinate/vel/force vector
52  inline double* vp(int i) {return _v+3*i;}
53  inline double* fp(int i) {return _f+3*i;}
54  inline double rSPH(int i, int j) {return _rSPH[i*3+j];}
55  inline double vSPH(int i, int j) {return _vSPH[i*3+j];}
56  inline unsigned int el(int i, int j) {return _el[i*dim+j];}
57  inline unsigned int* elp(int i, int j) {return _el+i*dim+j;} //returns
    pointer to element (i,j)
58  inline void setr(int i, int j, double d) {_r[i*3+j]=d;}
59  inline void incr(int i, int j, double d) {_r[i*3+j]+=d;}
60  inline void setv(int i, int j, double d) {_v[i*3+j]=d;}
61  inline void incv(int i, int j, double d) {_v[i*3+j]+=d;}
62  inline void setf(int i, int j, double d) {_f[i*3+j]=d;}
```

```

63  inline void incf(int i, int j, double d) {_f[i*3+j]+=d;}
64  inline void setrSPH(int i, int j, double d) {_rSPH[i*3+j]=d;}
65  inline void setvSPH(int i, int j, double d) {_vSPH[i*3+j]=d;}
66  inline void setel(int i, int j, unsigned int x) {_e1[dim*i+j]=x;}
67  inline void zerof() {for(int i=0; i<3*n; ++i) _f[i]=0.0;}
68  inline void zeror() {for(int i=0; i<3*n; ++i) _r[i]=0.0;}
69  inline void zerov() {for(int i=0; i<3*n; ++i) _v[i]=0.0;}
70  inline void zerorSPH() {for(int i=0; i<3*nSPH; ++i) _rSPH[i]=0.0;}
71  inline void zerovSPH() {for(int i=0; i<3*nSPH; ++i) _vSPH[i]=0.0;}
72  inline double rohSPH(int i) {return _rohSPH[i];}
73  inline void setrohSPH(int i, double d) {_rohSPH[i]=d;}
74  inline void zerorohSPH() {for(int i=0; i<nSPH; ++i) _rohSPH[i]=0.0;}
75  inline void set_refinement_option(int r){refinement_option=r;}
76  inline int get_refinement_option(){return refinement_option;}
77  inline void set_dr(double r){dr=r;}
78  inline double get_dr(){return dr;}
79
80  };
81
82  #endif

```

## A.2. dn.h

```

1  #ifndef DOUBLE_TO_NETWORK
2  #define DOUBLE_TO_NETWORK
3
4  //Berechnung der Parameter einer double d in der Darstellung  $m \cdot 2^{\text{exp}}$  mit  $0.5 < |m| < 1$ , sign in m schon enthalten
5  //und exp eine ganze Zahl;
6  //|m| wird als 2*8 Dezimalstellen in 2 unsigned ints (4 Bytes) a1 (digits 1...8) und int a2 (digits 9...16) gespeichert;
7  //dafür (ganze Zahl  $\leq 10^9$ ) sind 30 bits notwendig;
8  //|exp| kommt in ein unsigned short (2 Bytes); das VZ von m kommt aufs msb von a1 (0 für +, 1 für -), das von exp aufs msb von a2
9  // Umwandlung und Rückumwandlung über htods/l bzw. ntohs/l, und entsprechendes & für die VZ
10
11 //host to network double
12 //d ist die umzuwandelnde double, a1,a2,exp beinhalten dann die Information im Netzwerkybyteorder
13 void htodn(double d, unsigned int & a1, unsigned int & a2, unsigned short & exp);
14 //Rückumwandlung
15 //network to host double

```

```
16 //gibt die umgewandelte double zurück, a1,a2,exp beinhalten die Darstellung in
    Netzwerkbyteorder
17 double ntohd(unsigned int a1, unsigned int a2, unsigned short exp);
18
19 //gleiche Funktionalität wie zuvor, nur wird in einen char-Array reservierten
    Speicher geschrieben;
20 //Möglichkeit zur sequentiellen Verarbeitung von doubles (via double Arrays)
    bzw 2-fach indizierten double Arrays double**
21 //char muss Dimension 10*n haben (da jede Double umgewandelt 10 Bytes
    entspricht); n...Anzahl der umzuwandelnden doubles = Länge des arrays d*;
    Ergebnis sequentiell in x
22 void htond(char* x, double* d, int n);
23 void ntohd(char* x, double* d, int n);
24
25 //Umwandeln und Austauschen eines n x k double Arrays
26 //char muss Dimension 10*n*k haben (da jede Double umgewandelt 10 Bytes
    entspricht);
27 //double ** ist ein Array mit Dimensionen n x k; Daten sequentiell in x;
28 void htond(char* x, double** d, int n, int k);
29 void ntohd(char* x, double** d, int n, int k);
30
31 //Umwandeln und Senden eines 1-d double Arrays der Länge n*k in ein n x k
    double Array
32 //char muss Dimension 10*n*k haben (da jede Double umgewandelt 10 Bytes
    entspricht);
33 //double * ist ein Array mit Länge n*k; Daten sequentiell in x (hier
    zurückwandelbar mittels ntohd(char*, double**, int n, int k))
34 //Zuordnung zwischen 1-d n*k und 2-d n x k Array: d_2d[i][j] = d_1d[i*k+j]
35 void htond12(char* x, double* d, int n, int k);
36
37 //Rückumwandlung eines n x k double Arrays in ein 1-d Array der Länge n*k
38 //char muss Dimension 10*n*k haben (da jede Double umgewandelt 10 Bytes
    entspricht);
39 //double * ist ein Array mit Länge n*k; Daten sequentiell in x, dort
    zurückwandelbar mittels ntohd(char*, double**, int n, int k)
40 //Zuordnung zwischen 1-d n*k und 2-d n x k Array: d_2d[i][j] = d_1d[i*k+j]
41 void ntohd21(char* x, double* d, int n, int k);
42
43 #endif
```

### A.3. dn.cpp

```
1 #include <math.h>
2 #include <winsock2.h> // für htons etc // WICHTIG: beim Linker unter zus.
    Abhängigkeiten: ws2_32.lib einbinden!!
```

```

3
4 void htond(double d, unsigned int & a1, unsigned int & a2, unsigned short &
    exp){
5
6     double m;
7     int temp;
8
9     m=frexp(d,&temp); //m ist die Mantisse, temp der Exponent
10
11    if(m>=0 && temp>=0){
12        exp = (unsigned short) temp; //Exponent zur Basis 2
13        a1 = (unsigned int)1E8*m; //Digit 1 bis 8 von m + VZ von m
14        a2 = (unsigned int)(1E8*(m*1E8-a1)); //Digit 9-16 von m + VZ von exp
15    }
16    else if(m>=0 && temp<0){
17        exp = (unsigned short) -temp; //Exponent zur Basis 2
18        a1 = (unsigned int)1E8*m; //Digit 1 bis 8 von m + VZ von m
19        a2 = ((unsigned int)(1E8*(m*1E8-a1))); //Digit 9-16 von m
20        a2 |= 0x80000000; // VZ von exp
21    }
22    else if(m<0 && temp>=0){
23        exp = (unsigned short) temp; //Exponent zur Basis 2
24        a1 = ((unsigned int) (-1E8*m)); //Digit 1 bis 8 von m
25        a2 = (unsigned int)(-1E8*(m*1E8+a1)); //Digit 9-16 von m + VZ von exp
26        a1 |= 0x80000000; //VZ von m
27    }
28    else { //(m<0 && temp<0)
29        exp = (unsigned short) -temp; //Exponent zur Basis 2
30        a1 = ((unsigned int) (-1E8*m)); //Digit 1 bis 8 von m
31        a2 = (unsigned int)(-1E8*(m*1E8+a1)); //Digit 9-16 von m
32        a1 |= 0x80000000; //VZ von m
33        a2 |= 0x80000000; //VZ von exp
34    }
35
36    //Umwandlung in Netzwerk-Byteorder
37    a1 = htonl(a1);
38    a2 = htonl(a2);
39    exp = htons(exp);
40 }
41
42 double ntohd(unsigned int a1, unsigned int a2, unsigned short exp){
43
44     double m;
45     m = ntohl(a1) & 0x80000000 ? -(ntohl(a2)& 0x7FFFFFFF)*1E-16+(ntohl(a1) & 0
        x7FFFFFFF)*1E-8) : (ntohl(a2)& 0x7FFFFFFF)*1E-16+ntohl(a1)*1E-8;

```

```
46     return ntohl(a2) & 0x80000000 ? ldexp(m,-(int)(ntohs(exp) & 0x7FFF)) : ldexp
        (m,(int)ntohs(exp));
47
48 }
49
50 void htond(char* x, double* d, int n){
51
52     double m;
53     int temp;
54     for(int i=0; i<n; ++i){
55
56         m=frexp(d[i],&temp); //m ist die Mantisse, temp der Exponent
57
58         if(m>=0 && temp>=0){
59             *(unsigned short*)(x+i*10+8) = (unsigned short) temp; //Exponent zur
                Basis 2
60             *(unsigned int*)(x+10*i) = (unsigned int)1E8*m; //Digit 1 bis 8 von m +
                VZ von m
61             *(unsigned int*)(x+10*i+4) = (unsigned int)(1E8*(m*1E8-*(unsigned int*)(
                x+10*i))); //Digit 9-16 von m + VZ von exp
62         }
63         else if(m>=0 && temp<0){
64             *(unsigned short*)(x+i*10+8) = (unsigned short) -temp; //Exponent zur
                Basis 2
65             *(unsigned int*)(x+10*i) = (unsigned int)1E8*m; //Digit 1 bis 8 von m +
                VZ von m
66             *(unsigned int*)(x+10*i+4) = ((unsigned int)(1E8*(m*1E8-*(unsigned int*)(
                x+10*i)))); //Digit 9-16 von m
67             (*(unsigned int*)(x+10*i+4)) |= 0x80000000; // VZ von exp
68         }
69         else if(m<0 && temp>=0){
70             *(unsigned short*)(x+i*10+8) = (unsigned short) temp; //Exponent zur
                Basis 2
71             *(unsigned int*)(x+10*i) = ((unsigned int) (-1E8*m)); //Digit 1 bis 8 von
                m
72             *(unsigned int*)(x+10*i+4) = (unsigned int)(-1E8*(m*1E8+*(unsigned int*)(
                x+10*i))); //Digit 9-16 von m + VZ von exp
73             (*(unsigned int*)(x+10*i)) |= 0x80000000; //VZ von m
74         }
75         else { //(m<0 && temp<0)
76             *(unsigned short*)(x+i*10+8) = (unsigned short) -temp; //Exponent zur
                Basis 2
77             *(unsigned int*)(x+10*i) = ((unsigned int) (-1E8*m)); //Digit 1 bis 8 von
                m
```

```

78     *(unsigned int*)(x+10*i+4) = (unsigned int)(-1E8*(m*1E8+(*(unsigned int*)
      (x+10*i)))); //Digit 9-16 von m
79     (*(unsigned int*)(x+10*i)) |= 0x80000000; //VZ von m
80     (*(unsigned int*)(x+10*i+4)) |= 0x80000000; //VZ von exp
81 }
82
83 //Umwandlung in Netzwerk-Byteorder
84 *(unsigned int*)(x+10*i) = htonl(*(unsigned int*)(x+10*i));
85 *(unsigned int*)(x+10*i+4) = htonl(*(unsigned int*)(x+10*i+4));
86 *(unsigned short*)(x+i*10+8) = htons(*(unsigned short*)(x+i*10+8));
87 }
88 }
89
90 void ntohd(char* x, double* d, int n){
91
92     double m;
93     for(int i=0; i<n; ++i){
94         m = ntohl(*(unsigned int*)(x+10*i)) & 0x80000000 ? -((ntohl(*(unsigned int
          *)
          (x+10*i+4))& 0x7FFFFFFF)*1E-16+
95             (ntohl(*(unsigned int*)(x+10*i)) & 0x7FFFFFFF)*1E-8) : (ntohl(*(unsigned
          int*)(x+10*i+4)) & 0x7FFFFFFF)*1E-16+ntohl(*(unsigned int*)(x+10*i))
          *1E-8;
96         d[i] = ntohl(*(unsigned int*)(x+10*i+4)) & 0x80000000 ? ldexp(m,-(int)(
          ntohs(*(unsigned short*)(x+i*10+8)) & 0x7FFF)) : ldexp(m,(int)ntohs(*(
          unsigned short*)(x+i*10+8)));
97     }
98
99 }
100
101 void htond(char* x, double** d, int n, int k){
102
103     double m;
104     int temp;
105     int i = 0;
106
107     for(int p=0; p<k; ++p){
108
109         for(int r=0; r<n; ++r){
110
111             m=frexp(d[r][p],&temp); //m ist die Mantisse, temp der Exponent
112
113             if(m>=0 && temp>=0){
114                 *(unsigned short*)(x+i*10+8) = (unsigned short) temp; //Exponent zur
                  Basis 2

```

```
115     *(unsigned int*)(x+10*i) = (unsigned int)1E8*m; //Digit 1 bis 8 von m +
        VZ von m
116     *(unsigned int*)(x+10*i+4) = (unsigned int)(1E8*(m*1E8-*(unsigned int*)
        (x+10*i))); //Digit 9-16 von m + VZ von exp
117 }
118 else if(m>=0 && temp<0){
119     *(unsigned short*)(x+i*10+8) = (unsigned short) -temp; //Exponent zur
        Basis 2
120     *(unsigned int*)(x+10*i) = (unsigned int)1E8*m; //Digit 1 bis 8 von m +
        VZ von m
121     *(unsigned int*)(x+10*i+4) = ((unsigned int)(1E8*(m*1E8-*(unsigned int
        *)
        (x+10*i)))); //Digit 9-16 von m
122     (*(unsigned int*)(x+10*i+4)) |= 0x80000000; // VZ von exp
123 }
124 else if(m<0 && temp>=0){
125     *(unsigned short*)(x+i*10+8) = (unsigned short) temp; //Exponent zur
        Basis 2
126     *(unsigned int*)(x+10*i) = ((unsigned int) (-1E8*m)); //Digit 1 bis 8
        von m
127     *(unsigned int*)(x+10*i+4) = (unsigned int)(-1E8*(m*1E8+*(unsigned int
        *)
        (x+10*i))); //Digit 9-16 von m + VZ von exp
128     (*(unsigned int*)(x+10*i)) |= 0x80000000; //VZ von m
129 }
130 else { //(m<0 && temp<0)
131     *(unsigned short*)(x+i*10+8) = (unsigned short) -temp; //Exponent zur
        Basis 2
132     *(unsigned int*)(x+10*i) = ((unsigned int) (-1E8*m)); //Digit 1 bis 8
        von m
133     *(unsigned int*)(x+10*i+4) = (unsigned int)(-1E8*(m*1E8+*(unsigned int
        *)
        (x+10*i))); //Digit 9-16 von m
134     (*(unsigned int*)(x+10*i)) |= 0x80000000; //VZ von m
135     (*(unsigned int*)(x+10*i+4)) |= 0x80000000; //VZ von exp
136 }
137
138 //Umwandlung in Netzwerk-Byteorder
139 *(unsigned int*)(x+10*i) = htonl(*(unsigned int*)(x+10*i));
140 *(unsigned int*)(x+10*i+4) = htonl(*(unsigned int*)(x+10*i+4));
141 *(unsigned short*)(x+i*10+8) = htons(*(unsigned short*)(x+i*10+8));
142
143 ++i;
144 }
145
146 }
147 }
148
```

```

149 void ntohd(char* x, double** d, int n, int k){
150
151     double m;
152     int i=0;
153     for(int p=0; p<k; ++p){
154         for(int r=0; r<n; ++r){
155             m = ntohl(*(unsigned int*)(x+10*i)) & 0x80000000 ? -((ntohl(*(unsigned
156                 int*)(x+10*i+4)) & 0x7FFFFFFF)*1E-16+
157                 (ntohl(*(unsigned int*)(x+10*i)) & 0x7FFFFFFF)*1E-8) : (ntohl(*(unsigned
158                 int*)(x+10*i+4)) & 0x7FFFFFFF)*1E-16+ntohl(*(unsigned int*)(x+10*i)
159                 )*1E-8;
160             d[r][p] = ntohl(*(unsigned int*)(x+10*i+4)) & 0x80000000 ? ldexp(m,-(int)
161                 (ntohs(*(unsigned short*)(x+i*10+8)) & 0x7FFF)) : ldexp(m,(int)ntohs
162                 (*(unsigned short*)(x+i*10+8)));
163             ++i;
164         }
165     }
166 }
167
168 void htond12(char* x, double* d, int n, int k){
169
170     double m;
171     int temp;
172     int i = 0;
173
174     for(int p=0; p<k; ++p){
175
176         for(int r=0; r<n; ++r){
177
178             m=frexp(d[r*k+p],&temp); //m ist die Mantisse, temp der Exponent
179
180             if(m>=0 && temp>=0){
181                 *(unsigned short*)(x+i*10+8) = (unsigned short) temp; //Exponent zur
182                     Basis 2
183                 *(unsigned int*)(x+10*i) = (unsigned int)1E8*m; //Digit 1 bis 8 von m +
184                     VZ von m
185                 *(unsigned int*)(x+10*i+4) = (unsigned int)(1E8*(m*1E8-*(unsigned int*)
186                     (x+10*i))); //Digit 9-16 von m + VZ von exp
187             }
188             else if(m>=0 && temp<0){
189                 *(unsigned short*)(x+i*10+8) = (unsigned short) -temp; //Exponent zur
190                     Basis 2
191                 *(unsigned int*)(x+10*i) = (unsigned int)1E8*m; //Digit 1 bis 8 von m +
192                     VZ von m

```

```
184     *(unsigned int*)(x+10*i+4) = ((unsigned int)(1E8*(m*1E8-*(unsigned int
      *(x+10*i))))); //Digit 9-16 von m
185     *(unsigned int*)(x+10*i+4) |= 0x80000000; // VZ von exp
186 }
187 else if(m<0 && temp>=0){
188     *(unsigned short*)(x+i*10+8) = (unsigned short) temp; //Exponent zur
      Basis 2
189     *(unsigned int*)(x+10*i) = ((unsigned int) (-1E8*m)); //Digit 1 bis 8
      von m
190     *(unsigned int*)(x+10*i+4) = (unsigned int)(-1E8*(m*1E8+*(unsigned int
      *(x+10*i)))); //Digit 9-16 von m + VZ von exp
191     *(unsigned int*)(x+10*i) |= 0x80000000; //VZ von m
192 }
193 else { //(m<0 && temp<0)
194     *(unsigned short*)(x+i*10+8) = (unsigned short) -temp; //Exponent zur
      Basis 2
195     *(unsigned int*)(x+10*i) = ((unsigned int) (-1E8*m)); //Digit 1 bis 8
      von m
196     *(unsigned int*)(x+10*i+4) = (unsigned int)(-1E8*(m*1E8+*(unsigned int
      *(x+10*i)))); //Digit 9-16 von m
197     *(unsigned int*)(x+10*i) |= 0x80000000; //VZ von m
198     *(unsigned int*)(x+10*i+4) |= 0x80000000; //VZ von exp
199 }
200
201 //Umwandlung in Netzwerk-Byteorder
202 *(unsigned int*)(x+10*i) = htonl(*(unsigned int*)(x+10*i));
203 *(unsigned int*)(x+10*i+4) = htonl(*(unsigned int*)(x+10*i+4));
204 *(unsigned short*)(x+i*10+8) = htons(*(unsigned short*)(x+i*10+8));
205
206 ++i;
207 }
208
209 }
210
211 }
212
213 void ntohd21(char* x, double* d, int n, int k){
214
215     double m;
216     int i=0;
217     for(int p=0; p<k; ++p){
218         for(int r=0; r<n; ++r){
219             m = ntohl(*(unsigned int*)(x+10*i) & 0x80000000 ? -(ntohl(*(unsigned
                int*)(x+10*i+4))& 0x7FFFFFFF)*1E-16+
```

```

220     (ntohl(*(unsigned int*)(x+10*i)) & 0x7FFFFFFF)*1E-8) : (ntohl(*(unsigned
        int*)(x+10*i+4)) & 0x7FFFFFFF)*1E-16+ntohl(*(unsigned int*)(x+10*i)
        )*1E-8;
221     d[r*k+p] = ntohl(*(unsigned int*)(x+10*i+4)) & 0x80000000 ? ldexp(m,-(int
        )(ntohs(*(unsigned short*)(x+i*10+8)) & 0x7FFF)) : ldexp(m,(int)ntohs
        (*(unsigned short*)(x+i*10+8)));
222     ++i;
223     }
224 }
225
226 }

```

## A.4. exchange\_class\_windows.h

```

1  #include <string>
2  #include <windows.h>
3  #include "interface_baseclass.h"
4  //includes winsock2.h
5
6  //ws2_32.lib has to be included in additional linking dependencies!! --> done
    in WorkingModule
7
8  #ifndef EX_CLASS_WIN
9  #define EX_CLASS_WIN
10
11 class DataInit; //forward declaration
12 class ElementDataContainer;
13 class MBS;
14
15 class DataH: public interface_baseclass {
16
17 public:
18     //default constructor - if this is used, actual construction is done using
        init
19     DataH():isinitialized(false){};
20     DataH(const DataInit& a, ElementDataContainer* edc, const std::string&
        inputskript, MBS* mbs, double SPHmass=-1.);
21     //if SPHmass is specified, it is assigned to all particles instead of the
        parameter SPHparticlemass defined in the Model-txt
22
23     DataH(const DataH &); //copy constructor
24     DataH& operator=(const DataH&); //copy assignment
25     ~DataH();
26

```

```
27 void init(const DataInit& a, ElementDataContainer* edc, const std::string&
    inputskript, MBS* mbs, double SPHmass=-1.);
28 void sendrv();
29 void sendrvfull();
30 void sendrSPH();
31 void sendvSPH();
32 void getforce();
33 void getrSPH();
34 void getrohSPH();
35 void sendrohSPH();
36 void sendone(std::string) const; //sends one LIGGGHTS input script line to
    client
37 void send_command(std::string) const; //sends one command to client and
    responds appropriately, if e.g. data is sent from the client; possible
    commands: see exchange_class_Linux.h - int recv_command();
38 void set_timestep(double timestep); //sets and sends current size of
    timestep; has to be done before first "run" command
39 void closeTCP();
40 void getvSPH();
41 void send_ref_opt();
42 void send_ref_res();
43
44 protected:
45 SOCKET s; //s...server-socket (->listen mode), c...socket data exchange with
    client (after accept)
46 SOCKET c;
47 bool isinitialized; //true if memory was allocated and tcp connection was
    set up -> used in destructor
48 };
49
50 struct vec3D{
51     double c[3];
52     vec3D* next;
53 };
54
55 struct elemelist2D{
56     unsigned int el[2];
57     elemelist2D* next;
58 };
59
60 struct elemelist3D{
61     unsigned int el[3];
62     elemelist3D* next;
63 };
64
```

```

65 class DataInit {
66 public:
67     unsigned int n; //number of points
68     unsigned int nstat; //number of points which are assumed to be static points
        (boundaries fixed to ground)
69     //points with numers 0...(n-nstat-1) are treated as regular dynamic points,
        points with numbers n-nstat...n-1 are treated as static points, i.e.
70     //corresponding r and v is only exchanged in initialization; by default,
        nstat=0 (i.e., all points receive full data transfer)
71     unsigned int nel; //number of elements
72     unsigned int nSPH; //number of SPH particles
73     unsigned short dim; //dimension (2 for 2D, 3 for 3D)
74     elemList2D* el2D; //data with dynamic memory allocation
75     elemList3D* el3D;
76     vec3D* r;
77     vec3D* v;
78     vec3D* rSPH;
79     vec3D* vSPH;
80     DataInit(unsigned short d); // d = dim
81     void add_elem(int*);
82     inline void add_elem(int a,int b, int c=0){int temp[3]; temp[0]=a; temp[1]=b
        ; temp[2]=c; add_elem(temp);}
83     void add_point(double* r,double* v);
84     void add_pointSPH(double* r,double* v);
85     unsigned int get_el(int i, int j) const; // returns j-th point of element i;
        counting starts from 0;
86     double get_r(int i, int j) const; //returns j-th coordinate of i-th point;
        counting starts from 0;
87     double get_v(int i, int j) const;
88     double get_rSPH(int i, int j) const;
89     double get_vSPH(int i, int j) const;
90     void set_nstat(int a=0){nstat = a;}
91     ~DataInit();
92 };
93
94 #endif

```

## A.5. exchange\_class\_windows.cpp

```

1 #include "exchange_class_windows.h"
2 #include "interface_baseclass.h"
3 #include <fstream>
4 #include <cstdlib>
5 #include <windows.h>
6 //includes winsock2.h

```

```
7 #include "dn.h"
8 #include <iostream>
9 #include <sstream>
10
11 #include "MBS_includes.h"
12 #include "mbs.h"
13 //#include <assert.h>
14
15 using namespace std;
16
17 int startWinsock(){
18     WSADATA wsa;
19     return WSStartup(MAKEWORD(2,0),&wsa); //MAKEWORD ist ein Makro, das die
        Versionsnummer 2.0 in ein WORD (unsigned short) umwandelt
20 }
21
22 string convertInt(int number);
23 /*{
24 stringstream ss;//create a stringstream
25 ss << number;//add number to the stream
26 return ss.str();//return a string with the contents of the stream
27 */
28
29 string convertDouble(double number)
30 {
31     stringstream ss;//create a stringstream
32     ss << number;//add number to the stream
33     return ss.str();//return a string with the contents of the stream
34 }
35
36 DataH::DataH(const DataInit& a, ElementDataContainer* edc, const std::string&
        inputskript, MBS* mbs, double SPHmass){
37     init(a, edc, inputskript, mbs);
38 }
39
40 void DataH::init(const DataInit &a, ElementDataContainer* edc, const std::
        string& inputskript, MBS* mbs, double SPHmass){
41
42     isinitialized=true;
43     dt=0.0; //actual timestep is set using set_timestep(double)
44
45     int option = edc->TreeGetInt("LIGGGHTS_SPH_parameters.refinement_option",1);
46     if(option==0 || option==1 || option==2 || option==3)
47         set_refinement_option(option);
48     else
```

```

49     set_refinement_option(1);
50
51     set_dr(edc->TreeGetDouble("LIGGGHTS_SPH_parameters.refinement_resolution",
52                               edc->TreeGetDouble("LIGGGHTS_SPH_parameters.smoothinglength")));
53
54     //TCP-IP v4 server setup and initialization of IP, port, nSPH
55     //-----
56     long rc;
57     SOCKADDR_IN addr;
58     string temp1;
59     short port;
60     ifstream src;
61
62     port = short(edc->TreeGetInt("TCP_data.port"));
63     int ip1 = edc->TreeGetInt("TCP_data.ip1");
64     int ip2 = edc->TreeGetInt("TCP_data.ip2");
65     int ip3 = edc->TreeGetInt("TCP_data.ip3");
66     int ip4 = edc->TreeGetInt("TCP_data.ip4");
67     temp1.append(convertInt(ip1)).append(".").append(convertInt(ip2)).append(".")
68           .append(convertInt(ip3)).append(".").append(convertInt(ip4));
69
70     //get dimension
71     dim=a.dim;
72
73     //start Winsock
74     rc=startWinsock();
75
76     // ERROR CHECKING
77     if(rc!=0){
78         mbs->UO(UO_LVL_err).InstantMessageText(mystr("ERROR TCP/IP: start winsock,
79             error code: ") + mystr(rc) + mystr("\n"));
80         //system("PAUSE");
81         //return 1;
82     }
83     else{
84         mbs->UO() << "Winsock started" << "\n";
85     }
86
87     //create socket
88     s=socket(AF_INET,SOCK_STREAM,0);
89
90     // ERROR CHECKING
91     if(s==INVALID_SOCKET){

```

```
90     mbs->UO(UO_LVL_err).InstantMessageText(mystr("ERROR TCP/IP: Create socket,
          error code: ") + mystr(WSAGetLastError()) + mystr("\n"));
91     //system("PAUSE");
92     //return 1;
93 }
94 else{
95     mbs->UO()<< "Socket created" << "\n";
96 }
97
98 //bind server socket to fixed port/ip
99 memset(&addr,0,sizeof(SOCKADDR_IN)); //see also p.19, Beej's Guide
100 addr.sin_family=AF_INET;
101 addr.sin_port=htons(port);
102 addr.sin_addr.s_addr=inet_addr(temp1.c_str());
103 rc=bind(s,(SOCKADDR*)&addr,sizeof(SOCKADDR_IN));
104
105 //ERROR CHECKING
106 if(rc==SOCKET_ERROR){
107     mbs->UO(UO_LVL_err).InstantMessageText(mystr("ERROR TCP/IP: bind, error
          code: ") + mystr(WSAGetLastError()) + mystr("\n"));
108     //system("Pause");
109     //return 1;
110 }else{
111     mbs->UO()<< "Socket bound to IP " << temp1.c_str() << ", port " << port <<
          "\n";
112 }
113
114 //Listen-Modus für den Server-Socket
115 rc=listen(s,10);
116 //ERROR CHECKING
117 if(rc==SOCKET_ERROR){
118     mbs->UO(UO_LVL_err).InstantMessageText(mystr("ERROR TCP/IP: listen, error
          code: ") + mystr(WSAGetLastError()) + mystr("\n"));
119     //system("Pause");
120     //return 1;
121 }else{
122     mbs->UO()<< "server-socket is in listen-mode..." << "\n";
123 }
124
125 //Verbindung annehmen
126 c=accept(s,NULL,NULL);
127 //ERROR CHECKING
128 if(c==INVALID_SOCKET){
129     mbs->UO(UO_LVL_err).InstantMessageText(mystr("ERROR TCP/IP: accept, error
          code: ") + mystr(WSAGetLastError()) + mystr("\n"));
```

```

130     //return 1;
131 }else{
132     mbs->UO() << "new connection was accepted" << "\n";
133 }
134
135 //data transfer and initialization of LIGGGHTS and rSPH
136 //-----
137
138 //read in LIGGGHTS variable definitions from edc
139
140 ElementDataContainer* subedc = (edc->TreeFind("LIGGGHTS_SPH_parameters"))->
    GetEDC();
141 int subedc_len = subedc->Length();
142 for(int i=1; i<=subedc_len; ++i){
143     string vardef = "variable ";
144
145     vardef.append(subedc->Get(i).GetDataName()).append(" equal ").append(
        convertDouble(subedc->Get(i).GetDouble()));
146     sendone(vardef);
147 }
148 sendone("qqqq");
149
150 int convert;
151 nSPH=a.nSPH;
152 convert=htonl(nSPH);
153 send(c,reinterpret_cast<char *>(&convert),4,0);
154
155 string read;
156 unsigned int remain,length;
157 char trig[]="create_box"; // if this string is found at the beginning of an
    inputskript-line, nSPH particles are created at (0,0,0)
158 int trigl=strlen(trig);
159 bool testswitch = true;
160
161 //initial position for all particles; only used when particles are created
    in LIGGGHTS; later, overwritten with actual initial positions;
162 double xmi = 0.5*(edc->TreeGetDouble("LIGGGHTS_SPH_parameters.xmin")+edc->
    TreeGetDouble("LIGGGHTS_SPH_parameters.xmax"));
163 double ymi = 0.5*(edc->TreeGetDouble("LIGGGHTS_SPH_parameters.ymin")+edc->
    TreeGetDouble("LIGGGHTS_SPH_parameters.ymax"));
164 double zmi = 0.5*(edc->TreeGetDouble("LIGGGHTS_SPH_parameters.zmin")+edc->
    TreeGetDouble("LIGGGHTS_SPH_parameters.zmax"));
165 if(dim == 2) zmi = 0.0;
166
167 string createatoms = "create_atoms 1 single ";

```

```
168 createatoms.append(convertDouble(xmi)).append(" ").append(convertDouble(ymi)
    ).append(" ").append(convertDouble(zmi));
169 createatoms.append(" units box");
170
171 src.open(inputskript.c_str(),ios::binary|ios::in);
172 //assert(src.is_open());//"failed to open LIGGGHTS input script"
173 //error checking?
174
175 while(!getline(src,read).eof()){
176     cout << read << endl;
177     sendone(read);
178
179     //set mass and create particles after create_box command
180     if(strncmp(trig,read.c_str(),trigl)==0 && testswitch){
181         testswitch = false; //ensures that this is only done once
182
183         if(SPHmass > 0.){
184             string setmass = "mass 1 ";
185             setmass.append(convertDouble(SPHmass));
186             sendone(setmass);
187         }else{
188             string setmass = "mass 1 ${SPHparticlemass}";
189             sendone(setmass);
190         }
191
192         for(int i=0; i<nSPH; ++i){
193             //create SPH particles, initial positions (xmi,ymi,zmi) for all
                particles
194             //real initial coordinates are set via sendrSPH()
195             sendone(createatoms);
196         }
197     }
198 }
199
200 // }
201
202 sendone("qqqq");
203 src.close();
204 src.clear();
205
206 //memory allocation and initialization from DataInit object a
207 //-----
208
209 //if number of boundary points n is below ~50, size for transfer of f / r /
    v array is smaller than 1500 Bytes = TCP packet size
```

```
210 //apparently, in that case transfer is REALLY slow (WHY???) ---> create
    additional dummy boundary points, such that n = 60
211 //these points are not involved in any part of the calculation, as long as
    they are not referenced in the element list (so they can be arbitrarily
    initialized)
212
213 if(a.n > 60){
214     n = a.n;}
215 else{
216     n=60;
217 }
218 nel = a.nel;
219 convert=htonl(n);
220 send(c,reinterpret_cast<char *>(&convert),4,0);
221 nstat = a.nstat;
222 convert=htonl(nstat);
223 send(c,reinterpret_cast<char *>(&convert),4,0);
224 convert=htons(dim);
225 send(c,reinterpret_cast<char *>(&convert),2,0);
226
227 _r = new double[3*n];
228 _v = new double[3*n];
229 _f = new double[3*n];
230 for(int i=0; i<n; ++i){
231     for(int j=0; j<3; ++j){
232         if(i < a.n){
233             setr(i,j,a.get_r(i,j));
234             setv(i,j,a.get_v(i,j));
235         }
236         else{
237             setr(i,j,0.0); //dummy boundary points
238             setv(i,j,0.0);
239         }
240         setf(i,j,0.0); //force is set via getforce()
241     }
242 }
243
244 sendrvfull();
245
246 //create, initialize and send nel and el
247 _el = new unsigned int[dim*nel];
248 convert=htonl(nel);
249 send(c,reinterpret_cast<char *>(&convert),4,0);
250 for(int i=0; i<nel; ++i){
251     for(int j=0; j<dim; ++j){
```

```
252     setel(i,j,a.get_el(i,j));
253     convert=htonl(el(i,j));
254     send(c,reinterpret_cast<char *>(&convert),4,0);
255     }
256 }
257
258 //send refinement_option
259 send_ref_opt();
260 send_ref_res();
261
262 //initialize and send rSPH and vSPH
263 _rSPH = new double[3*nSPH];
264 _vSPH = new double[3*nSPH];
265 for(int i=0; i<nSPH; ++i){
266     for(int j=0; j<3; ++j){
267         setrSPH(i,j,a.get_rSPH(i,j));
268         setvSPH(i,j,a.get_vSPH(i,j));
269     }
270 }
271
272 sendrSPH();
273 sendvSPH();
274
275 _rohSPH = new double[nSPH];
276 zerorohSPH();
277
278 }
279
280 DataH::DataH(const DataH & obj){
281     if(isinitialized){
282         s=obj.s;
283         c=obj.c;
284         isinitialized=obj.isinitialized;
285         dt=obj.dt;
286         refinement_option = obj.refinement_option;
287         dr = obj.dr;
288         n=obj.n;
289         nstat=obj.nstat;
290         nSPH=obj.nSPH;
291         nel=obj.nel;
292         dim=obj.dim;
293         _rSPH = new double[3*nSPH];
294         _vSPH = new double[3*nSPH];
295         _rohSPH = new double[nSPH];
296         _el = new unsigned int[dim*nel];
```

```

297     _r = new double[3*n];
298     _v = new double[3*n];
299     _f = new double[3*n];
300     for(int i=0; i<3*n; ++i){
301         _f[i]=obj._f[i];
302         _v[i]=obj._v[i];
303         _r[i]=obj._r[i];
304     }
305     for(int i=0; i<dim*nel; ++i){
306         _el[i]=obj._el[i];
307     }
308     for(int i=0; i<3*nSPH; ++i){
309         _rSPH[i]=obj._rSPH[i];
310         _vSPH[i]=obj._vSPH[i];
311     }
312     for(int i=0; i<nSPH; ++i)
313         _rohSPH[i]=obj._rohSPH[i];
314 }
315 else{
316     s=obj.s;
317     c=obj.c;
318     isinitialized=obj.isinitialized;
319 }
320 }
321
322 DataH& DataH::operator=(const DataH& ref){
323     if(this != &ref){
324         if(isinitialized){
325             s=ref.s;
326             c=ref.c;
327             isinitialized=ref.isinitialized;
328             dt=ref.dt;
329             refinement_option = ref.refinement_option;
330             dr = ref.dr;
331             n=ref.n;
332             nstat=ref.nstat;
333             nSPH=ref.nSPH;
334             nel=ref.nel;
335             dim=ref.dim;
336
337             double* _rSPHtemp = new double[3*ref.nSPH];
338             double* _vSPHtemp = new double[3*ref.nSPH];
339             double* _rohSPHtemp = new double[ref.nSPH];
340             unsigned int* _eltemp = new unsigned int[ref.dim*nel];
341             double* _rtemp = new double[3*ref.n];

```

```
342     double* _vtemp = new double[3*ref.n];
343     double* _ftemp = new double[3*ref.n];
344
345     for(int i=0; i<3*ref.n; ++i){
346         _ftemp[i]=ref._f[i];
347         _vtemp[i]=ref._v[i];
348         _rtemp[i]=ref._r[i];
349     }
350     for(int i=0; i<ref.dim*ref.nel; ++i){
351         _eltemp[i]=ref._el[i];
352     }
353     for(int i=0; i<3*ref.nSPH; ++i){
354         _rSPHtemp[i]=ref._rSPH[i];
355         _vSPHtemp[i]=ref._vSPH[i];
356     }
357     for(int i=0; i<ref.nSPH; ++i)
358         _rohSPHtemp[i]=ref._rohSPH[i];
359
360     delete [] _rSPH;
361     delete [] _vSPH;
362     delete [] _rohSPH;
363     delete [] _r;
364     delete [] _f;
365     delete [] _v;
366     delete [] _el;
367
368     _rSPH = _rSPHtemp;
369     _vSPH = _vSPHtemp;
370     _rohSPH = _rohSPHtemp;
371     _r = _rtemp;
372     _v = _vtemp;
373     _f = _ftemp;
374     _el = _eltemp;
375
376 }
377 else{
378     s=ref.s;
379     c=ref.c;
380     isinitialized=ref.isinitialized;
381 }
382 }
383 return *this;
384 }
385
386 DataH::~DataH(){
```

```
387     if(isinitialized){
388
389         //close TCP sockets / cleanup
390         closeTCP();
391
392         //memory management
393
394         delete [] _r;
395         delete [] _v;
396         delete [] _f;
397         delete [] _rSPH;
398         delete [] _vSPH;
399         delete [] _rohSPH;
400         delete [] _el;
401     }
402 }
403
404 void DataH::closeTCP(){
405     closesocket(c); //error checking would be necessary here (e.g. after copy
406         construction)
407     closesocket(s);
408     WSACleanup();
409 }
410 void DataH::set_timestep(double timestep){
411     dt=timestep;
412     char temp[10];
413     int remain=10;
414     htons(temp,&dt,1);
415     while(remain!=0) remain-=send(c,temp+(10-remain),remain,0);
416 }
417
418 void DataH::send_ref_res(){
419     char temp[10];
420     int remain=10;
421     htons(temp,&dr,1);
422     while(remain!=0) remain-=send(c,temp+(10-remain),remain,0);
423 }
424
425 void DataH::getforce(){
426     char* mem = new char[30*(n-nstat)]; //memory which ntohs and htons is
427         working on; size = 10*(number of doubles in array)
428
429     //with consistency check
430     int remain=30*(n-nstat);
```

```
430 while(remain!=0) remain-=recv(c,mem+(30*(n-nstat)-remain),remain,0);
431 ntohd(mem,_f,3*(n-nstat));
432
433 delete [] mem;
434 }
435
436 void DataH::getrSPH(){
437 char* mem = new char[30*nSPH]; //memory which ntohd and htond is working on;
438 size = 10*(number of doubles in array)
439
440 //with consistency check
441 int remain=30*nSPH;
442 while(remain!=0) remain-=recv(c,mem+(30*nSPH-remain),remain,0);
443 ntohd(mem,_rSPH,3*nSPH);
444
445 delete [] mem;
446 }
447
448 void DataH::getvSPH(){
449 char* mem = new char[30*nSPH]; //memory which ntohd and htond is working on;
450 size = 10*(number of doubles in array)
451
452 //with consistency check
453 int remain=30*nSPH;
454 while(remain!=0) remain-=recv(c,mem+(30*nSPH-remain),remain,0);
455 ntohd(mem,_vSPH,3*nSPH);
456
457 delete [] mem;
458 }
459
460 void DataH::getrohSPH(){
461 char* mem = new char[10*nSPH]; //memory which ntohd and htond is working on;
462 size = 10*(number of doubles in array)
463
464 //with consistency check
465 int remain=10*nSPH;
466 while(remain!=0) remain-=recv(c,mem+(10*nSPH-remain),remain,0);
467 ntohd(mem,_rohSPH,nSPH);
468
469 delete [] mem;
470 }
471
```

```
472 void DataH::sendrSPH(){
473     char* mem = new char[30*nSPH]; //memory which ntohs and htons is working on;
         size = 10*(number of doubles in array)
474
475     //with consistency check
476     int remain=30*nSPH;
477     htons(mem,_rSPH,3*nSPH);
478     while(remain!=0) remain-=send(c,mem+(30*nSPH-remain),remain,0);
479
480     delete [] mem;
481
482 }
483
484 void DataH::sendvSPH(){
485     char* mem = new char[30*nSPH]; //memory which ntohs and htons is working on;
         size = 10*(number of doubles in array)
486
487     //with consistency check
488     int remain=30*nSPH;
489     htons(mem,_vSPH,3*nSPH);
490     while(remain!=0) remain-=send(c,mem+(30*nSPH-remain),remain,0);
491
492     delete [] mem;
493
494 }
495
496 void DataH::sendrohSPH(){
497     char* mem = new char[10*nSPH]; //memory which ntohs and htons is working on;
         size = 10*(number of doubles in array)
498
499     //with consistency check
500     int remain=10*nSPH;
501     htons(mem,_rohSPH,nSPH);
502     while(remain!=0) remain-=send(c,mem+(10*nSPH-remain),remain,0);
503
504     delete [] mem;
505
506 }
507
508 void DataH::sendrv(){
509     char* mem = new char[30*(n-nstat)]; //memory which ntohs and htons is
         working on; size = 10*(number of doubles in array)
510
511     //with consistency check
512     htons(mem,_r,3*(n-nstat));
```

```
513   int remain=30*(n-nstat);
514   while(remain!=0) remain-=send(c,mem+(30*(n-nstat)-remain),remain,0);
515
516   htond(mem,_v,3*(n-nstat));
517   remain=30*(n-nstat);
518   while(remain!=0) remain-=send(c,mem+(30*(n-nstat)-remain),remain,0);
519
520   delete [] mem;
521 }
522
523 void DataH::sendrvfull(){
524   char* mem = new char[30*n]; //memory which ntohs and htons is working on;
525                               size = 10*(number of doubles in array)
526
527   //with consistency check
528   htond(mem,_r,3*n);
529   int remain=30*n;
530   while(remain!=0) remain-=send(c,mem+(30*n-remain),remain,0);
531
532   htond(mem,_v,3*n);
533   remain=30*n;
534   while(remain!=0) remain-=send(c,mem+(30*n-remain),remain,0);
535
536   delete [] mem;
537 }
538
539 void DataH::sendone(string line) const {
540   unsigned int length=line.size()+1;
541   unsigned int remain = htonl(length);
542   send(c,reinterpret_cast<char *>(&remain),4,0);
543   remain = length;
544   char* temp = new char[length];
545   strcpy(temp,line.c_str());
546   while(remain!=0) remain-=send(c,temp+(length-remain),remain,0);
547   delete [] temp;
548 }
549
550 void DataH::send_ref_opt(){
551   unsigned int temp = htonl(refinement_option);
552   send(c,reinterpret_cast<char *>(&temp),4,0);
553 }
554
555 void DataH::send_command(string line) const {
556   unsigned int length=line.size()+1;
557   unsigned int remain = htonl(length);
```

```

557     send(c,reinterpret_cast<char *>(&remain),4,0);
558     remain = length;
559     char* temp = new char[length];
560     strcpy(temp,line.c_str());
561     while(remain!=0) remain-=send(c,temp+(length-remain),remain,0);
562
563     if(strncmp(temp,"send SPH",length)==0){
564         const_cast<DataH* const> (this)->getrSPH();
565         const_cast<DataH* const> (this)->getvSPH();
566         const_cast<DataH* const> (this)->getrohSPH();
567     }
568
569     if(strncmp(temp,"send f",length)==0){
570         const_cast<DataH* const> (this)->getforce();
571     }
572
573     if(strncmp(temp,"dummy",length)==0){
574     }
575
576     if(strncmp(temp,"recv rv",length)==0){
577         const_cast<DataH* const> (this)->sendrv();
578     }
579
580     delete [] temp;
581 }
582
583 DataInit::DataInit(unsigned short d){
584     dim = d;
585     r = new vec3D;
586     v = new vec3D;
587     r->next=NULL;
588     v->next=NULL;
589     rSPH = new vec3D;
590     vSPH = new vec3D;
591     rSPH->next=NULL;
592     vSPH->next=NULL;
593     n=0;
594     nel=0;
595     nSPH=0;
596     el2D = new elemList2D;
597     el3D = new elemList3D;
598     el2D->next=NULL;
599     el3D->next=NULL;
600 }
601

```

```
602 void DataInit::add_elem(int* points){
603     if(dim==2){
604         elemList2D* temp = e12D;
605         elemList2D* neu = new elemList2D;
606         while(temp->next) temp=temp->next;
607         for(int i=0; i<dim; ++i) (neu->el)[i]=points[i];
608         (neu->next)=NULL;
609         (temp->next)=neu;
610
611     }
612     else if(dim==3){ //dim==3
613         elemList3D* temp = e13D;
614         elemList3D* neu = new elemList3D;
615         while(temp->next) temp=temp->next;
616         for(int i=0; i<dim; ++i) (neu->el)[i]=points[i];
617         (neu->next)=NULL;
618         (temp->next)=neu;
619     }
620
621     nel++;
622 }
623
624 void DataInit::add_point(double* rglob, double* vglob){
625     vec3D* temp = r;
626     while(temp->next) temp=temp->next;
627     vec3D* neu = new vec3D;
628     for(int i=0; i<3; ++i) (neu->c)[i]=rglob[i];
629     (neu->next)=NULL;
630     (temp->next)=neu;
631
632     temp = v;
633     while(temp->next) temp=temp->next;
634     neu = new vec3D;
635     for(int i=0; i<3; ++i) (neu->c)[i]=vglob[i];
636     (neu->next)=NULL;
637     (temp->next)=neu;
638
639     n++;
640 }
641
642 void DataInit::add_pointSPH(double* rglob, double* vglob){
643     vec3D* temp = rSPH;
644     while(temp->next) temp=temp->next;
645     vec3D* neu = new vec3D;
646     for(int i=0; i<3; ++i) (neu->c)[i]=rglob[i];
```

```

647     (neu->next)=NULL;
648     (temp->next)=neu;
649
650     temp = vSPH;
651     while(temp->next) temp=temp->next;
652     neu = new vec3D;
653     for(int i=0; i<3; ++i) (neu->c)[i]=vglob[i];
654     (neu->next)=NULL;
655     (temp->next)=neu;
656
657     nSPH++;
658 }
659
660 unsigned int DataInit::get_el(int i, int j) const{ // get j-th point of
        element i
661     if(dim==2){
662         elemList2D* temp = el2D;
663         for(int m=0; m<i+1; ++m) temp = temp->next; //i+1 for counting from 0; 1st
            element actually is at el2D->next
664         return (temp->el)[j];
665     }
666     else if(dim==3){
667         elemList3D* temp = el3D;
668         for(int m=0; m<i+1; ++m) temp = temp->next;
669         return (temp->el)[j];
670     }
671 }
672
673 double DataInit::get_r(int i, int j) const { //get j-th coordinate of i-th
        point
674     vec3D* temp = r;
675     for(int m=0; m<i+1; ++m) temp = temp->next; //i+1 for counting from 0; 1st r
            actually is at r->next
676     return (temp->c)[j];
677 }
678
679 double DataInit::get_v(int i, int j) const { //get j-th coordinate of i-th
        point
680     vec3D* temp = v;
681     for(int m=0; m<i+1; ++m) temp = temp->next; //i+1 for counting from 0; 1st r
            actually is at r->next
682     return (temp->c)[j];
683 }
684

```

```
685 double DataInit::get_rSPH(int i, int j) const { //get j-th coordinate of i-th
        point
686     vec3D* temp = rSPH;
687     for(int m=0; m<i+1; ++m) temp = temp->next; //i+1 for counting from 0; 1st r
        actually is at r->next
688     return (temp->c)[j];
689 }
690
691 double DataInit::get_vSPH(int i, int j) const {
692     vec3D* temp = vSPH;
693     for(int m=0; m<i+1; ++m) temp = temp->next; //i+1 for counting from 0; 1st v
        actually is at r->next
694     return (temp->c)[j];
695 }
696
697 DataInit::~DataInit(){
698     elemList2D* temp = el2D;
699     elemList2D* temp1 = el2D->next;
700     while(temp1){
701         delete temp;
702         temp = temp1;
703         temp1=temp1->next;
704     }
705     delete temp;
706
707     elemList3D* temp3 = el3D;
708     elemList3D* temp31 = el3D->next;
709     while(temp31){
710         delete temp3;
711         temp3 = temp31;
712         temp31=temp31->next;
713     }
714     delete temp3;
715
716     vec3D* tempr = r;
717     vec3D* tempr1 = r->next;
718     while(tempr1){
719         delete tempr;
720         tempr = tempr1;
721         tempr1=tempr1->next;
722     }
723     delete tempr;
724
725     tempr = rSPH;
726     tempr1 = rSPH->next;
```

```
727 while(temp1){
728     delete tempr;
729     tempr = tempr1;
730     tempr1=tempr1->next;
731 }
732 delete tempr;
733
734 vec3D* tempv = v;
735 vec3D* tempv1 = v->next;
736 while(tempv1){
737     delete tempv;
738     tempv = tempv1;
739     tempv1=tempv1->next;
740 }
741 delete tempv;
742
743 tempv = vSPH;
744 tempv1 = vSPH->next;
745 while(tempv1){
746     delete tempv;
747     tempv = tempv1;
748     tempv1=tempv1->next;
749 }
750 delete tempv;
751
752 }
```

## A.6. fsi\_communication\_element.h

```
1  /*#*****
2  /*#
3  /*# filename:      fsi_ommunication_element.h
4  /*#
5  /*# project:      FSI
6  /*#
7  /*# author:       Markus Schoergenhuber, PG, JG
8  /*#
9  /*# generated:    March 2012
10 /*# description:   Model (master) & communication element for coupled (fluid
11                   structure interaction) simulation with Liggghts
12 /*#
13 /*# remarks:
14 /*# This file is part of the program package HOTINT and underlies the
15                   stipulations
```

```
15  ///  
16  ///  
17  ///  
18  ///  
19  ///  
20  ///  
21  ///  
22  ///  
23  ///  
24  ///  
25  ///  
26  
27  #include "exchange class Win.h"  
28  #include "MBS_includes_element.h"  
29  
30  using namespace std;  
31  
32  string convertInt(int number);  
33  
34  #pragma region FSI Communication Element  
35  
36  // one instance of this element has to be added to mbs (in function  
    Generate_FSI_04_Markus)  
37  // it provides communication via TCP/IP-communication-interface with Liggghts.  
38  // - TCP/IP-communication-interface initialized in method Initialize()  
39  // - outgoing and incoming data communication processed in method  
    StartTimeStep()  
40  // currently, PostNewtonStep(double t) only does redrawing  
41  // finalize called in ComputationFinished()  
42  //SPH particle positions are saved in XData inherited from Element  
43  
44  #ifndef FSI_COMMUNICATION_ELEMENT  
45  #define FSI_COMMUNICATION_ELEMENT  
46  
47  class FSI_Communication_Element : public Element  
48  {  
49  
50  protected:  
51
```

```
52 // for simulation driving (currently not in use)
53 int step;
54 int counter;
55
56 int dim;           // space dimension (2, or 3)
57 int nparticles;   // number of SPH particles (total)
58 int nequi;        // number of (LIGGGHTS-only) equilibration steps
59
60 double deltax;    //if deltax specified as a positive value, uniformy
                   //distributed displacements in the range [-0.5*deltax,0.5*deltax] are added
                   //to all particle positions
61
62 TArray<int> sph; //list containing the numbers of the SPHParticle2D (or
                   //later: some 3D) elements in MBS for SPH data management and visualization
                   //; element no sph(i) corresponds to
63 //particle i-1 in DataH (counting starts here from 0)
64 DataH dataobj; //coupling class
65
66 TArray<int> boundary; //boundary is a list containing the numbers of all
                       //GeomElements that are actually used as boundary elements (i.e. those
                       //which were created
67 //either using the method AddGeomLine2D or AddGeomTrig3D
68 TArray<int> boundary_flags; // boundary_flags(i) contains a flag
                               //corresponding to boundary element boundary(i); 1 ... regular element, 0
                               //... static boundary element (fixed to ground / mbs)
69 int nstat; //number of static boundary elements (fixed to MBS / with elemnr
            // =0)
70
71 TMatrix<int> loadlist; //in 2(3)D: Considering GeomElement with number
                       //boundary(i) with associated Element e, loadlist(j,i), i=1...boundary.
                       //Length(), j=1...2(3)
72 //contains the number of the load in e.loads acting on the j-th point of
                       //considered GeomElement (on element e)
73 bool isinitialized; //is set to 0 in construction, to 1 after TCP
                       //communication and DataH object have been set up
74
75 string inputsript;
76
77 ElementDataContainer* edc;
78
79 void InitializeDataCommunication();
80
81 void IncomingDataCommunication(int step);
82
83 double OutgoingDataCommunication();
```

```
84
85     void Finalize();
86
87 public:
88
89     // set element properties from outside via this method
90     void Set_FSI_Communication_Element(int dim, int nequi, ElementDataContainer*
        edc, std::string& inputscript, double deltaX=-1.);
91
92     //data access SPH particles 2D
93     void SetParticleData2D(int i, Vector2D& r, Vector2D& v, double rho);
94
95     void SetParticlePos2D(int i, Vector2D& r);
96
97     Vector2D GetParticlePos2D(int i);
98
99     Vector2D GetParticleVel2D(int i);
100
101     double GetParticleDensity2D(int i);
102
103     //data access SPH particles 3D
104     void SetParticleData3D(int i, Vector3D& r, Vector3D& v, double rho);
105
106     void SetParticlePos3D(int i, Vector3D& r);
107
108     Vector3D GetParticlePos3D(int i);
109
110     Vector3D GetParticleVel3D(int i);
111
112     double GetParticleDensity3D(int i);
113
114     //add one SPH particle in 2D
115     void AddSPH2D(MBS* mbs, const Vector& xg5, double radius, double mass, const
        Vector3D& color);
116
117     //add one SPH particle in 3D
118     void AddSPH3D(MBS* mbs, const Vector& xg7, double radius, double mass, const
        Vector3D& color);
119
120     //returns the global position of the k-th point of a GeomElement temp
121     Vector2D GetGlobPos2D(GeomElement* temp, int k);
122
123     //returns the global position of the k-th point of a GeomElement temp
124     Vector3D GetGlobPos3D(GeomElement* temp, int k);
125
```

```

126 //returns the global velocity of the k-th point of a GeomElement temp
127 Vector2D GetGlobVel2D(GeomElement* temp, int k);
128
129 //returns the global velocity of the k-th point of a GeomElement temp
130 Vector3D GetGlobVel3D(GeomElement* temp, int k);
131
132 //apply the force from boundary point DataH el(i-1,k-1) to corresponding
      point k of GeomElement with number boundary(i)
133 void ApplyForce2D(int i, int k);
134
135 //apply the force from boundary point DataH el(i-1,k-1) to corresponding
      point k of GeomElement with number boundary(i)
136 void ApplyForce3D(int i, int k);
137
138 //add a GeomLine2D as boundary element
139 void AddGeomLine2D(GeomLine2D& line);
140
141 //add a GeomTrig3D as boundary element
142 void AddGeomTrig3D(GeomTrig3D& trig);
143
144 //functions that use variables defined in the LIGGGHTS input scripts
145 //-----
146
147 //reset pair_style with actual viscosity
148 void ResetVisc();
149
150 //-----
151
152 //functions for particle and boundary creation in the model file
153 //-----
154
155 //NOTE: currently LIGGGHTS-sided only one constant mass is used for all
      particles - it is calculated as an average over all HotInt-sided defined
      particle masses!!
156
157 //fill a rectangular region with SPH particles on a regular grid, in xy plane
      ; initial velocity is 0
158 //xll...pos left lower vertex
159 //xru... right upper vertex
160 //h...approximate particle distance / lattice constant (exact if edge
      lengths are multiples of h)
161 //dens...density, rad...radius, mass...mass, col...color of all particles
162 void FillRectangle(Vector2D xll, Vector2D xru, double h, MBS* mbs, double
      dens, double rad, double mass, const Vector3D col);
163

```

```
164 //fill a rectangular region with SPH paricles on a regular grid, in xy plane
    ; initial velocity is 0; sets density to mass/(area per particle)
165 //(the appropriate value of the chosen configuration in 2D - note that mass
    in 2D is mass per depth unit)
166 //xll...pos left lower vertex
167 //xru... right upper vertex
168 //h...approximate particle distance / lattice constant (exact if edge
    lengths are multiples of h)
169 //rad...radius, mass...mass, col...color of all particles
170 void FillRectangleAutoDensity(Vector2D xll, Vector2D xru, double h, MBS* mbs
    , double rad, double mass, const Vector3D col);
171
172 //fill a rectangular region with SPH paricles on a regular grid, in xy plane
    ; initial velocity is 0; sets mass (i.e. a mass per depth unit in 2D) of
    particles to dens*(area per particle)
173 //currently: LIGGGHTS-sided only one constant mass is used for all particles
    - it is calculated as an average over all HotInt-sided defined particle
    masses
174 //xll...pos left lower vertex
175 //xru... right upper vertex
176 //h...approximate particle distance / lattice constant (exact if edge
    lengths are multiples of h)
177 //dens...density, rad...radius, mass...mass, col...color of all particles
178 void FSI_Communication_Element::FillRectangleAutoMass(Vector2D xll, Vector2D
    xru, double h, MBS* mbs, double dens, double rad, const Vector3D col);
179
180 //fill a rectangular region with SPH paricles on a regular grid, in xy plane
    ;
181 //initial velocity is uniformly distributed over [-0.5*v, 0.5*v] in every
    component;
182 //mass (i.e. a mass per depth unit in 2D) of particles is set to dens*(area
    per particle)
183 //currently: LIGGGHTS-sided only one constant mass is used for all particles
    - it is calculated as an average over all HotInt-sided defined particle
    masses
184 //xll...pos left lower vertex
185 //xru... right upper vertex
186 //h...approximate particle distance / lattice constant (exact if edge
    lengths are multiples of h)
187 //dens...density, rad...radius, mass...mass, col...color of all particles
188 void FSI_Communication_Element::FillRectangleAutoMass(Vector2D xll, Vector2D
    xru, double h, MBS* mbs, double dens, double rad, const Vector3D col,
    double v);
189
```

```

190 //fill a box with SPH paricles 3D on a regular grid, base face lies parallel
      to the xy plane; initial velocity is 0
191 //xll...pos left lower vertex of (rectangular) base
192 //xru... right upper vertex of (rectangular) top face
193 //h...approximate particle distance / lattice constant (exact if edge
      lengths are multiples of h)
194 //dens...density, rad...radius, mass...mass, col...color of all particles
195 void FillBox(Vector3D xll, Vector3D xru, double h, MBS* mbs, double dens,
      double rad, double mass, const Vector3D col);
196
197 //fill a box with SPH paricles 3D on a regular grid, base face lies parallel
      to the xy plane; initial velocity is 0; density is set to mass/(volume
      per particle)
198 //xll...pos left lower vertex of (rectangular) base
199 //xru... right upper vertex of (rectangular) top face
200 //h...approximate particle distance / lattice constant (exact if edge
      lengths are multiples of h)
201 //rad...radius, mass...mass, col...color of all particles
202 void FillBoxAutoDensity(Vector3D xll, Vector3D xru, double h, MBS* mbs,
      double rad, double mass, const Vector3D col);
203
204 //fill a box with SPH paricles 3D on a regular grid, base face lies parallel
      to the xy plane; initial velocity is 0; particle mass is set to dens*(
      volume per particle)
205 //currently: LIGGGHTS-sided only one constant mass is used for all particles
      - it is calculated as an average over all HotInt-sided defined particle
      masses
206 //xll...pos left lower vertex of (rectangular) base
207 //xru... right upper vertex of (rectangular) top face
208 //h...approximate particle distance / lattice constant (exact if edge
      lengths are multiples of h)
209 //dens...density, rad...radius, mass...mass, col...color of all particles
210 void FSI_Communication_Element::FillBoxAutoMass(Vector3D xll, Vector3D xru,
      double h, MBS* mbs, double dens, double rad, const Vector3D col);
211
212 //create a 2D boundary shape by adjacent line segments between points (p1,p2
      ), (p2,p3), ... (pn-1,pn)
213 //where {p1,...pn} is given by "points"
214 //number is the element number, to which the GeomElements are connected,
      width is line width, col...color
215 void CreateBoundary2D(MBS* mbs, int number, TArray<Vector2D> points, double
      wi, Vector3D col);
216
217 //import and add a stl-geometry defined in stl_file_name to an element
      elemnr as GeomTrig3Ds and use AlternativeShape for this element

```

```
218 //GeomTrig3Ds are registered as surface elements for FSI in
    fsi_communication_element, unless the flag AddToCommunicationElement is
    set to 0 (default is 1)
219 void AddSTLMeshTrigsToElement(MBS* mbs, int elemnr, mystr stl_file_name,
    double stretch_factor, const Vector3D& translation_vec, const Matrix3D&
    rotation_mat, const Vector3D& color, double transparency, int
    AddToCommunicationElement = 1);
220
221 //add a random offset uniformly distributed over [-0.5*x,0.5*x] to all
    particle coordinates
222 void FSI_Communication_Element::randomize_particle_positions(double x);
223
224 //-----
225
226 // default constructor - deprecated!
227 FSI_Communication_Element() {
228     isinitialized = false;
229     //loadlist.SetDim(3,0);
230 }
231
232 // main constructor - use this one in model function for creating an object
233 // nothing else to be done here
234 // set element properties by Set_FSI_Communication_Element
235 FSI_Communication_Element(MBS * mbs) : Element(mbs) {
236     isinitialized = false;
237     loadlist.SetDim(3,0);
238 }
239
240 // copy constructor
241 FSI_Communication_Element(const FSI_Communication_Element& e) : Element(e.
    mbs)
242 {
243     CopyFrom(e);
244 };
245
246 virtual void CopyFrom(const Element& e);
247
248 virtual Element* GetCopy();
249
250 virtual void Initialize();
251
252 virtual void StartTimeStep();
253
254 virtual void EndTimeStep();
255
```

```

256     virtual double PostNewtonStep(double t);
257
258     virtual int Dim() const {return dim;} // dimension of problem
259
260     virtual int DataS() const { return 0; } //old, not in use any more;
        nparticles*Dim(); } // size of xdata
261
262     virtual void DrawElement(){}
263
264     // these functions need to be defined just to prevent the program from a
        crash (due to inheritance from Element)
265     virtual Vector3D GetPosD() const { return Vector3D(); }
266     virtual Vector3D GetRefPosD() const { return Vector3D(); }
267 };
268
269 #endif
270
271 #pragma endregion

```

## A.7. fsi\_communication\_element.cpp

```

1  //*****
2  //#
3  //# filename:      fsi_ommunication_element.cpp
4  //#
5  //# project:       FSI
6  //#
7  //# author:        Markus Schoergenhummer, PG, JG
8  //#
9  //# generated:     March 2012
10 //# description:   Model (master) & communication element for coupled (fluid
        structure interaction) simulation with Liggghts
11 //#
12 //# remarks:
13 //#
14 //# This file is part of the program package HOTINT and underlies the
        stipulations
15 //# of the scientific or license agreement. It is therefore emphasized not to
        copy
16 //# or redistribute this file. The use of this file is only permitted for
        academic or scholar
17 //# research. It is forbidden to use any part of this code for military
        applications!
18 //# The Developer does not assume any liability for this code or for results
        obtained

```

```
19  /// within its use, nor shall he be liable for any direct, indirect, or other
    damage
20  /// resulting from use of this code.
21  ///
22  /// bug reports are welcome!!!
23  /// WWW: http://tmech.mechatronik.uni-linz.ac.at/staff/gerstmayr/gerstmayr.html
    html
24  /// email: jg@jku.at
25  ///*****
26
27  #include "fsi_communication_element.h"
28
29  #include "MBS_includes_element.h"
30  #include "ExtendedElements/sph_particle2D.h"
31  #include "ExtendedElements/sph_particle3D.h"
32  ///#include "csg_geometry.h"
33
34  #include <string>
35  #include <sstream>
36  #include <iostream>
37  #include <fstream>
38  #include <math.h>
39  #include <windows.h>
40
41  using namespace std;
42
43  string convertInt(int number)
44  {
45      stringstream ss;///create a stringstream
46      ss << number;///add number to the stream
47      return ss.str();///return a string with the contents of the stream
48  }
49
50  ///this function sorts the lists boundary and loadlist and corresponding
    flaglist boundary_flags such, that the first elements have boundary_flags(
    i) = 1,
51  ///and all remaining subsequent elements have boundary_flags(i)=0
52  ///boundary_flags(i) = 0 or 1 corresponds to element boundary(i)
53  ///return value is number of elements with flag 0
54  int sort(TArray<int>& boundary, TMatrix<int>& loadlist, TArray<int>&
    boundary_flags, int dim){
55      int count = 0;
56      TArray<int> boundary_temp;
57      TMatrix<int> loadlist_temp;
58
```

```

59  for(int flag=1; flag>=0; --flag){
60      for(int i=1; i<=boundary.Length(); ++i){
61          if(boundary_flags(i)==flag){
62              boundary_temp.Add(boundary(i));
63              loadlist_temp.Add(1,loadlist(1,i));
64              loadlist_temp.Add(2,loadlist(2,i));
65              if(dim==3) loadlist_temp.Add(3,loadlist(3,i));
66          }
67          if(boundary_flags(i)==0 && flag==1) ++count;
68      }
69  }
70
71  for(int i=1; i<=boundary.Length()-count; ++i)
72      boundary_flags(i)=1;
73  for(int i=boundary.Length()-count+1; i<=boundary.Length(); ++i)
74      boundary_flags(i)=0;
75
76  boundary=boundary_temp;
77  loadlist=loadlist_temp;
78  return count;
79  }
80
81  // one instance of this element has to be added to mbs (in function
      Generate_FSI_04_Markus)
82  // it provides communication via TCP/IP-communication-interface with Liggghts.
83  // - TCP/IP-communication-interface initialized in method Initialize()
84  // - outgoing and incoming data communication processed in method
      StartTimeStep()
85  // currently, PostNewtonStep(double t) only does redrawing
86  // finalize called in ComputationFinished()
87  //SPH particle positions are saved in XData inherited from Element
88
89  // these three methods need to be implemented for enabling communication
90  void FSI_Communication_Element::InitializeDataCommunication()
91  {
92
93      double r1[3],v1[3];
94      DataInit initobj(dim);
95
96      //sort boundary and flags such that first all dynamic elements are listed,
          then subsequently all static ones
97      nstat = sort(boundary,loadlist,boundary_flags,dim);
98
99      //add random displacements to particles, if deltax was set to a value >0
100     if(deltax>0) randomize_particle_positions(deltax);

```

```
101
102 // initialize communication interface
103
104 //create DataInit instance locally and initialize SPH with data from
      SPHParticle2D (numbers stored in Tarray sph)
105 //initialize a DataH object (inclusion as member in this class)
106
107 if(dim == 2){
108
109     Vector2D rvtemp;
110     GeomElement* temp;
111
112     //create r,v, el
113     r1[2]=0.0;
114     v1[2]=0.0;
115
116     //first add all dynamic (non-static points), then add all static points (
      boundary array must have been sorted previously)
117 for(int j=1; j<=boundary.Length(); ++j){
118     temp = mbs->GetDrawElement(boundary(j));
119     for(int k=1; k<=2; ++k){ //2 points for 2D case / GeomLine2D
120         rvtemp = GetGlobPos2D(temp,k);
121         r1[0]=rvtemp(1);
122         r1[1]=rvtemp(2);
123         rvtemp = GetGlobVel2D(temp,k);
124         v1[0]=rvtemp(1);
125         v1[1]=rvtemp(2);
126         initobj.add_point(r1,v1);
127         /*
128         //now this is done directly in AddGeomLine2D
129         //create and add loads, initialized with 0.0, and create loadlist --
      loads still have to be linked to elements (see below)
130         if(temp->GetElnum()!=0){
131             load.SetForceVector2D(Vector2D(0.0,0.0),temp->GetLocPoint2D(k));
132             const_cast<Element*>(temp->GetElement()).AddLoad(load);
133             loadlist.Add(k,temp->GetElement().NLoads());
134         }
135         */
136     }
137     initobj.add_elem(initobj.n-2,initobj.n-1);
138     //GeomElement with the number boundary(j) corresponds to el[j-1] in DataH
      (shift of 1 because counting starts from 1 in TArray boundary, but
      from 0 in el)
139 }
140
```

```

141 //set number of static boundary points
142 initobj.set_nstat(nstat*2);
143
144 //create rSPH, vSPH
145 for(int i=1; i<=nparticles; ++i){
146     rvtemp=GetParticlePos2D(i);
147     r1[0]=rvtemp(1);
148     r1[1]=rvtemp(2);
149     rvtemp=GetParticleVel2D(i);
150     v1[0]=rvtemp(1); //velocities of SPH particles are initialized to 0
151     v1[1]=rvtemp(2);
152     initobj.add_pointSPH(r1,v1);
153 }
154 }
155 else if(dim == 3){
156
157     Vector3D rvtemp;
158     GeomElement* temp;
159     int tcount = 0;
160
161     //create r,v,el
162     //first add all dynamic (non-static points), then add all static points (
163     //boundary array must have been sorted previously)
164     for(int j=1; j<=boundary.Length(); ++j){
165         temp = mbs->GetDrawElement(boundary(j));
166         for(int k=1; k<=3; ++k){ //2 points for 2D case / GeomLine2D
167             rvtemp = GetGlobPos3D(temp,k);
168             r1[0]=rvtemp(1);
169             r1[1]=rvtemp(2);
170             r1[2]=rvtemp(3);
171             rvtemp = GetGlobVel3D(temp,k);
172             v1[0]=rvtemp(1);
173             v1[1]=rvtemp(2);
174             v1[2]=rvtemp(3);
175             initobj.add_point(r1,v1);
176         }
177         initobj.add_elem(initobj.n-3,initobj.n-2,initobj.n-1);
178         //GeomElement with the number boundary(j) corresponds to el[j-1] in DataH
179         // (shift of 1 because counting starts from 1 in TArray boundary, but
180         // from 0 in el)
181     }
182
183     //set number of static boundary points
184     initobj.set_nstat(nstat*3);
185

```

```
183     //create rSPH, vSPH
184     for(int i=1; i<=nparticles; ++i){
185         rvtemp=GetParticlePos3D(i);
186         r1[0]=rvtemp(1);
187         r1[1]=rvtemp(2);
188         r1[2]=rvtemp(3);
189         rvtemp=GetParticleVel3D(i);
190         v1[0]=rvtemp(1); //velocities of SPH particles are initialized to 0
191         v1[1]=rvtemp(2);
192         v1[2]=rvtemp(3);
193         initobj.add_pointSPH(r1,v1);
194     }
195
196 }
197 else
198 {
199     mbs->UO(UO_LVL_warn) << "warning: FSI_Communication_Element is implemented
200         for 2D-case or 3D-case only!\n";
201 }
202
203 assert( Dim() == 2 || Dim() == 3 );
204
205 /*
206 //since loads are added now directly in AddGeomLine2D in the model file,
207     this is already done in Assemble()
208 //link loads to elements (normally this is done in MBS::Assemble() in the
209     model file -- should Assemble be called here to be sure?)
210 for(int i=1; i<=mbs->NE(); ++i){
211     mbs->GetElement(i).LinkLoads();
212 }
213 */
214
215 //initialize DataH instance dataobj, set up TCP connection, ...
216 //dataobj.init(initobj,inscript,initTCP);//actual initialization procedure
217     of DataH instance, and Linux-sided DataL instance(s) with LAMMPS instance
218     (s)
219
220
221 double averageSPHmass = 0.;
222 for(int i=1; i<=sph.Length(); ++i){
223     averageSPHmass+=mbs->GetElement(sph(i)).GetMass();
224 }
225 averageSPHmass/=double(sph.Length());
226
227 dataobj.init(initobj,edc,inputsript,mbs,averageSPHmass);
228 isinitialized = true;
229
```

```

223 //initialize densities; v and r have already been initialized in DataInit
      object
224 if(dim == 2){
225     for(int i=0; i<nparticles; ++i){
226         dataobj.setrohSPH(i,GetParticleDensity2D(i+1));
227     }
228 }
229 else if(dim == 3){
230     for(int i=0; i<nparticles; ++i){
231         dataobj.setrohSPH(i,GetParticleDensity3D(i+1));
232     }
233 }
234 else
235 {
236     mbs->UO(UO_LVL_warn) << "warning: FSI_Communication_Element is implemented
          for 2D-case or 3D-case only!\n";
237 }
238 assert( Dim() == 2 || Dim() == 3 );
239
240 dataobj.sendrohSPH();
241
242 mbs->UO()<< mystr("Total number of SPH particles: ") + mystr(int(dataobj.nSPH)
          ) + mystr("\n");
243 mbs->UO()<< mystr("with averaged constant mass (mass per depth unit) in 3D
          (2D) of ") + mystr(averageSPHmass) + mystr(" kg (kg/m)") + mystr("\n");
244 mbs->UO()<< mystr("and nominal density of ") + mystr(edc->TreeGetDouble("
          LIGGGHTS_SPH_parameters.SPHdensity")) + mystr(" kg/m^3") + mystr("\n");
245 mbs->UO()<< mystr("Total number of surface elements for FSI: ") + mystr(int(
          dataobj.nel)) + mystr("\n");
246 mbs->UO()<< mystr("Total number of FSI surface points: ") + mystr(int(dataobj.
          n)) + mystr("\n");
247 mbs->UO()<< mystr("Number of static FSI surface points: ") + mystr(int(dataobj
          .nstat)) + mystr("\n");
248
249 //dataobj.set_timestep(mbs->GetStepSize()); //set size of time step for the
          following steps
250 dataobj.set_timestep(1E-5);
251 dataobj.sendone(string("run ").append(convertInt(nequi))); // initialization
          and initial configuration with "run 0";
252 // "run x" (x a positive integer) will run x LIGGGHTS steps with constant (
          initial) boundaries (e.g. for equilibration)
253 //here a very large viscosity is used
254 ResetVisc(); //set viscosity to actual value
255 dataobj.sendone("run 0"); //recalculate forces and densities for actual
          viscosity before moving on to timestepping

```

```
256
257 }
258
259 void FSI_Communication_Element::IncomingDataCommunication(int step)
260 {
261     static int stepcount = 0;
262     ++stepcount;
263
264     if(dim == 2){
265
266         // incoming data communication
267
268         //get full SPH data set only in time intervals solset.storedata (in which
                the solution is stored)
269         //else: only forces (minimal data transfer), in given intervals (numbers of
                timesteps)
270
271         //see also int TimeInt::StoreResultsIsOn()
272
273         if(mbs->GetSolSet().storedata == 0)
274             dataobj.send_command("dummy");
275         else if(mbs->GetSolSet().storedata == -2){
276             dataobj.send_command("send SPH");
277             //write current SPH data to XData via SPHParticle2D::SetAllData(...)
278             for(int i=1; i<=nparticles; ++i)
279                 SetParticleData2D(i,Vector2D(dataobj.rSPH(i-1,0),dataobj.rSPH(i-1,1)),
                    Vector2D(dataobj.vSPH(i-1,0),dataobj.vSPH(i-1,1)),dataobj.rohSPH(i
                    -1));
280         }
281         else if(mbs->GetSolSet().storedata == -1 && (mbs->GetTime()+ mbs->
                GetStepSize() + 0.1*mbs->GetSolSet().minstepsize >= mbs->laststoredata
                + mbs->GetSolSet().maxstepsize)){
282             dataobj.send_command("send SPH");
283             //write current SPH data to XData via SPHParticle2D::SetAllData(...)
284             for(int i=1; i<=nparticles; ++i)
285                 SetParticleData2D(i,Vector2D(dataobj.rSPH(i-1,0),dataobj.rSPH(i-1,1)),
                    Vector2D(dataobj.vSPH(i-1,0),dataobj.vSPH(i-1,1)),dataobj.rohSPH(i
                    -1));
286         }
287         else if(mbs->GetSolSet().storedata > 0 && (mbs->GetTime()+ mbs->GetStepSize
                (+) + 0.1*mbs->GetSolSet().minstepsize) >= mbs->laststoredata + mbs->
                GetSolSet().storedata){
288             dataobj.send_command("send SPH");
289             //write current SPH data to XData via SPHParticle2D::SetAllData(...)
290             for(int i=1; i<=nparticles; ++i)
```

```

291     SetParticleData2D(i,Vector2D(dataobj.rSPH(i-1,0),dataobj.rSPH(i-1,1)),
        Vector2D(dataobj.vSPH(i-1,0),dataobj.vSPH(i-1,1)),dataobj.rohSPH(i
            -1));
292 }
293 else
294     dataobj.send_command("dummy");
295
296 //if(stepcount == edc->TreeGetInt("HOTINT_SPH_parameters.exchangeperiod"))
297 if(1){
298     dataobj.send_command("send f");
299     //apply forces in DataH::f appropriately to all non-static elements (
        elementnr != 0)
300     for(int i=1; i<=boundary.Length()-nstat; ++i)
301         for(int k=1; k<=2; ++k){
302             ApplyForce2D(i,k);
303         }
304
305     stepcount = 0;
306 }
307 else{
308     dataobj.send_command("dummy");
309 }
310
311 }
312 else if(dim == 3){
313
314     //compare to completely analogous dim==2 case
315
316     //TMStartTimer(30);
317
318     if(mbs->GetSolSet().storedata == 0){
319         dataobj.send_command("dummy");
320     }
321     else if(mbs->GetSolSet().storedata == -2){
322         dataobj.send_command("send SPH");
323
324         //write current (updated) SPH data to XData via SPHParticle2D::SetAllData
            (...)
325         for(int i=1; i<=nparticles; ++i)
326             SetParticleData3D(i,Vector3D(dataobj.rSPH(i-1,0),dataobj.rSPH(i-1,1),
                dataobj.rSPH(i-1,2)),Vector3D(dataobj.vSPH(i-1,0),dataobj.vSPH(i
                    -1,1),dataobj.vSPH(i-1,2)),dataobj.rohSPH(i-1));
327     }
328     else if(mbs->GetSolSet().storedata == -1 && (mbs->GetTime()+ mbs->
        GetStepSize() + 0.1*mbs->GetSolSet().minstepsize >= mbs->laststoredata

```

```
        + mbs->GetSolSet().maxstepsize)){
329     dataobj.send_command("send SPH");
330     //write current (updated) SPH data to XData via SPHParticle2D::SetAllData
        (...)
331     for(int i=1; i<=nparticles; ++i)
332         SetParticleData3D(i,Vector3D(dataobj.rSPH(i-1,0),dataobj.rSPH(i-1,1),
            dataobj.rSPH(i-1,2)),Vector3D(dataobj.vSPH(i-1,0),dataobj.vSPH(i
            -1,1),dataobj.vSPH(i-1,2)),dataobj.rohSPH(i-1));
333     }
334     else if(mbs->GetSolSet().storedata > 0 && (mbs->GetTime()+ mbs->GetStepSize
        ()+ 0.1*mbs->GetSolSet().minstepsize) >= mbs->laststoredata + mbs->
        GetSolSet().storedata){
335         dataobj.send_command("send SPH");
336         //write current (updated) SPH data to XData via SPHParticle2D::SetAllData
            (...)
337         for(int i=1; i<=nparticles; ++i)
338             SetParticleData3D(i,Vector3D(dataobj.rSPH(i-1,0),dataobj.rSPH(i-1,1),
                dataobj.rSPH(i-1,2)),Vector3D(dataobj.vSPH(i-1,0),dataobj.vSPH(i
                -1,1),dataobj.vSPH(i-1,2)),dataobj.rohSPH(i-1));
339     }
340     else
341         dataobj.send_command("dummy");
342
343     //TMStopTimer(30);
344
345     //if(stepcount == edc->TreeGetInt("HOTINT_SPH_parameters.exchangeperiod"))
346     if(1){
347         dataobj.send_command("send f");
348         //apply forces in DataH::f appropriately to all non-static elements (
            elementnr != 0)
349         for(int i=1; i<=boundary.Length()-nstat; ++i)
350             for(int k=1; k<=3; ++k){
351                 ApplyForce3D(i,k);
352             }
353
354         stepcount = 0;
355     }
356     else{
357         dataobj.send_command("dummy");
358     }
359 }
360 else
361 {
362     mbs->UO(UO_LVL_warn) << "warning: FSI_Communication_Element is implemented
        for 2D-case or 3D-case only!\n";
```

```

363 }
364 assert( Dim() == 2 || Dim() == 3 );
365
366 }
367
368 double FSI_Communication_Element::OutgoingDataCommunication()
369 {
370     static int stepcount = 0;
371     ++stepcount;
372
373     // set timestep
374     dataobj.set_timestep(mbs->GetStepSize());
375
376     //if(stepcount == edc->TreeGetInt("HOTINT_SPH_parameters.exchangeperiod"))
377     if(1){
378
379         if(dim == 2){
380
381             Vector2D rvtemp;
382             GeomElement* temp;
383             int p1;
384
385             // outgoing data communication
386
387             //update rv, except for static elements
388             for(int i=1; i<=boundary.Length()-nstat; ++i){
389                 temp=mbs->GetDrawElement(boundary(i));
390                 for(int j=1; j<=2; ++j){
391                     rvtemp=GetGlobPos2D(temp,j);
392                     p1=dataobj.el(i-1,j-1);
393                     dataobj.setr(p1,0,rvtemp(1));
394                     dataobj.setr(p1,1,rvtemp(2));
395                     rvtemp=GetGlobVel2D(temp,j);
396                     dataobj.setv(p1,0,rvtemp(1));
397                     dataobj.setv(p1,1,rvtemp(2));
398                 }
399             }
400
401         }
402         else if(dim == 3){
403
404             Vector3D rvtemp;
405             GeomElement* temp;
406             int p1;
407

```

```
408     // outgoing data communication
409
410     //update rv, except for static elements
411     for(int i=1; i<=boundary.Length()-nstat; ++i){
412         temp=mbs->GetDrawElement(boundary(i));
413         for(int j=1; j<=3; ++j){
414             rvtemp=GetGlobPos3D(temp,j);
415             p1=dataobj.el(i-1,j-1);
416             dataobj.setr(p1,0,rvtemp(1));
417             dataobj.setr(p1,1,rvtemp(2));
418             dataobj.setr(p1,2,rvtemp(3));
419             rvtemp=GetGlobVel3D(temp,j);
420             dataobj.setv(p1,0,rvtemp(1));
421             dataobj.setv(p1,1,rvtemp(2));
422             dataobj.setv(p1,2,rvtemp(3));
423         }
424     }
425
426 }
427 else
428 {
429     mbs->UO(UO_LVL_warn) << "warning: FSI_Communication_Element is
         implemented for 2D-case or 3D-case only!\n";
430 }
431 assert( Dim() == 2 || Dim() == 3 );
432
433 //sendrv and sendone
434
435 TMStartTimer(28);
436
437 dataobj.send_command("recv rv");
438 // mbs->UO() << dataobj.r(0,0) << "\n";
439
440 TMStopTimer(28);
441
442 stepcount = 0;
443 }
444 else{
445     dataobj.send_command("dummy");
446 }
447
448 dataobj.sendone("run 1 pre no post no"); //difference to "run 1": neighbor
         lists, forces,... are not recalculated, because still valid from previous
         step (except for very first step);
449 // full timing statistics are not printed
```

```

450     return 0.;
451 }
452
453 void FSI_Communication_Element::Finalize(){
454     //send termination string sendone("qqqq");
455     //exchange timing / error messages etc...
456
457     dataobj.set_timestep(mbs->GetStepSize()); //this and the next line are just
         for synchronization reasons
458     dataobj.send_command("dummy");
459     dataobj.sendone("qqqq");
460     dataobj.closeTCP(); //is done in destructor again... possible errors?
461     isinitialized=false;
462
463 }
464
465 // set element properties from outside via this method
466 void FSI_Communication_Element::Set_FSI_Communication_Element(int dim, int
         nequi, ElementDataContainer* edc, std::string& inputscrip, double deltax)
467 {
468     isinitialized = false;
469     FSI_Communication_Element::nequi = nequi;
470
471     FSI_Communication_Element::dim = dim;
472     FSI_Communication_Element::nparticles = sph.Length();
473
474     FSI_Communication_Element::edc = edc;
475     FSI_Communication_Element::inputscrip = inputscrip;
476
477     FSI_Communication_Element::deltax = deltax;
478
479     if (Dim() != 2 && Dim() != 3)
480     {
481         mbs->UO(UO_LVL_warn) << "warning: FSI_Communication_Element is implemented
         for 2D-case or 3D-case only!\n";
482     }
483     assert( Dim() == 2 || Dim() == 3 );
484 }
485
486 //data access SPH particles 2D
487
488 void FSI_Communication_Element::SetParticleData2D(int i, Vector2D& r, Vector2D
         & v, double rho){
489     (reinterpret_cast<SPHParticle2D*>(&(mbs->GetElement(sph(i)))))->SetAllData(r
         .X(), r.Y(), v.X(), v.Y(), rho);

```

```
490 }
491
492 void FSI_Communication_Element::SetParticlePos2D(int i, Vector2D& r){
493     (reinterpret_cast<SPHParticle2D*>(&(mbs->GetElement(sph(i))))->SetPosition(
494         r.X(), r.Y()));
495 }
496
497 Vector2D FSI_Communication_Element::GetParticlePos2D(int i){
498     return mbs->GetElement(sph(i)).GetRefPos2D();
499 }
500
501 Vector2D FSI_Communication_Element::GetParticleVel2D(int i){
502     //return Vector2D(0.,0.); //for testing ONLY
503     return mbs->GetElement(sph(i)).GetRefVel2D();
504 }
505
506 double FSI_Communication_Element::GetParticleDensity2D(int i){
507     return (reinterpret_cast<SPHParticle2D*>(&(mbs->GetElement(sph(i))))->
508         GetDensity());
509 }
510
511 //data access SPH particles 3D
512
513 void FSI_Communication_Element::SetParticleData3D(int i, Vector3D& r, Vector3D
514     & v, double rho){
515     (reinterpret_cast<SPHParticle3D*>(&(mbs->GetElement(sph(i))))->SetAllData(r
516         .X(), r.Y(), r.Z(), v.X(), v.Y(), v.Z(), rho);
517 }
518
519 void FSI_Communication_Element::SetParticlePos3D(int i, Vector3D& r){
520     (reinterpret_cast<SPHParticle3D*>(&(mbs->GetElement(sph(i))))->SetPosition(
521         r.X(), r.Y(), r.Z());
522 }
523
524 Vector3D FSI_Communication_Element::GetParticlePos3D(int i){
525     return mbs->GetElement(sph(i)).GetRefPos();
526 }
527
528 Vector3D FSI_Communication_Element::GetParticleVel3D(int i){
529     return mbs->GetElement(sph(i)).GetRefVel();
530 }
531
532 double FSI_Communication_Element::GetParticleDensity3D(int i){
533     return (reinterpret_cast<SPHParticle3D*>(&(mbs->GetElement(sph(i))))->
534         GetDensity());
```

```

529 }
530
531 //add one SPH particle in 2D
532 void FSI_Communication_Element::AddSPH2D(MBS* mbs, const Vector& xg5, double
      radius, double mass, const Vector3D& color){
533     int nr;
534     SPHParticle2D particle2d(mbs, xg5, radius, mass, color);
535     nr = mbs->AddElement(&particle2d);
536     sph.Add(nr);
537 }
538
539 //add one SPH particle in 3D
540 void FSI_Communication_Element::AddSPH3D(MBS* mbs, const Vector& xg7, double
      radius, double mass, const Vector3D& color){
541     int nr;
542     SPHParticle3D particle3d(mbs, xg7, radius, mass, color);
543     nr = mbs->AddElement(&particle3d);
544     sph.Add(nr);
545 }
546
547 //returns the global position of the k-th point of a GeomElement temp
548 Vector2D FSI_Communication_Element::GetGlobPos2D(GeomElement* temp, int k){
549     Vector2D rvtemp;
550     if(temp->GetElnum()==0) //check if element is fixed to the ground - no
      coordinate transformation necessary
551     rvtemp=temp->GetLocPoint2D(k);
552     else
553     rvtemp=temp->GetBody2D().GetPos2D(temp->GetLocPoint2D(k));
554     return rvtemp;
555 }
556
557 //returns the global position of the k-th point of a GeomElement temp
558 Vector3D FSI_Communication_Element::GetGlobPos3D(GeomElement* temp, int k){
559     Vector3D rvtemp;
560     if(temp->GetElnum()==0) //check if element is fixed to the ground - no
      coordinate transformation necessary
561     rvtemp=temp->GetLocPoint(k);
562     else
563     rvtemp=temp->GetBody3D().GetPos(temp->GetLocPoint(k));
564     return rvtemp;
565 }
566
567 //returns the global velocity of the k-th point of a GeomElement temp
568 Vector2D FSI_Communication_Element::GetGlobVel2D(GeomElement* temp, int k){
569     Vector2D rvtemp;

```

```
570     if(temp->GetElnum()==0) //check if element is fixed to the ground - no
        coordinate transformation necessary
571     rvtemp=Vector2D(0.0,0.0);
572     else
573     rvtemp=temp->GetBody2D().GetVel2D(temp->GetLocPoint2D(k));
574     return rvtemp;
575 }
576
577 //returns the global velocity of the k-th point of a GeomElement temp
578 Vector3D FSI_Communication_Element::GetGlobVel3D(GeomElement* temp, int k){
579     Vector3D rvtemp;
580     if(temp->GetElnum()==0) //check if element is fixed to the ground - no
        coordinate transformation necessary
581     rvtemp=Vector3D(0.0,0.0,0.0);
582     else
583     rvtemp=temp->GetBody3D().GetVel(temp->GetLocPoint(k));
584     return rvtemp;
585 }
586
587 //apply the force from boundary point DataH el(i-1,k-1) to corresponding point
        k of GeomElement with number boundary(i)
588 void FSI_Communication_Element::ApplyForce2D(int i, int k){
589     int p1;
590     p1=dataobj.el(i-1,k-1);
591
592     if (i > boundary.Length()) {assert(0 && "ApplyForce::boundary index problem"
        );}
593     if (mbs->NDrawElements() < boundary(i)) {assert(0 && "ApplyForce::boundary i
        larger than ndrawelements");}
594
595     //if(mbs->GetDrawElement(boundary(i))->GetElnum()!=0){ //this may be omitted
        only if it is certain that this function is not called for elements
        fixed to ground / MBS
596     if(1){
597
598     if (loadlist.Length() < k || loadlist.NCols(k) < i) {assert(0 && "
        ApplyForce::loadlist index problem!");}
599     if (mbs->GetDrawElement(boundary(i))->NP() < k) {assert(0 && "ApplyForce::
        locpoints problem");}
600     if (mbs->GetDrawElement(boundary(i))->GetElement().NLoads() < loadlist(k,i)
        ) {assert(0 && "ApplyForce::nloads < loadlist(k,i)");}
601
602     ((MBSLoad&)(mbs->GetDrawElement(boundary(i))->GetElement()).GetLoad(
        loadlist(k,i))).
```

```

603     SetForceVector2D(Vector2D(dataobj.f(p1,0),dataobj.f(p1,1)),mbs->
        GetDrawElement(boundary(i))->GetLocPoint2D(k));
604 }
605 }
606
607 //apply the force from boundary point DataH el(i-1,k-1) to corresponding point
        k of GeomElement with number boundary(i)
608 void FSI_Communication_Element::ApplyForce3D(int i, int k){
609     int p1;
610     p1=dataobj.el(i-1,k-1);
611
612     //if(mbs->GetDrawElement(boundary(i))->GetElnum()!=0){ //this may be omitted
        only if it is certain that this function is not called for elements
        fixed to ground / MBS
613     if(1){
614
615         ((MBSLoad&)(mbs->GetDrawElement(boundary(i))->GetElement()).GetLoad(
            loadlist(k,i))).
616         SetForceVector3D(Vector3D(dataobj.f(p1,0),dataobj.f(p1,1),dataobj.f(p1,2)
            ),mbs->GetDrawElement(boundary(i))->GetLocPoint(k));
617     }
618 }
619
620 //add a GeomLine2D as boundary element
621 void FSI_Communication_Element::AddGeomLine2D(GeomLine2D& line){
622     int nr;
623     MBSLoad load;
624     if(line.GetElnum()==0){
625         nr = mbs->Add(line); //add Geomline to MBS (fixed to ground)
626         boundary_flags.Add(0);
627     }
628     else{
629         const_cast<Element&>(line.GetElement()).Add(line); //add GeomLine connected
            to element line.GetElement() to MBS and add nr to the TArray<int>
            Element::drawelements
630         nr = mbs->NDrawElements();
631         boundary_flags.Add(1);
632     }
633     boundary.Add(nr); //access to the drawelement via mbs->GetDrawElement(nr)
        with nr given by the Tarray boundary
634
635     if(line.GetElnum()!=0){
636         for(int k=1; k<=2; k++){
637             load.SetForceVector2D(Vector2D(0.0,0.0),line.GetLocPoint2D(k));
638             const_cast<Element&>(line.GetElement()).AddLoad(load);

```

```
639     loadlist.Add(k,line.GetElement().NLoads()); //access to load on point k
        of drawelement mbs->GetDrawElement(nr==boundary(i)) acting on
        connected element
640     //mbs->GetDrawElement(nr).GetElement() via mbs->GetDrawElement(boundary(i)
        ).GetElement().GetLoad(loadlist(k,i))
641     }
642 }else{
643     for(int k=1; k<=2; k++){
644         loadlist.Add(k,0); //dummy entry
645     }
646 }
647
648 }
649
650 //add a GeomTrig3D as boundary element
651 void FSI_Communication_Element::AddGeomTrig3D(GeomTrig3D& trig){
652     int nr;
653     MBSLoad load;
654     if(trig.GetElnum()==0){
655         nr = mbs->Add(trig); //add Geomline to MBS (fixed to ground)
656         boundary_flags.Add(0);
657     }
658     else{
659         const_cast<Element&>(trig.GetElement()).Add(trig); //add GeomLine connected
        to element line.GetElement() to MBS and add nr to the TArray<int>
        Element::drawelements
660         nr = mbs->NDrawElements();
661         boundary_flags.Add(1);
662     }
663     boundary.Add(nr); //access to the drawelement via mbs->GetDrawElement(nr)
        with nr given by the TArray boundary
664
665     if(trig.GetElnum()!=0){
666         for(int k=1; k<=3; k++){
667             load.SetForceVector3D(Vector3D(0.0,0.0,0.0),trig.GetLocPoint(k));
668             const_cast<Element&>(trig.GetElement()).AddLoad(load);
669             loadlist.Add(k,trig.GetElement().NLoads()); //access to load on point k
        of drawelement mbs->GetDrawElement(nr==boundary(i)) acting on
        connected element
670             //mbs->GetDrawElement(nr).GetElement() via mbs->GetDrawElement(boundary(i)
        ).GetElement().GetLoad(loadlist(k,i))
671         }
672     }else{
673         for(int k=1; k<=3; k++){
674             loadlist.Add(k,0); //dummy entry
```

```

675     }
676   }
677
678 }
679
680 //functions that use variables defined in the LIGGGHTS input scripts
681 //-----
682 //reset pair_style with actual viscosity
683 void FSI_Communication_Element::ResetVisc(){
684   if(dim == 2)
685     dataobj.sendone("pair_style sph spiky2D ${smoothinglength} artVisc ${alpha}
686                   } 0. ${cAB} ${eta} #tensCorr 0.2");
687   //dataobj.sendone("pair_style sph cubicspline_2D ${smoothinglength} artVisc
688                   ${alphatemp} 0. ${cAB} ${eta} #tensCorr 0.2");
689   //variables used in this commands of course must have been defined
690   previously in the input script or txt
691   else if(dim == 3)
692     //dataobj.sendone("pair_style sph cubicspline ${smoothinglength} artVisc $
693                   {alpha} 0. ${cAB} ${eta} #tensCorr 0.2");
694     dataobj.sendone("pair_style sph spiky ${smoothinglength} artVisc ${alpha}
695                   0. ${cAB} ${eta} #tensCorr 0.2");
696   //variables used in this commands of course must have been defined
697   previously in the input script or txt
698 }
699
700 //-----
701 //functions for particle and boundary creation in the model file
702 //-----
703
704 //NOTE: currently LIGGGHTS-sided only one constant mass is used for all
705 particles - it is calculated as an average over all HotInt-sided defined
706 particle masses!!
707
708 //fill a rectangular region with SPH particles on a regular grid, in xy plane;
709 initial velocity is 0
710 //xll...pos left lower vertex
711 //xru... right upper vertex
712 //h...approximate particle distance / lattice constant (exact if edge lengths
713 are multiples of h)
714 //dens...density, rad...radius, mass...mass, col...color of all particles
715 void FSI_Communication_Element::FillRectangle(Vector2D xll, Vector2D xru,
716 double h, MBS* mbs, double dens, double rad, double mass, const Vector3D
717 col){
718   int nx=int(floor(fabs(xll.X()-xru.X())/h));

```

```
708   int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
709   double dx= fabs(xll.X()-xru.X())/double(nx);
710   double dy= fabs(xll.Y()-xru.Y())/double(ny);
711   if(nx==0) dx = fabs(xll.X()-xru.X());
712   if(ny==0) dy = fabs(xll.Y()-xru.Y());
713   for(int i=0; i<=nx; ++i)
714     for(int j=0; j<=ny; ++j)
715       AddSPH2D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,0.0,0.0,dens),rad,mass,col)
       ;
716   mbs->UO()<< (nx+1)*(ny+1) << " SPH particles created \n";
717 }
718
719 //fill a rectangular region with SPH paricles on a regular grid, in xy plane;
       initial velocity is 0; sets density to mass/(area per particle)
720 //(the appropriate value of the chosen configuration in 2D - note that mass
       in 2D is mass per depth unit)
721 //xll...pos left lower vertex
722 //xru... right upper vertex
723 //h...approximate particle distance / lattice constant (exact if edge lengths
       are multiples of h)
724 //rad...radius, mass...mass, col...color of all particles
725 void FSI_Communication_Element::FillRectangleAutoDensity(Vector2D xll,
       Vector2D xru, double h, MBS* mbs, double rad, double mass, const Vector3D
       col){
726   int nx=int(floor(fabs(xll.X()-xru.X())/h));
727   int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
728   double dx= fabs(xll.X()-xru.X())/double(nx);
729   double dy= fabs(xll.Y()-xru.Y())/double(ny);
730   if(nx==0) dx = fabs(xll.X()-xru.X());
731   if(ny==0) dy = fabs(xll.Y()-xru.Y());
732   double dens = mass/(dx*dy);
733   for(int i=0; i<=nx; ++i)
734     for(int j=0; j<=ny; ++j)
735       AddSPH2D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,0.0,0.0,dens),rad,mass,col)
       ;
736   mbs->UO()<< (nx+1)*(ny+1) << " SPH particles created \n";
737 }
738
739 //fill a rectangular region with SPH paricles on a regular grid, in xy plane;
       initial velocity is 0; sets mass (i.e. a mass per depth unit in 2D) of
       particles to dens*(area per particle)
740 //currently: LIGGGHTS-sided only one constant mass is used for all particles -
       it is calculated as an average over all HotInt-sided defined particle
       masses
741 //xll...pos left lower vertex
```

```

742 //xru... right upper vertex
743 //h...approximate particle distance / lattice constant (exact if edge lengths
      are multiples of h)
744 //dens...density, rad...radius, mass...mass, col...color of all particles
745 void FSI_Communication_Element::FillRectangleAutoMass(Vector2D xll, Vector2D
      xru, double h, MBS* mbs, double dens, double rad, const Vector3D col){
746     int nx=int(floor(fabs(xll.X()-xru.X())/h));
747     int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
748     double dx= fabs(xll.X()-xru.X())/double(nx);
749     double dy= fabs(xll.Y()-xru.Y())/double(ny);
750     if(nx==0) dx = fabs(xll.X()-xru.X());
751     if(ny==0) dy = fabs(xll.Y()-xru.Y());
752     double mass = dens*dx*dy;
753     for(int i=0; i<=nx; ++i)
754         for(int j=0; j<=ny; ++j)
755             AddSPH2D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,0.0,0.0,dens),rad,mass,col)
              ;
756     mbs->UO()<< (nx+1)*(ny+1) << " SPH particles created \n";
757 }
758
759 //fill a rectangular region with SPH particles on a regular grid, in xy plane;
760 //initial velocity is uniformly distributed over [-0.5*v, 0.5*v] in every
      component;
761 //mass (i.e. a mass per depth unit in 2D) of particles is set to dens*(area
      per particle)
762 //currently: LIGGGHTS-sided only one constant mass is used for all particles -
      it is calculated as an average over all HotInt-sided defined particle
      masses
763 //xll...pos left lower vertex
764 //xru... right upper vertex
765 //h...approximate particle distance / lattice constant (exact if edge lengths
      are multiples of h)
766 //dens...density, rad...radius, mass...mass, col...color of all particles
767 void FSI_Communication_Element::FillRectangleAutoMass(Vector2D xll, Vector2D
      xru, double h, MBS* mbs, double dens, double rad, const Vector3D col,
      double v){
768     int nx=int(floor(fabs(xll.X()-xru.X())/h));
769     int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
770     double dx= fabs(xll.X()-xru.X())/double(nx);
771     double dy= fabs(xll.Y()-xru.Y())/double(ny);
772     if(nx==0) dx = fabs(xll.X()-xru.X());
773     if(ny==0) dy = fabs(xll.Y()-xru.Y());
774     double mass = dens*dx*dy;
775     srand(time(NULL));
776     double rm = double(RAND_MAX);

```

```
777     for(int i=0; i<=nx; ++i)
778         for(int j=0; j<=ny; ++j)
779             AddSPH2D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,v*(rand()/rm-0.5),v*(rand()
                /rm-0.5),dens),rad,mass,col);
780     mbs->UO()<< (nx+1)*(ny+1) << " SPH particles created \n";
781 }
782
783 //fill a box with SPH particles 3D on a regular grid, base face lies parallel
        to the xy plane; initial velocity is 0
784 //xll...pos left lower vertex of (rectangular) base
785 //xru... right upper vertex of (rectangular) top face
786 //h...approximate particle distance / lattice constant (exact if edge lengths
        are multiples of h)
787 //dens...density, rad...radius, mass...mass, col...color of all particles
788 void FSI_Communication_Element::FillBox(Vector3D xll, Vector3D xru, double h,
        MBS* mbs, double dens, double rad, double mass, const Vector3D col){
789     int nx=int(floor(fabs(xll.X()-xru.X())/h));
790     int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
791     int nz=int(floor(fabs(xll.Z()-xru.Z())/h));
792     double dx= fabs(xll.X()-xru.X())/double(nx);
793     double dy= fabs(xll.Y()-xru.Y())/double(ny);
794     double dz= fabs(xll.Z()-xru.Z())/double(nz);
795     if(nx==0) dx = fabs(xll.X()-xru.X());
796     if(ny==0) dy = fabs(xll.Y()-xru.Y());
797     if(nz==0) dz = fabs(xll.Z()-xru.Z());
798     for(int i=0; i<=nx; ++i)
799         for(int j=0; j<=ny; ++j)
800             for(int k=0; k<=nz; ++k)
801                 AddSPH3D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,xll.Z()+k*dz,0.0,0.0,0.0,
                    dens),rad,mass,col);
802     mbs->UO()<< (nx+1)*(ny+1)*(nz+1) << " SPH particles created \n";
803 }
804
805 //fill a box with SPH particles 3D on a regular grid, base face lies parallel
        to the xy plane; initial velocity is 0; density is set to mass/(volume per
        particle)
806 //xll...pos left lower vertex of (rectangular) base
807 //xru... right upper vertex of (rectangular) top face
808 //h...approximate particle distance / lattice constant (exact if edge lengths
        are multiples of h)
809 //rad...radius, mass...mass, col...color of all particles
810 void FSI_Communication_Element::FillBoxAutoDensity(Vector3D xll, Vector3D xru,
        double h, MBS* mbs, double rad, double mass, const Vector3D col){
811     int nx=int(floor(fabs(xll.X()-xru.X())/h));
812     int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
```

```

813  int nz=int(floor(fabs(xll.Z()-xru.Z())/h));
814  double dx= fabs(xll.X()-xru.X())/double(nx);
815  double dy= fabs(xll.Y()-xru.Y())/double(ny);
816  double dz= fabs(xll.Z()-xru.Z())/double(nz);
817  if(nx==0) dx = fabs(xll.X()-xru.X());
818  if(ny==0) dy = fabs(xll.Y()-xru.Y());
819  if(nz==0) dy = fabs(xll.Z()-xru.Z());
820  double dens = mass/(dx*dy*dz);
821  for(int i=0; i<=nx; ++i)
822    for(int j=0; j<=ny; ++j)
823      for(int k=0; k<=nz; ++k)
824        AddSPH3D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,xll.Z()+k*dz,0.0,0.0,0.0,
            dens),rad,mass,col);
825  mbs->UO()<< (nx+1)*(ny+1)*(nz+1) << " SPH particles created \n";
826  }
827
828  //fill a box with SPH particles 3D on a regular grid, base face lies parallel
      to the xy plane; initial velocity is 0; particle mass is set to dens*(
      volume per particle)
829  //currently: LIGGGHTS-sided only one constant mass is used for all particles -
      it is calculated as an average over all HotInt-sided defined particle
      masses
830  //xll...pos left lower vertex of (rectangular) base
831  //xru... right upper vertex of (rectangular) top face
832  //h...approximate particle distance / lattice constant (exact if edge lengths
      are multiples of h)
833  //dens...density, rad...radius, mass...mass, col...color of all particles
834  void FSI_Communication_Element::FillBoxAutoMass(Vector3D xll, Vector3D xru,
      double h, MBS* mbs, double dens, double rad, const Vector3D col){
835  int nx=int(floor(fabs(xll.X()-xru.X())/h));
836  int ny=int(floor(fabs(xll.Y()-xru.Y())/h));
837  int nz=int(floor(fabs(xll.Z()-xru.Z())/h));
838  double dx= fabs(xll.X()-xru.X())/double(nx);
839  double dy= fabs(xll.Y()-xru.Y())/double(ny);
840  double dz= fabs(xll.Z()-xru.Z())/double(nz);
841  double mass = dens*dx*dy*dz;
842  for(int i=0; i<=nx; ++i)
843    for(int j=0; j<=ny; ++j)
844      for(int k=0; k<=nz; ++k)
845        AddSPH3D(mbs,Vector(xll.X()+i*dx,xll.Y()+j*dy,xll.Z()+k*dz,0.0,0.0,0.0,
            dens),rad,mass,col);
846  mbs->UO()<< (nx+1)*(ny+1)*(nz+1) << " SPH particles created \n";
847  }
848

```

```
849 //create a 2D boundary shape by adjacent line segments between points (p1,p2),
      (p2,p3), ... (pn-1,pn)
850 //where {p1,...,pn} is given by "points"
851 //number is the element number, to which the GeomElements are connected, width
      is line width, col...color
852 void FSI_Communication_Element::CreateBoundary2D(MBS* mbs, int number, TArray<
      Vector2D> points, double wi, Vector3D col){
853     GeomLine2D line2;
854     for(int i=1; i<points.Length(); ++i){
855         line2=GeomLine2D(mbs,number,points(i),points(i+1),col);
856         line2.SetDrawParam(Vector3D(wi, 10., 0.));
857         AddGeomLine2D(line2);
858     }
859
860 }
861
862 //import and add a stl-geometry defined in stl_file_name to an element elemnr
      as GeomTrig3Ds and use AlternativeShape for this element
863 //GeomTrig3Ds are registered as surface elements for FSI in
      fsi_communication_element, unless the flag AddToCommunicationElement is
      set to 0 (default is 1)
864 void FSI_Communication_Element::AddSTLMeshTrigsToElement(MBS* mbs, int elemnr,
      mystr stl_file_name, double stretch_factor, const Vector3D&
      translation_vec, const Matrix3D& rotation_mat, const Vector3D& color,
      double transparency, int AddToCommunicationElement)
865 {
866     GeomMesh3D mesh;
867     mesh.ReadSTLMesh(stl_file_name.c_str());
868     mesh.Stretch(stretch_factor);
869     mesh.Translate(translation_vec);
870     mesh.Rotation(rotation_mat);
871
872     int counter = 0;
873     for(int i=1; i <= mesh.NTrigs(); i++)
874     {
875         Vector3D p1, p2, p3, n;
876         mesh.GetTrig0(i, p1, p2, p3, n);
877
878         if ((p2-p1).Cross(p3-p2)*n < 0)
879         {
880             swap(p2,p3);
881             counter++;
882         }
883
884         //     GeomTrig3D trig(mbs, 0, p1, p2, p3, color); //elemnr must be passed
```

```

885     GeomTrig3D trig(mbs, elemnr, p1, p2, p3, color);
886     trig.SetTransparency(transparency);
887
888     if (elemnr)
889     {
890         Element * elem = &mbs->GetElement(elemnr);
891         elem->SetAltShape(1);
892         //elem->Add(trig);
893     }
894     //else
895     //{
896     // mbs->Add(trig);
897     //}
898     if(AddToCommunicationElement) AddGeomTrig3D(trig);
899 }
900 mbs->UO(UO_LVL_0) << "AddSTLMeshToElement for \' " << stl_file_name << "\':
      swapped " << counter << " of " << mesh.NTrigs() << " trigs.\n";
901 }
902
903 //add a random offset uniformly distributed over [-0.5*x,0.5*x] to all
      particle coordinates
904 void FSI_Communication_Element::randomize_particle_positions(double x){
905     srand(time(NULL));
906     double rm = double(RAND_MAX);
907     if(dim == 2){
908         Vector2D pos;
909         for(int i=1; i<=sph.Length(); ++i){
910             pos = GetParticlePos2D(i);
911             pos += Vector2D(x*(rand()/rm-0.5),x*(rand()/rm-0.5));
912             SetParticlePos2D(i,pos);
913         }
914     }
915     else if(dim == 3){
916         Vector3D pos;
917         for(int i=1; i<=sph.Length(); ++i){
918             pos = GetParticlePos3D(i);
919             pos += Vector3D(x*(rand()/rm-0.5),x*(rand()/rm-0.5),x*(rand()/rm-0.5));
920             SetParticlePos3D(i,pos);
921         }
922     }
923     else{
924         mbs->UO(UO_LVL_warn) << "warning: FSI_Communication_Element is implemented
      for 2D-case or 3D-case only!\n";
925     }
926 }

```

```
927
928 //-----
929
930 void FSI_Communication_Element::CopyFrom(const Element& e)
931 {
932     Element::CopyFrom(e);
933     const FSI_Communication_Element& ce = (const FSI_Communication_Element&)e;
934
935     // copy members
936     step = ce.step;
937     counter = ce.counter;
938     dim = ce.dim;
939     nparticles = ce.nparticles;
940     nequi = ce.nequi;
941
942     sph = ce.sph;
943     dataobj = ce.dataobj;
944     isinitialized=ce.isinitialized;
945     loadlist=ce.loadlist;
946     boundary=ce.boundary;
947     boundary_flags=ce.boundary_flags;
948
949     inputscript = ce.inputscript;
950     edc = ce.edc;
951     nstat = ce.nstat;
952
953     deltax = ce.deltax;
954 }
955
956 Element* FSI_Communication_Element::GetCopy()
957 {
958     Element* ec = new FSI_Communication_Element(*this);
959     return ec;
960 }
961
962 void FSI_Communication_Element::Initialize()
963 {
964     if(!isinitialized){
965         step = 0;
966         InitializeDataCommunication();
967     }
968 }
969
970 void FSI_Communication_Element::StartTimeStep()
971 {
```

```
972
973 double some_termination_criterion;
974 try
975 {
976     some_termination_criterion = OutgoingDataCommunication();
977 }
978 catch(mystr & message)
979 {
980     mbs->UO(UO_LVL_err) << message; // mbs->UO() creates output in output
           window at runtime
981 }
982
983 //mbs->UO(UO_LVL_sim) << "starting step " << ++step << "\n";
984 IncomingDataCommunication(step);
985 counter = 0;
986 }
987
988 void FSI_Communication_Element::EndTimeStep()
989 {
990     //mbs->UO() << "time = " << mbs->GetTime() << ", stepszie = " << mbs->
           GetStepSize() << ", end time = " << mbs->GetSolSet().endtime << "\n";
991     if(fabs(mbs->GetTime()-mbs->GetSolSet().endtime)<=1E-2*mbs->GetSolSet().
           minstepsize)
992         Finalize();
993
994 }
995
996 double FSI_Communication_Element::PostNewtonStep(double t)
997 {
998     //mbs->Get_pCFB()->ResultsUpdated(1); // redraw only, no data saving
999
1000     /* double some_termination_criterion;
1001     try
1002     {
1003         some_termination_criterion = OutgoingDataCommunication();
1004     }
1005     catch(mystr & message)
1006     {
1007         mbs->UO(UO_LVL_err) << message; // mbs->UO() creates output in output window
           at runtime
1008     return 0;
1009     }
1010     mbs->UO(UO_LVL_sim) << "iteration number " << ++counter << "; termination
           criterion: " << some_termination_criterion << "\n";
1011     */
```

```
1012 // due to some reasons there might be problems when only one iteration is
      performed
1013 // even if some_termination_criterion is small enough, and therefore we
      artificially
1014 // could perform the following test:
1015 //if(counter < 2)
1016 // return 1;
1017
1018 //return fabs(some_termination_criterion);
1019 return 0.;
1020
1021 }
```

## A.8. exchange\_class\_linux.h

```
1 //parallelized with MPI (has to run somewhere inbetween MPI_Init(NULL,NULL) (e
      .g.) and MPI_Finalize()
2 //only proc 0 does setup, communication and data exhchange
3
4 #include <vector>
5 #include "lammgs.h"
6 //#include "mpi.h"
7 #include "interface_baseclass.h"
8
9 #ifndef EX_CLASS_LINUX
10 #define EX_CLASS_LINUX
11
12 namespace LAMMPS_NS{
13   class LAMMPS; //forward declaration
14 }
15
16 using std::vector;
17 using std::cout;
18 using std::endl;
19
20 //container for refinement data (subtriangles, defined in barycentric
      coordinates w.r.t. original triangle) of an original triangle
21 class RefinedTriangle{
22 protected:
23   vector< vector< vector< double > > > b; //b[i][j][k] contains k-th
      barycentric coordinate of j-th vertex of i-th subtriangle
24 public:
25   RefinedTriangle(){ }
26   ~RefinedTriangle(){ }
```

```

27 void Add(double a1,double a2,double a3,double b1,double b2,double b3,double
    c1,double c2,double c3, int i){
28     vector< double > temp0,temp1,temp2;
29     temp0.push_back(a1); temp0.push_back(a2); temp0.push_back(a3); //vector
        containing barycentric coordinates of vertex 0
30     temp1.push_back(b1); temp1.push_back(b2); temp1.push_back(b3);
31     temp2.push_back(c1); temp2.push_back(c2); temp2.push_back(c3);
32     //std::cout << "created vertices of sub-triangle" << std::endl;
33
34     vector< vector< double > > vtemp; //vector containing coordinate tripels of
        all vertices of new subtriangle
35     vtemp.push_back(temp0); vtemp.push_back(temp1); vtemp.push_back(temp2);
36
37     //std::cout << "created sub-triangle: " << std::endl;
38     //std::cout << "(" << vtemp[0][0] << "," << vtemp[0][1] << "," << vtemp
        [0][2] << ")" << std::endl;
39     //std::cout << "(" << vtemp[1][0] << "," << vtemp[1][1] << "," << vtemp
        [1][2] << ")" << std::endl;
40     //std::cout << "(" << vtemp[2][0] << "," << vtemp[2][1] << "," << vtemp
        [2][2] << ")" << std::endl;
41
42     //vector< vector< vector< double > > > t;
43     ////cout << "declared local copy" << endl;
44     //t.push_back(vtemp);
45     //std::cout << "local copy successful" << std::endl;
46
47     b.push_back(vtemp); //add new subtriangle to b
48
49     //std::cout << b.size();
50     //b.push_back(vector< vector< double > >());
51     //std::cout << "added sub-triangle to list: " << std::endl;
52     //int size = b.size();
53     //std::cout << "(" << b[size-1][0][0] << "," << b[size-1][0][1] << "," << b
        [size-1][0][2] << ")" << std::endl;
54     //std::cout << "(" << b[size-1][1][0] << "," << b[size-1][1][1] << "," << b
        [size-1][1][2] << ")" << std::endl;
55     //std::cout << "(" << b[size-1][2][0] << "," << b[size-1][2][1] << "," << b
        [size-1][2][2] << ")" << std::endl;
56
57 }
58 double Get(int i,int j,int k){ return b[i][j][k]; } //return k-th
        barycentric coordinate of j-th vertex of i-th subtriangle
59 int Length(){return b.size();}
60 };
61

```

```
62 class DataL: public interface_baseclass {
63
64 public:
65     int proc; //number of current process
66     int nprocs; //total number of processes
67
68     DataL(const std::string* _ip = NULL, const short* _port = NULL, int narg=0,
           char **arg=NULL); //counterpiece to DataH constructor (see DataH)
69     /*DataL(const DataL&); //copy constructor
70     DataL& operator=(const DataL&); //copy assignment*/
71     ~DataL();
72
73     int readone(); // reads in one LIGGGHTS input script line; returns 1 for
           successful read, 0 if termination string "qqq" was received
74     int recv_command(); //receives one command and executes corresponding
           routines; returns 1 for known command, 0 for unknown command
75
76     void sendforce();
77     void sendrSPH();
78     void sendvSPH();
79     void getrSPH();
80     void getvSPH();
81     void getrv();
82     void getrvfull();
83     void set_timestep(); //used to receive and set LIGGGHTS timestep -- has to
           be done before first "run" command
84     void sendrohSPH();
85     void getrohSPH();
86     void get_ref_opt(); //receive refinement_option; cf. interface_baseclass::
           refinement_option
87     void get_ref_res(); //receive refinement resolution - cf.
           interface_baseclass::dr
88
89     //functions for 3D mesh refinement
90     //counting/indices start from 0 in any case
91
92     //add one subtriangle ABC with barycentric coordinates a1,a2,a3,b1,b2,b3,c1,
           c2,c3 to list of subtriangles ref_triangle[i] for original triangle i
93     void AddSubTriangle(int i, double a1, double a2,double a3,double b1,double
           b2,double b3,double c1,double c2,double c3) {ref_triangle[i].Add(a1,a2,a3
           ,b1,b2,b3,c1,c2,c3,i);}
94     //get m-th barycentric coordinate (m=0...2) of k-th vertex (k=0..2) of j-th
           subtriangle of i-th original triangle
95     double GetBC(int i,int j,int k,int m) {return ref_triangle[i].Get(j,k,m);}

```

```

96 //get m-th cartesian coordinate (m=0...2) of k-th vertex (k=0..2) of j-th
    subtriangle of i-th original triangle
97 double GetCC(int i,int j,int k,int m) {return GetBC(i,j,k,0)*r(el(i,0),m)+
    GetBC(i,j,k,1)*r(el(i,1),m)+GetBC(i,j,k,2)*r(el(i,2),m);}
98 //get number of sub-triangles corresponding to original triangle i
99 int nsub(int i){return ref_triangle[i].Length();}
100
101 //mesh refinement in case of refinement_option == 2 || 3 and dim == 3
102 void refine_mesh();
103 void recursive_refine(int i, const double* r1, const double* r2, const
    double* r3, double a1,double a2,double a3,double b1,double b2,double b3,
    double c1,double c2,double c3);
104
105 protected:
106 int c; //c...client-socket
107 LAMMPS_NS::LAMMPS* lp; //pointer to LAMMPS object, which is initialized and
    set up in the constructor
108 vector<RefinedTriangle> ref_triangle; //contains precalculated mesh
    refinement -- see notes; ref_triangle[i] contains refinement of surface
    triangle el(i) in terms of barycentric coordinates
109 //w.r.t. the original vertices;
110
111 };
112
113 #endif

```

## A.9. exchange\_class\_linux.cpp

```

1 #include <string>
2
3 // LAMMPS include files
4 #include "lammps.h"
5 #include "input.h"
6 #include "atom.h"
7 #include "update.h"
8 #include "force.h"
9 #include "modify.h"
10 #include "integrate.h"
11 #include "pair.h"
12 #include "fix.h"
13 //
14
15 #include <stdlib.h>
16 #include <unistd.h>
17 #include <errno.h>

```

```
18 #include <string.h>
19 #include <netdb.h>
20 #include <sys/types.h>
21 #include <netinet/in.h>
22 #include <sys/socket.h>
23 #include <arpa/inet.h> // for htons etc
24 #include <iostream>
25 #include <iomanip>
26 #include <cstring>
27 #include <math.h>
28 #include <iostream>
29
30 #include "dn.h"
31 #include "exchange_class_linux.h"
32 #include "interface_baseclass.h"
33
34 #include "mpi.h"
35
36 #include <vector>
37
38 using namespace LAMMPS_NS;
39 using namespace std;
40
41 DataL::DataL(const string* _ip, const short* _port, int nargs, char **arg){
42
43     dt=0.0; //actual timestep has to be set using set_timestep(double)
44
45     //TCP-IP v4 server setup and initialization of IP, port, nSPH
46     //-----
47     MPI_Comm_rank(MPI_COMM_WORLD,&proc);
48     MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
49
50     string a1,a2,a3,a4,temp1;
51     int stat=-1;
52     struct sockaddr_in addr;
53     short port;
54
55     if(proc == 0){
56
57         if(!_ip && !_port){
58
59             /*
60             cout << "Server-IP a1.a2.a3.a4 (IPv4): " << endl << "a1: " << endl;
61             cin >> a1;
62             cout << "a2: " << endl;
```

```

63     cin >> a2;
64     cout << "a3: " << endl;
65     cin >> a3;
66     cout << "a4: " << endl;
67     cin >> a4;
68     cout << "Port: " << endl;
69     cin >> port;
70     */
71     a1="192"; a2="168"; a3="56"; a4="1"; port=12345;
72
73     temp1.append(a1).append(".").append(a2).append(".").append(a3).append("."
74         ).append(a4);
75     }
76     else if(_ip && _port){
77         temp1=*_ip;
78         port=*_port;
79     }
80     //create socket
81     c=socket(AF_INET, SOCK_STREAM, 0);
82     //ERROR CHECKING
83     if(c==-1){
84         cout << "Error: socket, error number: " << strerror(errno) << endl;
85         MPI_Abort(MPI_COMM_WORLD,1);
86     }
87     else{
88         cout << "Socket erfolgreich erstellt" << endl;
89     }
90
91     //configure connection and connect socket
92
93     //memset(addr.sin_zero,'0',sizeof(addr.sin_zero)); //Nullsetzen des
94         Elements sin_zero nicht unbedingt notwendig
95     addr.sin_family=AF_INET; //use IPv4
96     addr.sin_port=htons(port);
97     addr.sin_addr.s_addr=inet_addr(temp1.c_str());
98 }
99
100 int waitingcounter = -1;
101 while(1){ //this waiting procedure has to be synchronized
102     if(proc == 0)
103         stat=connect(c,reinterpret_cast<struct sockaddr *>(&addr),sizeof(addr));
104     MPI_Bcast(&stat,1,MPI_INT,0,MPI_COMM_WORLD);
105     if(stat!=-1){
106         if(proc == 0)

```

```
106     cout << endl << "Verbindung mit " << temp1.c_str() << ", Port " << port
      << " erfolgreich" << endl;
107     break;
108 }else{
109     //cout << "Error: connect" << endl;
110     //MPI_Abort(MPI_COMM_WORLD,1);
111     if(proc == 0){
112         switch(waitingcounter){
113             case -1: cout << "Waiting to connect to " << temp1.c_str() << ", port
                    " << port << endl; break;
114             case 0: cout << "\r." << " " << flush; break;
115             case 1: cout << "\r.." << " " << flush; break;
116             case 2: cout << "\r..." << " " << flush; waitingcounter = -1; break;
117         }
118         ++waitingcounter;
119     }
120     sleep(1); //wait for 1 second
121 }
122 }
123
124 //create LAMMPS/LIGGGHTS object on each proc
125 //-----
126 lp = new LAMMPS(narg,arg,MPI_COMM_WORLD,this);
127
128 //data transfer and initialization of LIGGGHTS and rSPH
129 //-----
130 while(readone()); // read input script lines (part 1, variable definitions)
131
132 if(proc == 0){
133     recv(c,reinterpret_cast<char *>(&nSPH),4,0);
134     nSPH=ntohl(nSPH);
135 }
136 MPI_Bcast(&nSPH,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
137
138 while(readone()); // read input script lines (part 2)
139
140 //memory allocation and initialization of e1 and nel, r,v,f
141 //-----
142 if(proc == 0){
143     recv(c,reinterpret_cast<char *>(&n),4,0);
144     n=ntohl(n);
145     recv(c,reinterpret_cast<char *>(&nstat),4,0);
146     nstat=ntohl(nstat);
147     recv(c,reinterpret_cast<char *>(&dim),2,0);
148     dim=ntohs(dim);
```

```

149 }
150 MPI_Bcast(&n,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
151 MPI_Bcast(&nstat,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
152 MPI_Bcast(&dim,1,MPI_UNSIGNED_SHORT,0,MPI_COMM_WORLD);
153
154 _r = new double[3*n];
155 _v = new double[3*n];
156 _f = new double[3*n];
157
158 zerof();
159 zeror();
160 zerov();
161
162 getrvfull(); //initialize r and v
163
164 //create and initialize nel and el -- only has to be done once
165 if(proc == 0){
166     recv(c,reinterpret_cast<char *>(&nel),4,0);
167     nel=ntohl(nel);
168 }
169 MPI_Bcast(&nel,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
170
171 _el = new unsigned int[dim*nel];
172
173 if(proc==0){
174     for(int i=0; i<nel; ++i){
175         for(int j=0; j<dim; ++j){
176             recv(c,reinterpret_cast<char *>(elp(i,j)),4,0);
177             setel(i,j,ntohl(el(i,j)));
178         }
179     }
180 }
181
182 MPI_Bcast(_el,dim*nel,MPI_UNSIGNED,0,MPI_COMM_WORLD);
183
184 //get refinement option and calculate pre-refined mesh (for 3D); initialize
185     vector<RefinedTriangle> ref_triangle
186
187 ref_triangle.resize(nel,RefinedTriangle());
188
189 get_ref_opt();
190 //std::cout << "proc " << proc << ": " << "refinement option: " <<
191     get_refinement_option() << std::endl;
192 if(proc==0) std::cout << "refinement option: " << get_refinement_option() <<
193     std::endl;

```

```
191
192   get_ref_res();
193   //std::cout << "proc " << proc << ": " << "refinement resolution: " <<
       get_dr() << std::endl;
194   if(proc==0) std::cout << "refinement resolution: " << get_dr() << std::endl;
195
196   if(dim==3){
197       if(get_refinement_option()==2 || get_refinement_option()==3){ //in this
           case, calculate pre-refined mesh
198           refine_mesh(); //done on all procs
199       }
200   }
201
202   _rSPH = new double[3*nSPH];
203   _vSPH = new double[3*nSPH];
204   getrSPH();
205   getvSPH();
206
207   _rohSPH = new double[nSPH];
208   zerorohSPH();
209
210 }
211
212 void DataL::set_timestep(){
213
214   if(proc==0){
215       char temp[10];
216       int remain=10;
217       while(remain!=0) remain--recv(c,temp+(10-remain),remain,0);
218       ntohs(temp, &dt, 1);
219   }
220   MPI_Bcast(&dt,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
221
222   //the following is done in analogy to what is done in the LIGGGHTS fix
       fix_dt_reset.cpp (in function end_of_step())
223
224   // reset update->dt and other classes that depend on it
225   // rRESPA, pair style, fixes
226   lp->update->dt = dt;
227   if (strcmp(lp->update->integrate_style,"respa") == 0) lp->update->integrate
       ->reset_dt();
228   if (lp->force->pair) lp->force->pair->reset_dt();
229   for (int i = 0; i < lp->modify->nfix; i++) lp->modify->fix[i]->reset_dt();
230 }
231
```

```

232 void DataL::sendforce(){
233     double* ftemp;
234     if(proc == 0){
235         ftemp = new double[3*(n-nstat)];
236         for(int i=0; i<3*(n-nstat); ++i) ftemp[i]=0.0;
237     }
238     MPI_Reduce(_f,ftemp,3*(n-nstat),MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
239     //now ftemp is filled and updated
240
241     if(proc == 0){
242         char* mem = new char[30*(n-nstat)]; //memory which ntohd and htdon is
                working on; size = 10*(number of doubles in array)
243
244         //with consistency check
245         int remain=30*(n-nstat);
246         htond(mem,ftemp,3*(n-nstat));
247         while(remain!=0) remain-=send(c,mem+(30*(n-nstat)-remain),remain,0);
248
249         delete [] mem;
250         delete [] ftemp;
251     }
252 }
253
254 void DataL::sendrSPH(){
255     //get coordinates from LIGGGHTS
256     zerorSPH();
257     double **x = lp->atom->x;
258     int *tag = lp->atom->tag;
259     int nlocal = lp->atom->nlocal;
260     int id,offset;
261     for (int i = 0; i < nlocal; i++) {
262         id = tag[i]; //global number of local particle i, range 1...nSPH --> id-1
                actual global number
263         for(int j=0; j<3; ++j) setrSPH((id-1),j,x[i][j]);
264     }
265
266     double* rtemp;
267     if(proc == 0){
268         rtemp = new double[3*nSPH];
269         for(int i=0; i<3*nSPH; ++i) rtemp[i] = 0.0;
270     }
271
272     MPI_Reduce(_rSPH,rtemp,3*nSPH,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
273     //now rtemp is filled and updated on proc0
274

```

```
275  if(proc == 0){
276    char* mem = new char[30*nSPH]; //memory which ntohs and htons is working on
        ; size = 10*(number of doubles in array)
277    //with consistency check
278    int remain=30*nSPH;
279    htons(mem,rtemp,3*nSPH);
280    while(remain!=0) remain-=send(c,mem+(30*nSPH-remain),remain,0);
281
282    delete [] mem;
283    delete [] rtemp;
284  }
285 }
286
287 void DataL::sendvSPH(){
288   //get coordinates from LIGGGHTS
289   zerovSPH();
290   double **v = lp->atom->v;
291   int *tag = lp->atom->tag;
292   int nlocal = lp->atom->nlocal;
293   int id,offset;
294   for (int i = 0; i < nlocal; i++) {
295     id = tag[i]; //global number of local particle i, range 1...nSPH --> id-1
        actual global number
296     for(int j=0; j<3; ++j) setvSPH((id-1),j,v[i][j]);
297   }
298
299   double* vtemp;
300   if(proc == 0){
301     vtemp = new double[3*nSPH];
302     for(int i=0; i<3*nSPH; ++i) vtemp[i] = 0.0;
303   }
304
305   MPI_Reduce(_vSPH,vtemp,3*nSPH,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
306   //now rtemp is filled and updated on proc0
307
308   if(proc == 0){
309     char* mem = new char[30*nSPH]; //memory which ntohs and htons is working on
        ; size = 10*(number of doubles in array)
310     //with consistency check
311     int remain=30*nSPH;
312     htons(mem,vtemp,3*nSPH);
313     while(remain!=0) remain-=send(c,mem+(30*nSPH-remain),remain,0);
314
315     delete [] mem;
316     delete [] vtemp;
```

```
317 }
318 }
319
320 void DataL::sendrohSPH(){
321     //get densities from LIGGGHTS
322     zerorohSPH();
323     double *density = lp->atom->density;
324     int *tag = lp->atom->tag;
325     int nlocal = lp->atom->nlocal;
326     int id,offset;
327     for (int i = 0; i < nlocal; i++) {
328         id = tag[i]; //global number of local particle i, range 1..nSPH --> id-1
329         //actual global number
330         setrohSPH((id-1),density[i]);
331     }
332
333     double* rohtemp;
334     if(proc == 0){
335         rohtemp = new double[nSPH];
336         for(int i=0; i<nSPH; ++i) rohtemp[i] = 0.0;
337     }
338     MPI_Reduce(_rohSPH,rohtemp,nSPH,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
339     //now rohtemp is filled and updated on proc0
340
341     if(proc == 0){
342         char* mem = new char[10*nSPH]; //memory which ntohs and htons is working on
343         ; size = 10*(number of doubles in array)
344         //with consistency check
345         int remain=10*nSPH;
346         htons(mem,rohtemp,nSPH);
347         while(remain!=0) remain-=send(c,mem+(10*nSPH-remain),remain,0);
348
349         delete [] mem;
350         delete [] rohtemp;
351     }
352
353 void DataL::getrSPH(){
354
355     if(proc == 0){
356         char* mem = new char[30*nSPH]; //memory which ntohs and htons is working on
357         ; size = 10*(number of doubles in array)
358
359         //with consistency check
```

```
359     int remain=30*nSPH;
360     while(remain!=0) remain-=recv(c,mem+(30*nSPH-remain),remain,0);
361     ntohd(mem,_rSPH,3*nSPH);
362
363     delete [] mem;
364 }
365 MPI_Bcast(_rSPH,3*nSPH,MPI_DOUBLE,0,MPI_COMM_WORLD);
366
367 //rSPH initialization in LIGGGHTS atom array
368 double** x = lp->atom->x;
369 int m;
370 for(int i=0; i<nSPH; ++i){
371     if((m = lp->atom->map(i+1))>=0){ //map(i) returns local number of particle
372         with global number i (or <0 if i not one of local particles)
373         for(int j=0; j<3; ++j){
374             x[m][j]=rSPH(i,j);
375         }
376         // cout << "Teilchen " << i << " initialisiert mit rSPH = " << rSPH(i
377         ,0) << " " << rSPH(i,1) << " " << rSPH(i,2) << endl;
378     }
379 }
380
381 void DataL::getvSPH(){
382
383     if(proc == 0){
384         char* mem = new char[30*nSPH]; //memory which ntohd and htond is working on
385         ; size = 10*(number of doubles in array)
386
387         //with consistency check
388         int remain=30*nSPH;
389         while(remain!=0) remain-=recv(c,mem+(30*nSPH-remain),remain,0);
390         ntohd(mem,_vSPH,3*nSPH);
391
392         delete [] mem;
393     }
394     MPI_Bcast(_vSPH,3*nSPH,MPI_DOUBLE,0,MPI_COMM_WORLD);
395
396     //vSPH initialization in LIGGGHTS atom array
397     double** v = lp->atom->v;
398     int m;
399     for(int i=0; i<nSPH; ++i){
400         if((m = lp->atom->map(i+1))>=0){ //map(i) returns local number of particle
401             with global number i (or <0 if i not one of local particles)
```

```

400     for(int j=0; j<3; ++j){
401         v[m][j]=vSPH(i,j);
402     }
403     //     cout << "Teilchen " << i << " initialisiert mit vSPH = " << vSPH(i
         ,0) << " " << vSPH(i,1) << " " << vSPH(i,2) << endl;
404 }
405 }
406
407 }
408
409 void DataL::getrohSPH(){
410
411     if(proc == 0){
412         char* mem = new char[10*nSPH]; //memory which ntohd and htohdn is working on
         ; size = 10*(number of doubles in array)
413
414         //with consistency check
415         int remain=10*nSPH;
416         while(remain!=0) remain-=recv(c,mem+(10*nSPH-remain),remain,0);
417         ntohd(mem,_rohSPH,nSPH);
418
419         delete [] mem;
420     }
421     MPI_Bcast(_rohSPH,nSPH,MPI_DOUBLE,0,MPI_COMM_WORLD);
422
423     //rohSPH initialization in LIGGGHTS atom array
424     double* roh = lp->atom->density;
425     int m;
426     for(int i=0; i<nSPH; ++i){
427         if((m = lp->atom->map(i+1))>=0){ //map(i) returns local number of particle
         with global number i (or <0 if i not one of local particles)
428             roh[m]=rohSPH(i);
429         }
430     }
431
432 }
433
434 void DataL::getrv(){
435     if(proc == 0){
436         char* mem = new char[30*(n-nstat)]; //memory which ntohd and htohdn is
         working on; size = 10*(number of doubles in array)
437
438         //with consistency check
439         int remain=30*(n-nstat);
440         while(remain!=0) remain-=recv(c,mem+(30*(n-nstat)-remain),remain,0);

```

```
441     ntohd(mem,_r,3*(n-nstat));
442
443     remain=30*(n-nstat);
444     while(remain!=0) remain-=recv(c,mem+(30*(n-nstat)-remain),remain,0);
445     ntohd(mem,_v,3*(n-nstat));
446
447     delete [] mem;
448 }
449 MPI_Bcast(_r,3*(n-nstat),MPI_DOUBLE,0,MPI_COMM_WORLD);
450 MPI_Bcast(_v,3*(n-nstat),MPI_DOUBLE,0,MPI_COMM_WORLD);
451 }
452
453 void DataL::getrvfull(){
454     if(proc == 0){
455         char* mem = new char[30*n]; //memory which ntohd and htond is working on;
456         size = 10*(number of doubles in array)
457
458         //with consistency check
459         int remain=30*n;
460         while(remain!=0) remain-=recv(c,mem+(30*n-remain),remain,0);
461         ntohd(mem,_r,3*n);
462
463         remain=30*n;
464         while(remain!=0) remain-=recv(c,mem+(30*n-remain),remain,0);
465         ntohd(mem,_v,3*n);
466
467         delete [] mem;
468     }
469     MPI_Bcast(_r,3*n,MPI_DOUBLE,0,MPI_COMM_WORLD);
470     MPI_Bcast(_v,3*n,MPI_DOUBLE,0,MPI_COMM_WORLD);
471 }
472 DataL::~DataL(){
473     if(proc == 0){
474         //close TCP sockets / cleanup
475         close(c);
476     }
477
478     //memory management
479     delete [] _r;
480     delete [] _v;
481     delete [] _f;
482     delete [] _el;
483     delete [] _rSPH;
484     delete [] _vSPH;
```

```

485     delete [] _rohSPH;
486
487     delete lp;
488 }
489
490 int DataL::readone(){
491     unsigned int length,remain;
492     if(proc == 0){
493         recv(c,reinterpret_cast<char *>(&length),4,0);
494         length=ntohl(length);
495     }
496     MPI_Bcast(&length,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
497     remain=length;
498     char* temp = new char[length];
499     if(proc == 0){
500         while(remain!=0) remain-=recv(c,temp+(length-remain),remain,0);
501     }
502     MPI_Bcast(temp,length,MPI_CHAR,0,MPI_COMM_WORLD);
503     if(temp[0]=='q' && temp[1]=='q' && temp[2]=='q' && temp[3]=='q'){
504         delete [] temp;
505         return 0;
506     }
507     else{
508         lp->input->one(temp);
509         //std::cout << temp << std::endl;
510         delete [] temp;
511         return 1;
512     }
513 }
514
515 void DataL::get_ref_opt(){
516     if(proc == 0){
517         recv(c,reinterpret_cast<char *>(&refinement_option),4,0);
518         refinement_option=ntohl(refinement_option);
519     }
520     MPI_Bcast(&refinement_option,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
521 }
522
523 void DataL::get_ref_res(){
524     if(proc==0){
525         char temp[10];
526         int remain=10;
527         while(remain!=0) remain-=recv(c,temp+(10-remain),remain,0);
528         ntohd(temp, &dr, 1);
529     }

```

```
530 MPI_Bcast(&dr,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
531 }
532
533 int DataL::recv_command(){
534     unsigned int length,remain;
535     if(proc == 0){
536         recv(c,reinterpret_cast<char *>(&length),4,0);
537         length=ntohl(length);
538     }
539     MPI_Bcast(&length,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
540     remain=length;
541     char* temp = new char[length];
542     if(proc == 0){
543         while(remain!=0) remain-=recv(c,temp+(length-remain),remain,0);
544     }
545     MPI_Bcast(temp,length,MPI_CHAR,0,MPI_COMM_WORLD);
546
547     if(strncmp(temp,"send SPH",length)==0){
548         this->sendrSPH();
549         this->sendvSPH();
550         this->sendrohSPH();
551
552         delete [] temp;
553         return 1;
554     }
555
556     if(strncmp(temp,"send f",length)==0){
557         this->sendforce();
558
559         delete [] temp;
560         return 1;
561     }
562
563     if(strncmp(temp,"dummy",length)==0){
564         delete [] temp;
565         return 1;
566     }
567
568     if(strncmp(temp,"recv rv",length)==0){
569         this->getrv();
570
571         delete [] temp;
572         return 1;
573     }
574
```

```

575     delete [] temp;
576     return 0;
577 }
578
579 //refines original triangle i recursively until edges are shorter than dr
580 //saves data in terms of barycentric coordinates in ref_triangle
581 //a1...c3 vertices of current sub-triangle in barycentric coordinates
582 //r1,r2,r3 ... vertices of original triangle in cartesian coordinates
583 void DataL::recursive_refine(int i, const double* r1, const double* r2, const
    double* r3, double a1,double a2,double a3,double b1,double b2,double b3,
    double c1,double c2,double c3){
584
585     //vertices of current sub-triangle in cartesian coordinates
586     double x11 = a1*r1[0]+a2*r2[0]+a3*r3[0];
587     double x12 = a1*r1[1]+a2*r2[1]+a3*r3[1];
588     double x13 = a1*r1[2]+a2*r2[2]+a3*r3[2];
589
590     double x21 = b1*r1[0]+b2*r2[0]+b3*r3[0];
591     double x22 = b1*r1[1]+b2*r2[1]+b3*r3[1];
592     double x23 = b1*r1[2]+b2*r2[2]+b3*r3[2];
593
594     double x31 = c1*r1[0]+c2*r2[0]+c3*r3[0];
595     double x32 = c1*r1[1]+c2*r2[1]+c3*r3[1];
596     double x33 = c1*r1[2]+c2*r2[2]+c3*r3[2];
597
598     double e1 = (x21-x11)*(x21-x11) + (x22-x12)*(x22-x12) + (x23-x13)*(x23-x13);
599     double e2 = (x31-x21)*(x31-x21) + (x32-x22)*(x32-x22) + (x33-x23)*(x33-x23);
600     double e3 = (x31-x11)*(x31-x11) + (x32-x12)*(x32-x12) + (x33-x13)*(x33-x13);
601
602     double rc2 = dr*dr;
603
604     if(e1 > rc2 || e2 > rc2 || e3 > rc2) //make a recursive sub-division if one
        side of current triangle is longer than dr=sqrt(rc2)
605     {
606         recursive_refine(i,r1,r2,r3,a1,a2,a3,0.5*(a1+b1),0.5*(a2+b2),0.5*(a3+b3)
            ,0.5*(a1+c1),0.5*(a2+c2),0.5*(a3+c3));
607         recursive_refine(i,r1,r2,r3,0.5*(a1+b1),0.5*(a2+b2),0.5*(a3+b3),b1,b2,b3
            ,0.5*(b1+c1),0.5*(b2+c2),0.5*(b3+c3));
608         recursive_refine(i,r1,r2,r3,0.5*(a1+c1),0.5*(a2+c2),0.5*(a3+c3),0.5*(b1+c1)
            ,0.5*(b2+c2),0.5*(b3+c3),c1,c2,c3);
609         recursive_refine(i,r1,r2,r3,0.5*(a1+b1),0.5*(a2+b2),0.5*(a3+b3),0.5*(a1+c1)
            ,0.5*(a2+c2),0.5*(a3+c3),0.5*(b1+c1),0.5*(b2+c2),0.5*(b3+c3));
610     }
611     else
612     {

```

```

613     AddSubTriangle(i,a1,a2,a3,b1,b2,b3,c1,c2,c3);
614 }
615 }
616
617 void DataL::refine_mesh(){
618     const double* r1;
619     const double* r2;
620     const double* r3;
621     for(int i=0; i<nel; ++i){
622         r1 = rp(el(i,0));
623         r2 = rp(el(i,1));
624         r3 = rp(el(i,2));
625         //if(proc == 0){
626         // std::cout << "proc " << proc << ": " << "refining element " << i << std
        //         ::endl;
627         // //std::cout << r1 << " " << r2 << " " << r3 << std::endl;
628         //}
629         recursive_refine(i,r1,r2,r3,1.,0.,0.,0.,1.,0.,0.,0.,1.);
630     }
631
632     if(proc==0){
633         int count = 0;
634         for(int i=0; i<nel; ++i)
635             count += nsub(i);
636         std::cout << "Total number of 3D surface elements (after pre-refinement): "
        << count << std::endl;
637     }
638 }

```

## A.10. fix\_FSI\_SPH\_v2\_1.h

```

1  /* -----
2  LIGGGHTS - LAMMPS Improved for General Granular and Granular Heat
3  Transfer Simulations
4
5  www.liggghts.com | www.cfdem.com
6  Christoph Kloss, christoph.kloss@cfdem.com
7
8  LIGGGHTS is based on LAMMPS
9  LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
10 http://lammps.sandia.gov, Sandia National Laboratories
11 Steve Plimpton, sjplimp@sandia.gov
12
13 Copyright (2003) Sandia Corporation. Under the terms of Contract
14 DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains

```

```
15 certain rights in this software. This software is distributed under
16 the GNU General Public License.
17
18 See the README file in the top-level LAMMPS directory.
19 ----- */
20
21 /* -----
22 Contributing author for SPH:
23 Andreas Aigner (CD Lab Particulate Flow Modelling, JKU)
24 andreas.aigner@jku.at
25 ----- */
26
27 /* -----
28 FSI and coupling to external program by Markus Schörghöhner, mkschoe@gmail.
   com
29 ----- */
30
31 #ifndef CLASS_CELL
32 #define CLASS_CELL
33
34 struct intdyn{ //dynamic integer array
35     int x;
36     intdyn* next;
37 };
38
39 class cell_list{
40 public:
41     int nx; //number of cells in each direction
42     int ny;
43     int nz;
44     double lx; // cell dimensions
45     double ly;
46     double lz;
47     double xmin;
48     double xmax;
49     double ymin;
50     double ymax;
51     double zmin;
52     double zmax;
53     double rc;
54     intdyn*** cl; //cell list
55     cell_list(double xlo, double xhi, double ylo, double yhi, double zlo, double
         zhi, double rc); // rc ... maximum range of wall interaction
56     ~cell_list();
```

```
57 void hash(double* r, int i); //hash particle with (local) number i and
    coordinates r
58 void dyn_delete(); //clear complete cell list
59 };
60
61 //save cell indices in x,y,z direction of a particle with coordinates r in i,j
    ,k; cell grid defined by cl;
62 void getindex(int& i, int& j, int& k, double* r, cell_list& cl);
63
64 #endif
65
66 #ifdef FIX_CLASS
67
68 FixStyle(wall/sph_fsi,FixWallSPH_FSI)
69
70 #else
71
72 #ifndef LMP_FIX_WALL_SPH_FSI_H
73 #define LMP_FIX_WALL_SPH_FSI_H
74
75 #include "fix_sph.h"
76
77 namespace LAMMPS_NS {
78
79 class FixWallSPH_FSI : public FixSPH {
80 public:
81 double rc; //cut-off
82 double r0; //equilibrium distance
83 double k; //maximum repulsive force for particle-wall distance r=0
84 double t; //maximum viscous force between fluid and wall for particle-wall
    distance r=0 and deltaV = 1 m/s
85 //defines viscosity for fluid-wall interaction --> see notes
86 FixWallSPH_FSI(class LAMMPS *, int, char **);
87 ~FixWallSPH_FSI();
88 int setmask();
89 void init();
90 void setup(int vflag);
91 void post_force(int vflag);
92 void post_force_respa(int vflag, int ilevel, int iloop);
93 };
94
95 }
96
97 #endif
98 #endif
```

```

99
100 #ifndef FSI3D
101 #define FSI3D
102 //-----//
103 //FSI wall interaction routines 3D
104 //-----//
105
106 //calculates interaction of one surface triangle with the local SPH particles
    and writes results to LIGGGHTS and DataL force arrays
107 //7-point Gauss surface integration
108 //using recursive mesh-refinement
109
110 //ngauss... number of Gauss points
111 //rbcgauss... barycentric coordinates of Gauss points (rbcgauss[0] is the
    coordinate tripe1 corresponding to the first Gauss point)
112 //wgauss... corresponding weights
113 //r,v,f ... local LIGGGHTS SPH arrays
114 //cl... cell_list for cell search (pair interactions)
115 //rc2... square of force cut-off radius rc
116 //h... smoothing lenght correspondence (h = rc/2)
117 //alpha, beta... parameters for repulsive force
118 //t... parameter for viscous force (wall friction)
119 //r1,r2,r3 ... vertices of basis triangle (from DataL object)
120 //v1,.. corresponding velocities (from DataL object)
121 //f1,.. corresponding forces (from DataL object)
122 //x11...x33 ... coordinates of vertices of current sub-triangle in recursion
123
124 void rec_triangle_int(int ngauss, double rbcgauss[][3], double wgauss[],
    double** x, double** v, double** f, cell_list& cl,
125     const double& rc2, const double& h, const double& alpha, const
    double& beta, const double& t,
126     const double* r1, const double* r2, const double* r3, const double*
    v1, const double* v2, const double* v3, double* f1, double* f2,
    double* f3,
127     double x11, double x12, double x13, double x21, double x22, double
    x23, double x31, double x32, double x33);
128
129 //same as rec_triangle_int, but without recursion
130 void triangle_int(int ngauss, double rbcgauss[][3], double wgauss[], double**
    x, double** v, double** f, cell_list& cl,
131     const double& rc2, const double& h, const double& alpha, const double&
    beta, const double& t,
132     const double* r1, const double* r2, const double* r3, const double* v1
    , const double* v2, const double* v3, double* f1, double* f2,
    double* f3,

```

```
133     double x11, double x12, double x13, double x21, double x22, double x23
        , double x31, double x32, double x33);
134
135 //calculates barycentric coordinates of point x=(p,q,r) in triangle with
        vertices r1=(x1,y1,z1),r2=(x2,y2,z2),r3=(x3,y3,z3) and saves data in b1,b2
        ,b3
136 void calc_barycentric_coordinates(double& b1, double& b2, double& b3, double p
        , double q, double r, double x1, double x2, double x3, double y1, double
        y2, double y3, double z1, double z2, double z3);
137
138 /* //used for old version of calc_barycentric_coordinates
139 //calculate barycentric coordinates for points in xy/xz/yz-layer; cf. also
        calc_barycentric_coordinates
140 void calc_barycentric_coordinates_yzlayer(double b1,double b2,double b3,double
        q,double r,double y1,double y2,double y3,double z1,double z2,double z3);
141 void calc_barycentric_coordinates_xyayer(double b1,double b2,double b3,double
        p,double q,double x1,double x2,double x3,double y1,double y2,double y3);
142 void calc_barycentric_coordinates_xzlayer(double b1,double b2,double b3,double
        p,double r,double x1,double x2,double x3,double z1,double z2,double z3);
143 */
144
145 //area of a triangle with vertices r1,r2,r3
146 double triangle_area(const double* r1, const double* r2, const double* r3);
147
148 //area of a triangle with vertices r1=(x11,x12,x13),r2=(x21,x22,x23),r3=(x31,
        x32,x33)
149 double triangle_area(double x11, double x12, double x13, double x21, double
        x22, double x23, double x31, double x32, double x33);
150
151 //square of area of a triangle with vertices r1=(x11,x12,x13),r2=(x21,x22,x23)
        ,r3=(x31,x32,x33)
152 double triangle_area_sq(double x11, double x12, double x13, double x21, double
        x22, double x23, double x31, double x32, double x33);
153
154 #endif
```

## A.11. fix\_FSI\_SPH\_v2\_1.cpp

```
1  /* -----
2  LIGGGHTS - LAMMPS Improved for General Granular and Granular Heat
3  Transfer Simulations
4
5  www.liggghts.com | www.cfdem.com
6  Christoph Kloss, christoph.kloss@cfdem.com
7
```

```
8 LIGGGHTS is based on LAMMPS
9 LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
10 http://lammps.sandia.gov, Sandia National Laboratories
11 Steve Plimpton, sjplimp@sandia.gov
12
13 Copyright (2003) Sandia Corporation. Under the terms of Contract
14 DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
15 certain rights in this software. This software is distributed under
16 the GNU General Public License.
17
18 See the README file in the top-level LAMMPS directory.
19 ----- */
20
21 /* -----
22 Contributing author for SPH:
23 Andreas Aigner (CD Lab Particulate Flow Modelling, JKU)
24 andreas.aigner@jku.at
25 ----- */
26
27 /* -----
28 FSI and coupling to external program by Markus Schörghöhumer, mkschoe@gmail.
29 com
30 ----- */
31 #include "math.h"
32 #include "stdlib.h"
33 #include "string.h"
34 #include "atom.h"
35 #include "force.h"
36 #include "pair.h"
37 #include "modify.h"
38 #include "memory.h"
39 #include "domain.h"
40 #include "respa.h"
41 #include "update.h"
42 #include "error.h"
43 #include "sph_kernels.h"
44 #include "fix_FSI_SPH_v2_1.h"
45 #include <iostream>
46 #include "exchange_class_linux.h"
47
48 #include "mpi.h"
49
50 using namespace LAMMPS_NS;
51 using namespace std;
```

```
52
53 cell_list::cell_list(double xlo, double xhi, double ylo, double yhi, double
    zlo, double zhi, double rc):
54 xmin(xlo), xmax(xhi), ymin(ylo), ymax(yhi), zmin(zlo), zmax(zhi), rc(rc) {
55     nx = floor((xmax-xmin)/rc);
56     ny = floor((ymax-ymin)/rc);
57     nz = floor((zmax-zmin)/rc);
58     if(nx==0) ++nx;
59     if(ny==0) ++ny;
60     if(nz==0) ++nz;
61     lx = (xmax-xmin)/nx;
62     ly = (ymax-ymin)/ny;
63     lz = (zmax-zmin)/nz;
64     cl = new intdyn***[nx];
65     for(int i=0; i<nx; ++i){
66         cl[i] = new intdyn**[ny];
67         for(int j=0; j<ny; ++j){
68             cl[i][j] = new intdyn*[nz];
69             for(int k=0; k<nz; ++k){
70                 cl[i][j][k] = NULL;
71             }
72         }
73     }
74 }
75
76 void cell_list::hash(double* r, int num){
77     int i,j,k;
78     getindex(i,j,k,r,*this);
79     if( i>=0 && i<nx && j>=0 && j<ny && k>=0 && k<nz ){
80
81         //std::cout << "hash: " << "num " << num << "(i,j,k): (" << i << ", " << j
            << ", " << k << ")" << std::endl;
82
83         intdyn* temp = cl[i][j][k];
84         if(temp){
85             while(temp->next) temp=temp->next; //find last element - temp == NULL
86             temp->next = new intdyn;
87             temp->next->x = num;
88             temp->next->next = NULL;
89         }
90         else{
91             cl[i][j][k] = new intdyn;
92             cl[i][j][k]->x = num;
93             cl[i][j][k]->next = NULL;
94         }

```

```
95  }
96  else{
97      std::cout << "hash: particle outside domain" << " " << r[0] << " " << r[1]
          << " " << r[2] << std::endl;
98      MPI_Abort(MPI_COMM_WORLD,1);
99  }
100 }
101
102 void cell_list::dyn_delete(){
103     intdyn* temp;
104     intdyn* temp1;
105
106     for(int i=0; i<nx; ++i){
107         for(int j=0; j<ny; ++j){
108             for(int k=0; k<nz; ++k){
109                 if(cl[i][j][k]){
110                     temp = cl[i][j][k];
111                     while(temp){
112                         temp1 = temp->next;
113                         delete temp;
114                         temp = temp1;
115                     }
116                     cl[i][j][k]=NULL;
117                 }
118             }
119         }
120     }
121
122 }
123
124 cell_list::~cell_list(){
125     intdyn* temp;
126     intdyn* temp1;
127
128     for(int i=0; i<nx; ++i){
129         for(int j=0; j<ny; ++j){
130             for(int k=0; k<nz; ++k){
131                 temp = cl[i][j][k];
132                 while(temp){
133                     temp1 = temp->next;
134                     delete temp;
135                     temp = temp1;
136                 }
137             }
138             delete [] cl[i][j];
```

```
139     }
140     delete [] cl[i];
141 }
142 delete [] cl;
143 }
144
145 void getindex(int& i, int& j, int& k, double* r, cell_list& cl){
146     i = int( (r[0]-cl.xmin)/cl.lx );
147     j = int( (r[1]-cl.ymin)/cl.ly );
148     k = int( (r[2]-cl.zmin)/cl.lz );
149     double eps=(cl.xmax-cl.xmin)*1E-12;
150     if(abs(cl.xmax-r[0])<eps && i==cl.nx) --i; //in case a SPH particle lies "
        exactly" on the domain border
151     if(abs(cl.ymax-r[1])<eps && j==cl.ny) --j; //don't know how LAMMPS handles
        this (for local atoms)
152     if(abs(cl.zmax-r[2])<eps && k==cl.nz) --k; //doesn't matter for boundary (
        Gauss) particles, since all of them
153     //are hashed on each proc and i,j,k = -1 or nx,ny,nz is allowed
154     if(abs(cl.xmin-r[0])<eps && i==-1) ++i;
155     if(abs(cl.ymin-r[1])<eps && j==-1) ++j;
156     if(abs(cl.zmin-r[2])<eps && k==-1) ++k;
157 }
158
159 /* ----- */
160
161 FixWallSPH_FSI::FixWallSPH_FSI(LAMMPS *lmp, int nargs, char **arg) :
162 FixSPH(lmp, nargs, arg)
163 {
164     //first 3 arguments: fix id, group id, fix style
165     //parameters of force calculation as following (see post_force()): rc, r0, k
        , t
166     if(nargs != 7) error->all("Illegal fix wall/sph_fsi command");
167     rc = force->numeric(arg[3]);
168     r0 = force->numeric(arg[4]);
169     k = force->numeric(arg[5]);
170     t = force->numeric(arg[6]);
171 }
172
173 /* ----- */
174
175 FixWallSPH_FSI::~FixWallSPH_FSI()
176 {
177
178 }
179
```

```

180  /* ----- */
181
182  int FixWallSPH_FSI::setmask()
183  {
184      int mask = 0;
185      mask |= POST_FORCE;
186      mask |= POST_FORCE_RESPA;
187      return mask;
188  }
189
190  /* ----- */
191
192  void FixWallSPH_FSI::init()
193  {
194      FixSPH::init();
195
196      if (strcmp(update->integrate_style,"respa") == 0)
197          nlevels_respa = ((Respa *) update->integrate)->nlevels;
198  }
199
200  /* ----- */
201
202  void FixWallSPH_FSI::setup(int vflag)
203  {
204      if (strcmp(update->integrate_style,"verlet") == 0)
205          post_force(vflag);
206      else {
207          ((Respa *) update->integrate)->copy_flevel_f(nlevels_respa-1);
208          post_force_respa(vflag,nlevels_respa-1,0);
209          ((Respa *) update->integrate)->copy_f_flevel(nlevels_respa-1);
210      }
211  }
212
213  /* ----- */
214
215  void FixWallSPH_FSI::post_force(int vflag)
216  {
217      DataL* obj = lmp->exchange;
218      double dr_refine = obj->get_dr();
219
220      /**/parameters for wall interaction - will be read in via input script and
221          included as class members
222
221      double rc = 1.;
222      double r0=0.75*2*h; //equilibrium distance for adhesive effect
223      double k=100.; //maximum repulsive force at d=0

```

```
224 double t=1.; //maximum visuous force (at d=0) and deltaV = 1 m/s in wall/SPH
      interaction
225 */
226
227 double h=rc/2.; //cut-off is 2*h
228 double hinv = 1.0/h;
229 double alpha = k/(4.0*h*h*r0*(4.0*h-r0)); // see notes
230
231 double beta = (2.0*h-r0)*(2.0*h-r0);
232
233 //double alpha = k/(2.0*h*2.0*h*2.0*h*2.0*h*2.0*h*2.0*h*2.0*h*2.0*h-beta
      *2.0*h*2.0*h); //8th order repulsion, 2nd order attraction
234
235 cell_list cl(lmp->domain->sublo[0],lmp->domain->subhi[0],lmp->domain->sublo
      [1],
236 lmp->domain->subhi[1],lmp->domain->sublo[2],lmp->domain->subhi[2],rc);
237
238 int nlocal = lmp->atom->nlocal;
239 int* mask = lmp->atom->mask;
240 double** x = lmp->atom->x;
241 //cl.dyn_delete(); //not needed; would be, if cell list was created only
      once and included as member in the DataL / exchange class
242
243 for(int i=0; i<nlocal; ++i){ //create cell list of local SPH particles
244     if (mask[i] & groupbit){
245         cl.hash(x[i],i);
246     }
247 }
248
249 //intdyn* temp2;
250 //for(int c1=0; c1<cl.nx; ++c1) // move through cell and all neighboring
      cells
251 // for(int c2=0; c2<cl.ny; ++c2)
252 // for(int c3=0; c3<cl.nz; ++c3){
253 //     temp2=cl.cl[c1][c2][c3];
254 //     while(temp2){
255 //         int num=temp2->x; //current local SPH particle number
256 //         std::cout << "num out: " << num << std::endl;
257 //         temp2 = temp2->next;
258 //     }
259 // }
260
261 //reset force array of boundary and calculate interaction
262 obj->zerof();
263 double** f= lmp->atom->f;
```

```

264 double** v= lmp->atom->v;
265 int nel = obj->nel;
266 int dim = obj->dim;
267
268 if(dim == 2){
269
270     double d,l,w1,w2;
271     double floc[3],ploc[3],vloc[3];
272     double fabs;
273     int ix,iy,iz,num;
274     double* r1,*r2,*v1,*v2,*f1,*f2;
275     intdyn* temp;
276
277     //3-point 1d Gauss-Legendre integration
278     int ngauss = 3;
279     double gaussp[]={-sqrt(3./5.),0.,sqrt(3./5.)};
280     double weights[]={5./9.,8./9.,5./9.};
281     //subdivision of line segments in nsub sub-segments with equal lengths dl
        <= rc, if l > rc
282     //reason for this: see MM file "Single Boundary Particles vs Convolution
        Integral" and notes
283     int nsub;
284     double dl;
285
286     //double alphad = 15.0/(7.0*h*h*abs(asin(1.0))*2.0); //normalization of
        cubic spline kernel in 2D -- not needed anymore
287
288     for(int i=0; i<nel; ++i){
289         r1=obj->rp(obj->el(i,0)); //get coordinates and velocities of boundary
            points (end points of line segment)
290         v1=obj->vp(obj->el(i,0));
291         f1=obj->fp(obj->el(i,0));
292         r2=obj->rp(obj->el(i,1));
293         v2=obj->vp(obj->el(i,1));
294         f2=obj->fp(obj->el(i,1));
295         l=sqrt( (r1[0]-r2[0])*(r1[0]-r2[0])+(r1[1]-r2[1])*(r1[1]-r2[1])+(r1[2]-r2
            [2])*(r1[2]-r2[2]) );
296         //calculate necessary subdivisions
297
298         /*
299         //nsub=ceil(l/rc); //dl <= rc
300         nsub=ceil(l/(dr_refine)); //dl <= dr (cf. interface_baseclass::dr
301         dl = l/double(nsub);
302         */
303

```

```
304 //test version based on iterative bisection
305 double div=1.0;
306 int nsub=1;
307 while(1){
308     if(1/div > dr_refine){
309         div*=2.0;
310         nsub*=2;
311     }
312     else
313         break;
314 }
315 dl = 1/div;
316
317 //std::cout << l << " " << dl << " " << nsub << std::endl;
318
319 for(int isub=1; isub<=nsub; ++isub){
320
321     for(int j=0; j<ngauss; ++j){ //for every line segment (after subdivision
322         ) 3-point Gauss-Legendre integration
323         w2 = (double(isub-1)+0.5*(gaussp[j]+1.0))*dl/l; //weights for force
324             distribution -- corresponding to lin interpolation
325         w1 = 1.0 - w2;
326         for(int m=0; m<3; ++m){
327             ploc[m]=r1[m]+w2*(r2[m]-r1[m]); //calculate local Gauss point
328         }
329         getindex(ix,iy,iz,ploc,cl);
330
331         //if(i==6)
332         // std::cout << "i: " << i << ", isub: " << isub << ", d_left: " <<
333             ploc[0] << " d_right: " << 0.05-ploc[0] << std::endl;
334
335         if(ix<=cl.nx && ix>=-1 && iy<=cl.ny && iy>=-1 && iz<=cl.nz && iz>=-1){
336             //check if interaction possible
337
338             for(int m=0; m<3; ++m){
339                 vloc[m]=v1[m]+w2*(v2[m]-v1[m]); // velocity corresponding to ploc
340                 with linear interpolation
341             }
342
343             for(int c1=ix-1; c1<=ix+1; ++c1) // move through cell and all
344                 neighboring cells
345             for(int c2=iy-1; c2<=iy+1; ++c2)
346                 for(int c3=iz-1; c3<=iz+1; ++c3)
347                     if(c1<cl.nx && c1>=0 && c2<cl.ny && c2>=0 && c3<cl.nz && c3>=0){
348                         //check if valid cell
```

```

342         //move through all SPH particles in the cell
343         temp=c1.c1[c1][c2][c3];
344         while(temp){
345             num=temp->x; //current local SPH particle number
346
347             //std::cout << "num: " << num << " c1: " << c1 << " c2: " <<
                c2 << " c3: " << c3 << std::endl;
348
349             d=sqrt( (ploc[0]-x[num][0])*(ploc[0]-x[num][0])+(ploc[1]-x[
                num][1])*(ploc[1]-x[num][1])+(ploc[2]-x[num][2])*(ploc
                [2]-x[num][2]) );
350
351             if(d<=2.0*h){ //cut-off
352                 //boundary traction/repulsion acc to Müller,Schrim,Teschner
                    et al.: Interaction of fluids with deformable solids
                    2004
353
354                 fabs = (2.0*h-d)*(2.0*h-d);
355                 fabs = alpha*(fabs*fabs-beta*fabs);
356
357                 /*fabs = (2.0*h-d)*(2.0*h-d);
358                 fabs*=fabs;
359                 fabs = alpha*(fabs*fabs-beta*(2.0*h-d)*(2.0*h-d));
360                 */
361
362                 //calculate force on SPH particle and add to force arrays
363                 for(int q=0; q<3; ++q){
364                     floc[q]=0.5*d1*weights[j]*fabs*(x[num][q]-ploc[q])/d;
365                     //if(0.5*hinv*d<0.1) std::cout << "Warning: SPH-wall
                        distance d/rc < 1/10 encountered" << std::endl;
366
367                     if(floc[q]!=floc[q]){
368                         std::cout << "floc rep: " << floc[q] << std::endl;
369                         //std::cin.get();
370                         MPI_Abort(MPI_COMM_WORLD,1);
371                     }
372                     if(f[num][q]!=f[num][q]){
373                         std::cout << "f rep: " << f[num][q] << std::endl;
374                         //std::cin.get();
375                         MPI_Abort(MPI_COMM_WORLD,1);
376                     }
377
378                     f[num][q]+=floc[q];
379                     //force distribution to end points of line segment, see
                        notes and MM file

```

```
380         f1[q]-=w1*floc[q];
381         f2[q]-=w2*floc[q];
382     }
383
384     //if(1){//num<=3 || num ==52 || num==53 || num==54){//num
        ==0 || num==54){
385     // std::cout << "Teilchen Nr. " << num << std::endl;
386     // std::cout << "r = (" <<x[num] [0]<<", "<<x[num] [1]<<",
        "<<x[num] [2]<<)"<<std::endl;
387     // std::cout << "v = ("<<v[num] [0]<<", "<<v[num] [1]<<", "<<
        v[num] [2]<<)"<<std::endl;
388     // std::cout << "floc rep = ("<<floc[0]<<", "<<floc[1]<<",
        "<<floc[2]<<)"<<std::endl;
389     // std::cout << "f = ("<<f[num] [0]<<", "<<f[num] [1]<<", "<<
        f[num] [2]<<)"<<std::endl;
390     // std::cout << "ploc " << i << ", isub " << isub << ": "
        <<ploc[0]<<", "<<ploc[1]<<", "<<ploc[2]<<)"<<std::endl
        ;
391     // std::cout << "vloc = ("<<vloc[0]<<", "<<vloc[1]<<", "<<
        vloc[2]<<)"<<std::endl;
392     // std::cout << "d/h = " << d/h <<std::endl;
393     //}
394
395     //boundary friction / viscous terms acc. to: see above,
        but with normalized cubic spline kernel
396     fabs = d*hinu;
397
398     /* //for cubic spline:
399
400     if(fabs<=1.0)
401     fabs = 0.5*t*(-2.0+3.0*fabs);
402     else
403     fabs = 0.5*t*(2.0-fabs);
404     */
405
406     //for spiky kernel
407     fabs = 0.5*t*(2.0-fabs);
408
409     //calculate force on SPH particle and add to force arrays
410     for(int q=0; q<3; ++q){
411         floc[q]=0.5*d1*weights[j]*fabs*(vloc[q]-v[num][q]);
412
413         /*if(vloc[q]!=vloc[q]){
414         std::cout << "vwall: " << vloc[q] << std::endl;
415         std::cin.get();
```

```

416     }
417     if(v[num][q]!=v[num][q]){
418     std::cout << "vSPH: " << v[num][q] << std::endl;
419     std::cin.get();
420     }*/
421     if(floc[q]!=floc[q]){
422     std::cout << "floc visc: " << floc[q] << std::endl;
423     //std::cin.get();
424     MPI_Abort(MPI_COMM_WORLD,1);
425     }
426     if(f[num][q]!=f[num][q]){
427     std::cout << "f visc: " << f[num][q] << std::endl;
428     //std::cin.get();
429     MPI_Abort(MPI_COMM_WORLD,1);
430     }
431
432     f[num][q]+=floc[q];
433     //force distribution to end points of line segment, see
434     //notes and MM file
435     f1[q]-=w1*floc[q];
436     f2[q]-=w2*floc[q];
437     }
438
439     //if(1){//num<=3 || num ==52 || num==53 || num==54){//num
440     //==0 || num==54){
441     // std::cout << "Teilchen Nr. " << num << std::endl;
442     // std::cout << "r = (" <<x[num][0]<< ", "<<x[num][1]<< ",
443     // "<<x[num][2]<< ")"<<std::endl;
444     // std::cout << "v = ("<<v[num][0]<< ", "<<v[num][1]<< ",
445     // "<<v[num][2]<< ")"<<std::endl;
446     // std::cout << "floc visc = ("<<floc[0]<< ", "<<floc
447     // [1]<< ", "<<floc[2]<<)"<<std::endl;
448     // std::cout << "f = ("<<f[num][0]<< ", "<<f[num][1]<< ",
449     // "<<f[num][2]<<)"<<std::endl;
450     // std::cout << "ploc " << i << ", isub " << isub << ": "
451     // <<ploc[0]<< ", "<<ploc[1]<< ", "<<ploc[2]<<)"<<std:
452     // endl;
453     // std::cout << "vloc = ("<<vloc[0]<< ", "<<vloc[1]<< ",
454     // "<<vloc[2]<<)"<<std::endl;
455     // std::cout << "d/h = " << d/h <<std::endl;
456     //}
457
458     }
459     temp=temp->next;
460 }

```

```
452         }
453     }
454 }
455 }
456 }
457 }
458 else if(dim==3){
459
460     //dim == 3;
461     int option = obj->get_refinement_option();
462     if(!(option==1 || option==2 || option==3 || option==0)){
463         std::cout << "invalid refinement option: " << " " << option << std::endl;
464         MPI_Abort(MPI_COMM_WORLD,1);
465     }
466
467     //0...no refinement, 1... recursive refinement based on original triangles
468     //2... pre-refined mesh, no additional refinement in time-stepping, 3... as
469     //1, but based on pre-refined mesh; default is 1
470
471     //local Gauss weights and points (in barycentric coordinates)
472     int ngauss=7;
473     double dri = 1.0/3.0;
474     double a0 = 0.05971587;
475     double b0 = 0.47014206;
476     double c0 = 0.79742699;
477     double d0 = 0.10128651;
478     double e0 = (155.0+sqrt(15.0))/1200.0;
479     double f0 = (155.0-sqrt(15.0))/1200.0;
480
481     double rbcgauss[][3]={{dri,dri,dri},{a0,b0,b0},{b0,a0,b0},{b0,b0,a0},{c0,d0
482         ,d0},{d0,c0,d0},{d0,d0,c0}};
483     double wgauss[]={9.0/40.0,e0,e0,e0,f0,f0,f0};
484
485     double *r1,*r2,*r3,*v1,*v2,*v3,*f1,*f2,*f3;
486
487     if(option == 0){
488         for(int i=0; i<nel; ++i){
489             r1=obj->rp(obj->el(i,0)); //get coordinates and velocities of boundary
490             //points (end points of line segment)
491             v1=obj->vp(obj->el(i,0));
492             f1=obj->fp(obj->el(i,0));
493             r2=obj->rp(obj->el(i,1));
494             v2=obj->vp(obj->el(i,1));
495             f2=obj->fp(obj->el(i,1));
```

```

493     r3=obj->rp(obj->el(i,2));
494     v3=obj->vp(obj->el(i,2));
495     f3=obj->fp(obj->el(i,2));
496
497     triangle_int(ngauss, rbcgauss, wgauss, x, v, f, cl, dr_refine*dr_refine,
498               h, alpha, beta, t, r1, r2, r3, v1, v2, v3, f1, f2, f3,
499               r1[0],r1[1],r1[2],r2[0],r2[1],r2[2],r3[0],r3[1],r3[2]);
500   }
501   else if(option==1){
502     for(int i=0; i<nel; ++i){
503       r1=obj->rp(obj->el(i,0)); //get coordinates and velocities of boundary
504         points (end points of line segment)
505       v1=obj->vp(obj->el(i,0));
506       f1=obj->fp(obj->el(i,0));
507       r2=obj->rp(obj->el(i,1));
508       v2=obj->vp(obj->el(i,1));
509       f2=obj->fp(obj->el(i,1));
510       r3=obj->rp(obj->el(i,2));
511       v3=obj->vp(obj->el(i,2));
512       f3=obj->fp(obj->el(i,2));
513
514       rec_triangle_int(ngauss, rbcgauss, wgauss, x, v, f, cl, dr_refine*
515         dr_refine, h, alpha, beta, t, r1, r2, r3, v1, v2, v3, f1, f2, f3,
516         r1[0],r1[1],r1[2],r2[0],r2[1],r2[2],r3[0],r3[1],r3[2]);
517     }
518   else if(option==2){
519     for(int i=0; i<nel; ++i){
520       r1=obj->rp(obj->el(i,0)); //get coordinates and velocities of boundary
521         points (end points of line segment)
522       v1=obj->vp(obj->el(i,0));
523       f1=obj->fp(obj->el(i,0));
524       r2=obj->rp(obj->el(i,1));
525       v2=obj->vp(obj->el(i,1));
526       f2=obj->fp(obj->el(i,1));
527       r3=obj->rp(obj->el(i,2));
528       v3=obj->vp(obj->el(i,2));
529       f3=obj->fp(obj->el(i,2));
530
531       for(int j=0; j<obj->nsub(i); ++j){
532         triangle_int(ngauss, rbcgauss, wgauss, x, v, f, cl, dr_refine*
533           dr_refine, h, alpha, beta, t, r1, r2, r3, v1, v2, v3, f1, f2, f3,

```

```
533         obj->GetCC(i,j,0,0),obj->GetCC(i,j,0,1),obj->GetCC(i,j,0,2),obj->
           GetCC(i,j,1,0),obj->GetCC(i,j,1,1),obj->GetCC(i,j,1,2),obj->GetCC
           (i,j,2,0),obj->GetCC(i,j,2,1),obj->GetCC(i,j,2,2));
534
535     }
536 }
537
538 }
539 else if(option==3){
540     for(int i=0; i<nel; ++i){
541         r1=obj->rp(obj->el(i,0)); //get coordinates and velocities of boundary
           points (end points of line segment)
542         v1=obj->vp(obj->el(i,0));
543         f1=obj->fp(obj->el(i,0));
544         r2=obj->rp(obj->el(i,1));
545         v2=obj->vp(obj->el(i,1));
546         f2=obj->fp(obj->el(i,1));
547         r3=obj->rp(obj->el(i,2));
548         v3=obj->vp(obj->el(i,2));
549         f3=obj->fp(obj->el(i,2));
550
551         for(int j=0; j<obj->nsub(i); ++j){
552
553             rec_triangle_int(ngauss, rbcgauss, wgauss, x, v, f, cl, dr_refine*
           dr_refine, h, alpha, beta, t, r1, r2, r3, v1, v2, v3, f1, f2, f3,
554             obj->GetCC(i,j,0,0),obj->GetCC(i,j,0,1),obj->GetCC(i,j,0,2),obj->
           GetCC(i,j,1,0),obj->GetCC(i,j,1,1),obj->GetCC(i,j,1,2),obj->GetCC
           (i,j,2,0),obj->GetCC(i,j,2,1),obj->GetCC(i,j,2,2));
555         }
556     }
557
558 }
559
560 }
561 else{
562     cout << "Implementation only for 2D or 3D!" << endl;
563     MPI_Abort(MPI_COMM_WORLD,1);
564 }
565
566 //now, on each proc, exchange->f contains all local force contributions
567 //MPI_Reduce is done when the sendforce() member function is called in the
           envoking program
568
569 }
570
```

```

571  /* ----- */
572
573  void FixWallSPH_FSI::post_force_respa(int vflag, int ilevel, int iloop)
574  {
575      if (ilevel == nlevels_respa-1) post_force(vflag);
576  }
577
578  //-----//
579  //FSI wall interaction routines 3D
580  //-----//
581
582  //calculates interaction of one surface triangle with the local SPH particles
      and writes results to LIGGGHTS and DataL force arrays
583  //7-point Gauss surface integration
584  //using recursive mesh-refinement
585
586  //ngauss... number of Gauss points
587  //rbcgauss... barycentric coordinates of Gauss points (rbcgauss[0] is the
      coordinate tripel corresponding to the first Gauss point)
588  //wgauss... corresponding weights
589  //x,v,f ... local LIGGGHTS SPH arrays
590  //cl... cell_list for cell search (pair interactions)
591  //rc2... square of force cut-off radius rc
592  //h... smoothing lenght correspondence (h = rc/2)
593  //alpha, beta... parameters for repulsive force
594  //t... parameter for viscuous force (wall friction)
595  //r1,r2,r3 ... vertices of basis triangle (from DataL object)
596  //v1,.. corresponding velocities (from DataL object)
597  //f1,.. corresponding forces (from DataL object)
598  //x11...x33 ... coordinates of vertices of current sub-triangle in recursion
599
600  void rec_triangle_int(int ngauss, double rbcgauss[][3], double wgauss[],
      double** x, double** v, double** f, cell_list& cl,
601      const double& rc2, const double& h, const double& alpha, const
      double& beta, const double& t,
602      const double* r1, const double* r2, const double* r3, const double*
      v1, const double* v2, const double* v3, double* f1, double* f2,
      double* f3,
603      double x11, double x12, double x13, double x21, double x22, double
      x23, double x31, double x32, double x33){
604
605      double a2 = (x21-x11)*(x21-x11) + (x22-x12)*(x22-x12) + (x23-x13)*
      (x23-x13);
606      double b2 = (x31-x21)*(x31-x21) + (x32-x22)*(x32-x22) + (x33-x23)*
      (x33-x23);

```

```
607         double c2 = (x31-x11)*(x31-x11) + (x32-x12)*(x32-x12) + (x33-x13)*(
           x33-x13);
608
609         if(a2 > rc2 || b2 > rc2 || c2 > rc2) //make a recursive sub-
           division if one side of current triangle is longer than rc
610     {
611         rec_triangle_int(ngauss,rbcgauss,wgauss,x,v,f,cl,rc2,h,alpha,beta
           ,t,r1,r2,r3,v1,v2,v3,f1,f2,f3,x11,x12,x13,0.5*(x11+x21),0.5*(
           x12+x22),0.5*(x13+x23),0.5*(x11+x31),0.5*(x12+x32),0.5*(x13+
           x33));
612         rec_triangle_int(ngauss,rbcgauss,wgauss,x,v,f,cl,rc2,h,alpha,beta
           ,t,r1,r2,r3,v1,v2,v3,f1,f2,f3,0.5*(x11+x21),0.5*(x12+x22)
           ,0.5*(x13+x23),x21,x22,x23,0.5*(x21+x31),0.5*(x22+x32),0.5*(
           x23+x33));
613         rec_triangle_int(ngauss,rbcgauss,wgauss,x,v,f,cl,rc2,h,alpha,beta
           ,t,r1,r2,r3,v1,v2,v3,f1,f2,f3,0.5*(x11+x31),0.5*(x12+x32)
           ,0.5*(x13+x33),0.5*(x21+x31),0.5*(x22+x32),0.5*(x23+x33),x31,
           x32,x33);
614         rec_triangle_int(ngauss,rbcgauss,wgauss,x,v,f,cl,rc2,h,alpha,beta
           ,t,r1,r2,r3,v1,v2,v3,f1,f2,f3,0.5*(x11+x21),0.5*(x12+x22)
           ,0.5*(x13+x23),0.5*(x11+x31),0.5*(x12+x32),0.5*(x13+x33)
           ,0.5*(x21+x31),0.5*(x22+x32),0.5*(x23+x33));
615     }
616     else
617     {
618         triangle_int(ngauss,rbcgauss,wgauss,x,v,f,cl,rc2,h,alpha,beta,t,
           r1,r2,r3,v1,v2,v3,f1,f2,f3,x11,x12,x13,x21,x22,x23,x31,x32,
           x33);
619     }
620 }
621
622 //same as rec_triangle_int, but without recursion
623 void triangle_int(int ngauss, double rbcgauss[][3], double wgauss[], double**
           x, double** v, double** f, cell_list& cl,
624         const double& rc2, const double& h, const double& alpha, const double&
           beta, const double& t,
625         const double* r1, const double* r2, const double* r3, const double* v1
           , const double* v2, const double* v3, double* f1, double* f2,
           double* f3,
626         double x11, double x12, double x13, double x21, double x22, double x23
           , double x31, double x32, double x33){
627
628         double hinv=1.0/h;
629
630         double ploc[3],vloc[3],floc[3],d,fabs,area;
```

```

631     double b1,b2,b3; //barycentric coordinates of local Gauss points w.r
        .t. the original triangle
632     int ix,iy,iz,num;
633     intdyn* temp;
634
635     for(int j=0; j<ngauss; ++j){
636         //barycentric coordinates of Gauss points w.r.t. the original (un-
            refined) triangle are used for force redistribution (c.f. notes
            and MM notebook)
637         //and linear interpolation of quantities such as velocities (
            defined only on original triangle vertices)
638
639         //local Gauss point
640         ploc[0]=rbcgauss[j][0]*x11+rbcgauss[j][1]*x21+rbcgauss[j][2]*x31;
641         ploc[1]=rbcgauss[j][0]*x12+rbcgauss[j][1]*x22+rbcgauss[j][2]*x32;
642         ploc[2]=rbcgauss[j][0]*x13+rbcgauss[j][1]*x23+rbcgauss[j][2]*x33;
643
644         getindex(ix,iy,iz,ploc,cl); //hash local Gauss point
645
646         if(ix<=cl.nx && ix>=-1 && iy<=cl.ny && iy>=-1 && iz<=cl.nz && iz
            >=-1){ //check if interaction possible
647
648             //barycentric coords w.r.t. the original triangle
649             calc_barycentric_coordinates(b1,b2,b3,ploc[0],ploc[1],ploc[2],r1
                [0],r2[0],r3[0],r1[1],r2[1],r3[1],r1[2],r2[2],r3[2]);
650             //velocity of local Gauss point
651             for(int m=0; m<3; ++m) vloc[m]=b1*v1[m]+b2*v2[m]+b3*v3[m];
652
653             //area of current sub-triangle
654             area = triangle_area(x11,x12,x13,x21,x22,x23,x31,x32,x33);
655
656             for(int c1=ix-1; c1<=ix+1; ++c1) // move through cell and all
                neighboring cells
657                 for(int c2=iy-1; c2<=iy+1; ++c2)
658                     for(int c3=iz-1; c3<=iz+1; ++c3)
659                         if(c1<cl.nx && c1>=0 && c2<cl.ny && c2>=0 && c3<cl.nz && c3
                            >=0){ //check if valid cell
660                             //move through all SPH particles in the cell
661                             temp=cl.cl[c1][c2][c3];
662                             while(temp){
663                                 num=temp->x; //current local SPH particle number
664                                 d=sqrt( (ploc[0]-x[num][0])*(ploc[0]-x[num][0])+(ploc[1]-
                                    x[num][1])*(ploc[1]-x[num][1])+(ploc[2]-x[num][2])*(
                                    ploc[2]-x[num][2]) );
665                                 if(d<=2.0*h){ //cut-off

```

```
666 //boundary traction/repulsion acc to Müller,Schrim,
        Teschner et al.: Interaction of fluids with
        deformable solids 2004
667 fabs = (2.0*h-d)*(2.0*h-d);
668 fabs = alpha*(fabs*fabs-beta*fabs);
669
670 //calculate force on SPH particle and add to force
        arrays
671 for(int q=0; q<3; ++q){
672     floc[q]=area*wgauss[j]*fabs*(x[num][q]-ploc[q])/d;
673     //if(0.5*hinv*d<0.1) std::cout << "Warning: SPH-wall
        distance d/rc < 1/10 encountered" << std::endl;
674
675     if(floc[q]!=floc[q]){
676         std::cout << "floc rep 3D: " << floc[q] << std::endl
        ;
677         MPI_Abort(MPI_COMM_WORLD,1);
678     }
679     if(f[num][q]!=f[num][q]){
680         std::cout << "f rep 3D: " << f[num][q] << std::endl;
681         MPI_Abort(MPI_COMM_WORLD,1);
682     }
683
684     f[num][q]+=floc[q];
685     //force distribution to original triangle vertices,
        see notes and MM file
686     f1[q]-=b1*floc[q];
687     f2[q]-=b2*floc[q];
688     f3[q]-=b3*floc[q];
689 }
690
691 //boundary friction / viscous terms acc. to: see above,
        but with normalized cubic spline kernel
692 fabs = d*hinv;
693
694 /* //for cubic spline:
695
696     if(fabs<=1.0)
697         fabs = 0.5*t*(-2.0+3.0*fabs);
698     else
699         fabs = 0.5*t*(2.0-fabs);
700 */
701
702 //for spiky kernel
703 fabs = 0.5*t*(2.0-fabs);
```

```

704
705         //calculate force on SPH particle and add to force
           arrays
706     for(int q=0; q<3; ++q){
707         floc[q]=area*wgauss[j]*fabs*(vloc[q]-v[num][q]);
708
709         /*if(vloc[q]!=vloc[q]){
710             std::cout << "vwall: " << vloc[q] << std::endl;
711             std::cin.get();
712         }
713         if(v[num][q]!=v[num][q]){
714             std::cout << "vSPH: " << v[num][q] << std::endl;
715             std::cin.get();
716         }*/
717     if(floc[q]!=floc[q]){
718         std::cout << "floc visc 3D: " << floc[q] << std:::
           endl;
719
720         //for debug
721         //std::cout << "(b1,b2,b3) = " << "(" << b1 << ", "
           << b2 << ", " << b3 << ")" << std::endl;
722         //std::cout << "vloc[0] = " << vloc[0] << " " << "
           vloc[1] = " << vloc[1] << " " << "vloc[2] = " <<
           vloc[2] << std::endl;
723         //std::cout << "v[num][0] = " << v[num][0] << " " <<
           "v[num][1] = " << v[num][1] << " " << "v[num][2]
           = " << v[num][2] << std::endl;
724         //std::cout << "fabs = " << fabs << " " << "wgauss[j
           ] = " << wgauss[j] << " " << "area = " << area <<
           std::endl;
725
726         MPI_Abort(MPI_COMM_WORLD,1);
727     }
728     if(f[num][q]!=f[num][q]){
729         std::cout << "f visc 3D: " << f[num][q] << std::endl
           ;
730         MPI_Abort(MPI_COMM_WORLD,1);
731     }
732
733     f[num][q]+=floc[q];
734     //force distribution to end points of line segment,
           see notes and MM file
735     f1[q]-=b1*floc[q];
736     f2[q]-=b2*floc[q];
737     f3[q]-=b3*floc[q];

```

```
738         }
739     }
740     temp=temp->next;
741 }
742 }
743 }
744 }
745
746 }
747
748 //calculates barycentric coordinates of point x=(p,q,r) in triangle with
       vertices r1=(x1,y1,z1),r2=(x2,y2,z2),r3=(x3,y3,z3) and saves data in b1,b2
       ,b3
749 //from Christer Ericson: Real-Time Collision Detection, San Francisco 2005
750 void calc_barycentric_coordinates(double& b1, double& b2, double& b3, double p
       , double q, double r, double x1, double x2, double x3, double y1, double
       y2, double y3, double z1, double z2, double z3){
751     double d00 = (x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1);
752     double d11 = (x3-x1)*(x3-x1)+(y3-y1)*(y3-y1)+(z3-z1)*(z3-z1);
753     double d01 = (x2-x1)*(x3-x1)+(y2-y1)*(y3-y1)+(z2-z1)*(z3-z1);
754     double d20 = (p-x1)*(x2-x1)+(q-y1)*(y2-y1)+(r-z1)*(z2-z1);
755     double d21 = (p-x1)*(x3-x1)+(q-y1)*(y3-y1)+(r-z1)*(z3-z1);
756     double denom = d00*d11 - d01*d01;
757     b2 = (d11*d20 - d01*d21)/denom;
758     b3 = (d00*d21 - d01*d20)/denom;
759     b1 = 1. - b2 - b3;
760 }
761
762 /*
763
764 //WORKS FINE, but slower?
765
766 //calculates barycentric coordinates of point x=(p,q,r) in triangle with
       vertices r1=(x1,y1,z1),r2=(x2,y2,z2),r3=(x3,y3,z3) and saves data in b1,b2
       ,b3
767 //acc to http://mathworld.wolfram.com/ArealCoordinates.html
768 void calc_barycentric_coordinates(double& b1, double& b2, double& b3, double p
       , double q, double r, double x1, double x2, double x3, double y1, double
       y2, double y3, double z1, double z2, double z3){
769     double a = triangle_area_sq(x1,y1,z1,x2,y2,z2,x3,y3,z3);
770     double a1 = triangle_area_sq(p,q,r,x2,y2,z2,x3,y3,z3);
771     double a2 = triangle_area_sq(p,q,r,x3,y3,z3,x1,y1,z1);
772     b1 = sqrt(a1/a);
773     b2 = sqrt(a2/a);
774     b3 = 1.-b1-b2;
```

```

775 }
776
777 */
778
779 /* prior versions - may yield 1/0 cases */
780 /*
781 //calculates barycentric coordinates of point x=(p,q,r) in triangle with
      vertices r1=(x1,y1,z1),r2=(x2,y2,z2),r3=(x3,y3,z3) and saves data in b1,b2
      ,b3
782 void calc_barycentric_coordinates(double& b1, double& b2, double& b3, double p
      , double q, double r, double x1, double x2, double x3, double y1, double
      y2, double y3, double z1, double z2, double z3){
783 //treat the case of all the points lying within xy,xz,yz layer independently
784 if(x1==0 && x2==0 && x3==0){
785 calc_barycentric_coordinates_yzlayer(b1,b2,b3,q,r,y1,y2,y3,z1,z2,z3);
786 //for debug
787 if(b1!=b1 || b2!=b2 || b3!=b3)
788 std::cout << "yz layer, (b1,b2,b3) = " << "(" << b1 << ", " << b2 << ", " <<
      b3 << ")" << std::endl;
789 }
790 else if(y1==0 && y2==0 && y3==0){
791 calc_barycentric_coordinates_xzlayer(b1,b2,b3,p,r,x1,x2,x3,z1,z2,z3);
792 //for debug
793 if(b1!=b1 || b2!=b2 || b3!=b3)
794 std::cout << "xz layer, (b1,b2,b3) = " << "(" << b1 << ", " << b2 << ", " <<
      b3 << ")" << std::endl;
795 }
796 else if(z1==0 && z2==0 && z3==0){
797 calc_barycentric_coordinates_xyayer(b1,b2,b3,p,q,x1,x2,x3,y1,y2,y3);
798 //for debug
799 if(b1!=b1 || b2!=b2 || b3!=b3)
800 std::cout << "xy layer, (b1,b2,b3) = " << "(" << b1 << ", " << b2 << ", " <<
      b3 << ")" << std::endl;
801 }
802 else{
803 double n = x3*y2*z1 - x2*y3*z1 - x3*y1*z2 + x1*y3*z2 + x2*y1*z3 - x1*y2*z3;
804 b1 = (r*x3*y2 - r*x2*y3 - q*x3*z2 + p*y3*z2 + q*x2*z3 - p*y2*z3)/n;
805 b2 = -(r*x3*y1 - r*x1*y3 - q*x3*z1 + p*y3*z1 + q*x1*z3 - p*y1*z3)/n;
806 b3 = 1.0 - b1 - b2;
807 //for debug
808 if(b1!=b1 || b2!=b2 || b3!=b3){
809 std::cout << "regular, (b1,b2,b3) = " << "(" << b1 << ", " << b2 << ", " << b3
      << ")" << std::endl;
810 std::cout << "r1 = " << "(" << x1 << ", " << y1 << ", " << z1 << ")" << std:::
      endl;

```

```
811 std::cout << "r2 = " << "(" << x2 << ", " << y2 << ", " << z2 << ")" << std::
    endl;
812 std::cout << "r3 = " << "(" << x3 << ", " << y3 << ", " << z3 << ")" << std::
    endl;
813 }
814 }
815 }
816
817 //calculate barycentric coordinates for points in yz-layer; cf. also
    calc_barycentric_coordinates
818 void calc_barycentric_coordinates_yzlayer(double b1,double b2,double b3,double
    q,double r,double y1,double y2,double y3,double z1,double z2,double z3){
819 b1 = (r*y2 - r*y3 - q*z2 + y3*z2 + q*z3 - y2*z3)/(y2*z1 - y3*z1 - y1*z2 + y3*
    z2 + y1*z3 - y2*z3);
820 b2 = (r*(-y1 + y3) - y3*z1 + q*(z1 - z3) + y1*z3)/(y3*(-z1 + z2) + y2*(z1 - z3
    ) + y1*(-z2 + z3));
821 b3 = 1 - b1 - b2;
822 }
823
824 //calculate barycentric coordinates for points in xy-layer; cf. also
    calc_barycentric_coordinates
825 void calc_barycentric_coordinates_xyayer(double b1,double b2,double b3,double
    p,double q,double x1,double x2,double x3,double y1,double y2,double y3){
826 b1 = (q*x2 - q*x3 - p*y2 + x3*y2 + p*y3 - x2*y3)/(x2*y1 - x3*y1 - x1*y2 + x3*
    y2 + x1*y3 - x2*y3);
827 b2 = (q*(-x1 + x3) - x3*y1 + p*(y1 - y3) + x1*y3)/(x3*(-y1 + y2) + x2*(y1 - y3
    ) + x1*(-y2 + y3));
828 b3 = 1 - b1 - b2;
829 }
830
831 //calculate barycentric coordinates for points in xz-layer; cf. also
    calc_barycentric_coordinates
832 void calc_barycentric_coordinates_xzlayer(double b1,double b2,double b3,double
    p,double r,double x1,double x2,double x3,double z1,double z2,double z3){
833 b1 = (r*x2 - r*x3 - p*z2 + x3*z2 + p*z3 - x2*z3)/(x2*z1 - x3*z1 - x1*z2 + x3*
    z2 + x1*z3 - x2*z3);
834 b2 = (r*(-x1 + x3) - x3*z1 + p*(z1 - z3) + x1*z3)/(x3*(-z1 + z2) + x2*(z1 - z3
    ) + x1*(-z2 + z3));
835 b3 = 1 - b1 - b2;
836 }
837 */
838
839 //area of a triangle with vertices r1,r2,r3
840 double triangle_area(const double* r1, const double* r2, const double* r3){
841     return 0.5*sqrt(
```

```

842     ((r1[1]-r2[1])*(r1[2]-r3[2])-(r1[1]-r3[1])*(r1[2]-r2[2]))*((r1[1]-r2[1])*(
      r1[2]-r3[2])-(r1[1]-r3[1])*(r1[2]-r2[2])) +
843     ((r1[0]-r3[0])*(r1[2]-r2[2])-(r1[0]-r2[0])*(r1[2]-r3[2]))*((r1[0]-r3[0])*(
      r1[2]-r2[2])-(r1[0]-r2[0])*(r1[2]-r3[2])) +
844     ((r1[0]-r2[0])*(r1[1]-r3[1])-(r1[0]-r3[0])*(r1[1]-r2[1]))*((r1[0]-r2[0])*(
      r1[1]-r3[1])-(r1[0]-r3[0])*(r1[1]-r2[1]))
845     );
846 }
847
848 //area of a triangle with vertices r1=(x11,x12,x13),r2=(x21,x22,x23),r3=(x31,
      x32,x33)
849 double triangle_area(double x11, double x12, double x13, double x21, double
      x22, double x23, double x31, double x32, double x33){
850     return 0.5*sqrt(
851         ((x12-x22)*(x13-x33)-(x13-x23)*(x12-x32))*((x12-x22)*(x13-x33)-(x13-x23)*(
          x12-x32)) +
852         ((x11-x31)*(x13-x23)-(x11-x21)*(x13-x33))*((x11-x31)*(x13-x23)-(x11-x21)*(
          x13-x33)) +
853         ((x11-x21)*(x12-x32)-(x11-x31)*(x12-x22))*((x11-x21)*(x12-x32)-(x11-x31)*(
          x12-x22))
854     );
855 }
856
857 //square of area of a triangle with vertices r1=(x11,x12,x13),r2=(x21,x22,x23)
      ,r3=(x31,x32,x33)
858 double triangle_area_sq(double x11, double x12, double x13, double x21, double
      x22, double x23, double x31, double x32, double x33){
859     return 0.25*(
860         ((x12-x22)*(x13-x33)-(x13-x23)*(x12-x32))*((x12-x22)*(x13-x33)-(x13-x23)*(
          x12-x32)) +
861         ((x11-x31)*(x13-x23)-(x11-x21)*(x13-x33))*((x11-x31)*(x13-x23)-(x11-x21)*(
          x13-x33)) +
862         ((x11-x21)*(x12-x32)-(x11-x31)*(x12-x22))*((x11-x21)*(x12-x32)-(x11-x31)*(
          x12-x22))
863     );
864 }

```

## A.12. wrapper code LINUX.cpp

```

1  #include "dn.h"
2  #include "exchange_class_Linux.h"
3  #include "mpi.h"
4  #include <cassert>
5  #include <iostream>
6  #include <sstream>

```

```
7 #include <sys/time.h>
8 #include <string>
9 using namespace std;
10
11 int str2int (const string &str) {
12     stringstream ss(str);
13     int n;
14     ss >> n;
15     return n;
16 }
17
18 void coupling(DataL& obj, int proc){
19
20     obj.getrohSPH(); // set initial densities
21     obj.set_timestep(); //get timestep
22     //cout << "time step read" << endl;
23
24     timeval t1,t2,res;
25     double tin=0.0;
26     gettimeofday(&t1,NULL);
27
28     obj.readone(); //initializing steps for equilibration with very high
        viscosity
29     obj.readone(); // actual viscosity
30     obj.readone(); //"run 0" to recalculate forces for new viscosity
31
32     gettimeofday(&t2,NULL);
33     timersub(&t2,&t1,&res);
34     tin+=res.tv_sec*1000.0; //add consumed time in ms
35     tin+=res.tv_usec/1000.0;
36
37     //cout << endl << tin << endl;
38
39     double ctime=0.0;
40     int count = 0;
41
42     while(1){
43
44         obj.set_timestep(); //get timestep
45         if(obj.recv_command()==0){ //get r,v or dummy
46             if(proc==0) cout << "received unknown command from server" << endl;
47             break;
48         }
49
50         //run 1 time step "run 1 pre no post no"
```

```
51     if(proc == 0) gettimeofday(&t1,NULL);
52     if(!obj.readone()){
53         break;
54     }
55     if(proc == 0){
56         gettimeofday(&t2,NULL);
57         timersub(&t2,&t1,&res);
58         ctime+=res.tv_sec*1000.0; //add consumed time in ms
59         ctime+=res.tv_usec/1000.0;
60         ++count;
61     }
62
63     if(obj.recv_command()==0){ //send force or force,rSPH,vSPH,rohSPH,...
64         if(proc==0) cout << "received unknown command from server" << endl;
65         break;
66     }
67     if(obj.recv_command()==0){ //send force or dummy
68         if(proc==0) cout << "received unknown command from server" << endl;
69         break;
70     }
71
72 }
73
74 if(proc == 0){
75     cout << "total time of FSI calculation (in ms): " << ctime << " time per
76         timestep: " << ctime/count << endl;
77     cout << "time of initializing step(s) - LIGGGHTS only: " << tin << endl;
78 }
79 }
80
81 int main(int argc, char* argv[]){
82
83     if(argc < 3){
84         cout << "To run this client, enter: mpirun -np x name ip port [liggghts
85             command line args]" << endl << "x ... number of procs" << endl <<
86             "name ... name of the client executable" << endl << "e.g.: mpirun -np 4
87             main 192.168.56.1 12345 -log none -screen none" << endl;
88     }
89
90     MPI_Finalize();
91     return 0;
92 }
93
94 MPI_Init(&argc,&argv);
95
96 int proc,nprocs;
```

```
93 MPI_Comm_rank(MPI_COMM_WORLD,&proc); //ESSENTIAL CALLS!! even if information
    is not used explicitly
94 MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
95
96 // std::string a1,a2,a3,a4,
97 std::string temp1;
98 short port;
99
100 if(proc == 0){
101     /*
102     cout << "Server-IP a1.a2.a3.a4 (IPv4): " << endl << "a1: " << endl;
103     cin >> a1;
104     cout << "a2: " << endl;
105     cin >> a2;
106     cout << "a3: " << endl;
107     cin >> a3;
108     cout << "a4: " << endl;
109     cin >> a4;
110     cout << "Port: " << endl;
111     cin >> port;
112     */
113     // a1="192"; a2="168"; a3="12"; a4="34"; port=12345;
114     // a1="140"; a2="78"; a3="135"; a4="130"; port=12345;
115
116     //temp1.append(a1).append(".").append(a2).append(".").append(a3).append
        (".").append(a4);
117     temp1 = (string)argv[1];
118     port = (short)str2int((string)argv[2]);
119 }
120
121 while(1){
122     if(argc > 3){
123         DataL obj(&temp1,&port,argc-2,argv+2); //argv+2 ... pointer arithmetics
124         coupling(obj,proc);
125     }
126     else{
127         DataL obj(&temp1,&port);
128         coupling(obj,proc);
129     }
130 }
131 }
132
133 MPI_Finalize();
134 return 0;
135 }
```