UNIVERSITY OF APPLIED SCIENCE MÜNSTER

MASTER'S THESIS

Implementation of a viscoelastic fluid model to simulate single bubbles with the surface-tracking method

by PAUL-JASPER SAHR

April 7, 2020

Research at:
Berlin Institut of Technology
Chair of Chemical & Process Engineering

Wissenschaftliche Leitung: PROF. DR.-ING. M. KRAUME
Erstprüfung: PROF. DR.-ING. M. ALTENDORFNER
Wissenschaftliche Betreuung und Zweitprüfung: M.SC. F. ENDERS

Declaration of Authorship

Hiermit versichere ich, dass ich diese Arbeit selbstständig angefertigt habe und die benutzten Hilfsmittel/Quellen angegeben worden sind.

Berlin, 07.04.2020

_____

Paul-Jasper Sahr

Diese Abschlussarbeit wurde dem Prüfungsausschuss des Fachbereiches Chemieingenieurwesen vorlegt am:

_____

Abstract

The simulation of a free rising bubble in a viscoelastic fluid is a fundamental problem. Although many fluids of technical interest show viscoelastic fluid behaviour, a profound understanding of the behaviour of viscoelastic fluids is still lacking. Previous studies have simulated single bubbles in a viscoelastic fluid through with solvers using a volume-of-fluid solver of OpenFOAM, an object-oriented C++ library for simulations in continuum mechanics. Using a surface-tracking method will allow a sharp representation of the interface due to a moving computational mesh, which deforms according to the interface motion. However, a free rising bubble in a viscoelastic fluid has still not been simulated successfully as the description of the boundary conditions at the interface is difficult and requires special attention. In this work, the underlying mathematical model is modified viscoelastically through the implementation of an extra stress tensor in the code base of the existing bubbleInterTrackFoam solver. In addition, the viscoelasticModel library is modified and used as a foundation for the calculation of the viscoelastic stress tensor using the Phan-Thien–Tanner constitutive model. Pseudo 2D simulations with parameters corresponding to water are performed and compared to simulations with the original solver. Although the new solver did not reproduce the expected results, more pseudo 2D simulations showed that the implemented viscoelastic model reacts as expected while the viscoelastic model parameters are small. However, with higher degrees of viscoelasticity unexpected results are received and the implementation failed, most likely due missing information about the choice of appropriate boundary conditions for the stress tensor at the interface. Furthermore, 3D simulations were performed with the original solver and varying numerical parameters, all of which crashed after unexpectedly short run-times. The biggest positive effect on the run-times showed the alteration of the MRF properties with an optimum value of 0.5. This is already a step forward and the code of this current work and suggestions for improvement can be used for further research and adjustments regarding free rising bubble simulations in a viscoelastic fluid.

Contents

List of Figures

List of Tables

List of Code Samples

List of Abbreviations

ALE	Arbitrary Langrarian-Eulerian
BICGStab	Biconjugate Gradient Stabilised Method
CFD	Computational Fluid Dynamics
CUBISTA	Convergent and Universally Bounded Interpolation Scheme for the Treatment of Advection
DIC	Simplified Diagonal-based Incomplete Cholesky preconditioner
DILU	Diagonal Incomplete-LU method
FA	Finite Area
FAM	Finite Area Method
FV	Finite Volume
FVM	Finite Volume Method
ILU0	Incomplete-LU with no fill-ins Method
MAC	Marker And Cells Method
MPI	Message Passing Interface
MRF	Moving Reference Frame
HLRN	Nordeutscher Verbund für Hoch- und Höchstleistungsrechnen
OpenFOAM	Open-Source Field Operation and Manipulation
PISO	Pressure-Implicit with Splitting of Operators
PCG	Preconditioned Conjugate Gradients Method
SIMPLE	Semi-Implicit Method of Pressure-Linked Equations
VOF	Volume-Of-Fluid Method

List of Symbols

Latin letters:

$\mathbf{D}$	Deformation gradient tensor	s^{-1}
$\mathbf{g}$	Gravitational acceleration	$\mathrm{m\,s}^{-2}$
$\mathbf{I}$	Unit tensor	1
L	Arc length along ∂S	m
$\mathbf{m}$	Outer unit binormal on ∂S	1
$\mathbf{n}$	Unit normal vector	1
$\mathbf{n}_f$	Unit normal vector on cell face	1
P	Total pressure	$\mathrm{N\,m}^{-2}$ ($\mathrm{kg\,m}^{-1}\,\mathrm{s}^{2}$)
p	Dynamic pressure	$\mathrm{N\,m}^{-2}$ ($\mathrm{kg\,m}^{-1}\,\mathrm{s}^{2}$)
$\mathbf{r}$	Position vector	m
S	Surface area	m^2
t	Time	s
V	Volume	m
$\mathbf{v}$	Velocity vector	$\mathrm{m\,s}^{-1}$
$\mathbf{v}_f$	Velocity on cell face vector	$\mathrm{m\,s}^{-1}$
$\mathbf{v}_P$	Velocity in cell centroid vector	$\mathrm{m\,s}^{-1}$
$\mathbf{v}_S$	Velocity of the interface vector	$\mathrm{m\,s}^{-1}$
$\mathbf{v}_t$	Tangential component of the velocity vector	$\mathrm{m\,s}^{-1}$

Greek letters:

ϵ	Rate of extension	s^{-1}
κ	Twice mean curvature of the free-surface	1
λ	Relaxation time	s
μ_p	Constant dynamic polymer viscosity coefficient	s ($\mathrm{kg\,m}^{-1}\,\mathrm{s}$)
μ_s	Dynamic solvent viscosity	Pa s ($\mathrm{kg\,m}^{-1}\,\mathrm{s}$)
$\boldsymbol{\nu}_n$	Normal component of the velocity vector	$\mathrm{m\,s}^{-1}$
$\mathbf{\Pi}$	Total stress tensor	Pa ($\mathrm{N\,m}^{-2}$)
φ	Volume fraction	1
ρ	Fluid density	$\mathrm{kg\,m}^{-3}$
σ	Surface tension coefficient	$\mathrm{N\,m}^{-1}$ ($\mathrm{kg\,s}^{-2}$)
$\boldsymbol{\tau}$	Total viscous stress tensor	Pa ($\mathrm{N\,m}^{-2}$)
$\boldsymbol{\tau}_f$	Total viscous stress tensor at the cell face	Pa ($\mathrm{N\,m}^{-2}$)
$\boldsymbol{\tau}_p$	Extra elastic stress tensor	Pa ($\mathrm{N\,m}^{-2}$)
$\boldsymbol{\tau}_s$	Viscous stress tensor	Pa ($\mathrm{N\,m}^{-2}$)
$\overset{\triangledown}{\boldsymbol{\tau}}_p$	Oldroyd's upper-convected derivative	Pa ($\mathrm{N\,m}^{-2}$)
ζ	Constant Phan-Thien–Tanner model parameter	1

Chapter 1

Introduction

The possibilities of predicting and calculating physical phenomena have fundamentally changed with the transformation of an analogue world into a digitalised age. Nowadays, modern simulation software collect holistic data and information, which was done prior to the invention of the computer by often expensive apparatuses and test stands.

The subject of computational fluid dynamics (CFD) is the description and prediction of problems involving fluid flows by using numerical software. Decreasing costs for computational power and evolving cloud-computing technologies have resulted in numerous new fields of application for fluid simulations. The numerical description of gas-liquid multi-phase processes is one of these current research fields. As of now, a reliable description of these processes with new, complex methods is often only possible in Newtonian media, while many fluids of industrial interest show non-Newtonian behaviour. This makes the further development of methods for the simulation of these media essential.

Viscoelasticty is a kind of non-Newtonian behaviour that combines characteristics of viscous fluids and elastic solids. While viscoelastic fluids behave viscous, they also store energy for a certain amount of time in form of stress. This special type of time-depended rheological behaviour is for example observed in polymer melts and solutions.

The simulation of a free rising bubble in a viscoelastic fluid is a very fundamental problem, which allows to gain a more profound understanding of the behaviour of viscoelastic fluids. It is known that the viscoelastic properties influence the flow field and bubble shape. This fundamental knowledge of the simulation of viscoelastic multi-phase systems can then be up-scaled for a better prediction of technical processes.

The subject of this thesis is the implementation of a viscoelastic fluid model in a surface-tracking solver for OpenFOAM. As a basis for the implementation of a viscoelastic model, a solver for the simulation of free rising single bubble in Newtonian fluids developed by TUKOVIĆ [2005] is used.

The solver is combined with a foam-extend library for the calculation of the viscoelastic stress tensor in order to simulate a bubble in a viscoelastic fluid. One of the major difficulties during the simulation of free surfaces is the description of the boundary conditions at the interface. In the method proposed by TUKOVIĆ, these are calculated through a force equilibrium at the interface and updated every iteration step. Before the implementation in the code, this force balance is modified viscoelastically and afterwards simulations for validation are performed. These simulations are performed with a pseudo 2D mesh. Furthermore, a 3D geometry is created and meshed to perform simulations.

This work may also serve as an introduction in the field of free surface simulation with the surface tracking method. Although the surface-tracking model is found in more advanced publications, the link between the numerical model and its underlying code is not clear. Therefore, the aim is to provide a more detailed explanation of the model fundamentals and the derivation of the mathematical background.

1.1 Objectives

The aim of this work is the simulation of a free rising single bubble in a viscoelastic fluid through the following steps:

1. Modify the mathematical model of the surface-tracking solver in order to implement a viscoelastic fluid model.
2. Explain the implementation of the viscoelastic model through several code samples.
3. Test the new developed solver with simulations on a pseudo 2D mesh.
4. Perform simulations with the original and the new solver on 3D meshes.

Chapter 2

Fundamentals

2.1 Computational Fluid Dynamics

The CFD is an established method for the numerical analysis of problems involving fluid flow. Using the Navier-Stokes-Equations, the method is used to solve fluid flow, heat transfer and associated problems in an approximative way. The basis of any numerical flow analysis is formed by the conservation laws of momentum, mass and energy. However, in the case of isothermal calculation, only the conservation of momentum and mass are required for the numerical flow analysis. The momentum equation derived by NAVIER and STOKES to describe the motion of viscous substances is given in a general convective form by Equation 2.1 below:

$$\rho \frac{D\mathbf{v}}{Dt} = -\boldsymbol{\nabla} p + \boldsymbol{\nabla} \cdot \boldsymbol{\tau} + \rho \mathbf{g} \tag{2.1}$$

with the density ρ, the velocity vector $\mathbf{v}$, the time t, the pressure p, the total viscous stress tensor $\boldsymbol{\tau}$ and the gravitational acceleration $\mathbf{g}$. The left side of the equation describes acceleration of the volume element. The right side of the equation represents pressure, viscous and gravitational forces acting on the volume element.

The conservation of mass is described through the so-called continuity equation for fluid dynamics given by the equation below. The equation states that mass is conserved in any non-nuclear continuum mechanics analysis:

$$\frac{D\rho}{Dt} + \rho(\boldsymbol{\nabla} \cdot \mathbf{v}) = 0. \tag{2.2}$$

In the early days of fluid dynamics, analytical solution were possible only for strongly simplified problems, and mathematicians are still searching today for a general solution of the Navier-Stokes-Equation. In the beginning of numerical flow simulation only the solution of fairly simple flow problems and geometries was possible. Since scientists solved first differential equations describing flow problems with computational support, the applications for CFD became more complex over the years. Nowadays, three dimensional, turbulent flows in complex, or even moving geometries, do not pose a challenge for commercial software with the Finite Volume Method (FVM) to evaluate the partial differential equations. Solution approaches for many different problems have been established but there are still areas in numerical flow simulation that do not have standard solutions and are in the focus of current research. These includes for example, the simulation of processes with supercritical fluids for the precipitation of nano particles, the

simulation of aero-acoustics or the development of advanced algorithms in CFD [Cardoso et al. 2018; Sadrehaghighi 2020]. Another field of research that is a subject in this thesis, is the simulation of free surface, multi-phase flows. In the course of time different methods for the simulation of free surface flows have been developed. The following sections are giving a brief introduction into the FVM and methods of free surface modelling, presenting some general approaches and a deeper look at the *surface-tracking method*.

2.1.1 Finite Volume Method

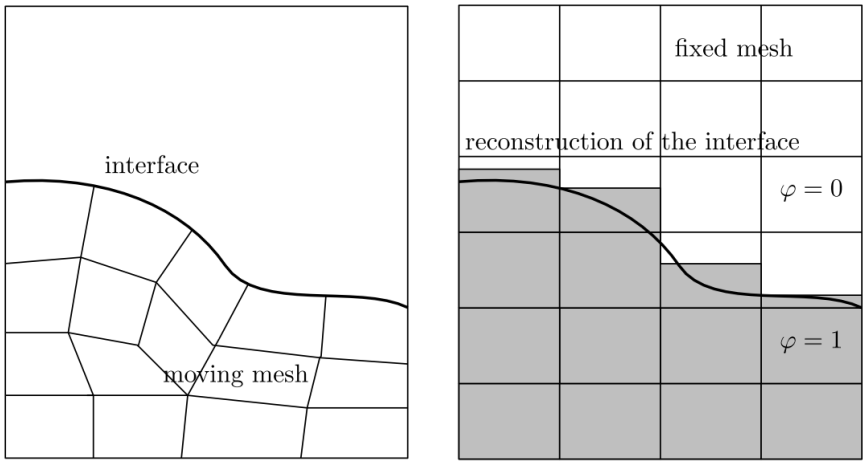
The FVM is the most commonly used discretisation method in fluid mechanics. It is a method for the approximation of the partial differential equations of the conservation laws. The continuous fluid is discretised through a mesh into a finite amount of control volumes, which can have any polygonal or polyhedral shape. Therefore, even complicated geometries can be meshed easily. The transport equations are then solved through integration over each control volume approximately. The volume integrals are transformed into surface integrals using the Gauss's theorem. With FVM, the convective and diffusive fluxes on the surfaces of each cell are evaluated explicitly, which gives it a great advantage over other numerical methods (FDM or FEM) in CFD because the conservation of fluxes at the cell boundaries is fundamental [Schwarze 2013].

2.1.2 Free Surface Modelling

Interfacial multi-phase flows appear nowadays in many engineering or mathematical problems. In the last years, many models and numerical methods have been developed to encounter problems such as bubble transport, extraction, chemical reaction, mass-transfer, separation and other interfacial flow involving problems. However, not every multi-phase simulation requires an exact representation of the interface. In many technical applications with a bigger score, it is often sufficient to simulate the multi-phase region as an approximation (e.g. as a bubble region) and formulate mixed properties for this region. For the exact description of the microscopic phenomena on the free surface however, an approximation as a multi-phase region is not sufficient.

In order to guarantee a precise description of the conditions at the free surface, an exact description of the interface is necessary [Gopala and Wachem 2008]. Discontinuities across the moving and deforming interface are part of the nature of these problems and still represent a great challenge in simulation [Quan and Schmidt 2007]. Even with decreasing computational costs, free surface simulations are still very demanding due to the highly nonlinear characteristics of these problems. In the field of CFD, many numerical methods for free surface modelling have been proposed over the years, each having advantages and disadvantages on their own.

There are two ways to characterise a flow mathematically: The Euler and the Lagrange approach. In the Eulerian methods, a coordinate system is defined in a fixed grid dividing the flow field into a finite number of cells. Each cell is assigned properties, such as a velocity, according to that fixed coordinates.



(A) Lagrangian methods (B) Eulerian methods

FIGURE 2.1: Two categories of models for the tracking of the free surface [Caboussat 2005]

The Lagrangian methods are based on the displacement of a system of coordinates at each point. Here, every point of the flowing domain is moved with the velocity of each individual fluid element. In the description of free surface flows, this refer also to the displacement of the interface between the two phases making a sharp representation of the interface possible.

Figure 2.1 shows a Langrarian and a Eulerian method for the simulation of a free surface flow. Eulerian methods like the *volume-of-fluid* (VOF) method are a well established and fairly popular for the capability of adopting to difficult topologies. Disadvantageous is the loss of detail with regard to the interface representation as shown in Figure 2.1b. In addition to fluid properties, every cell has a scalar φ assigned that defines the volume fraction of the cell. This value is used to distinguish between the phases and calculate mixed properties for the boundary cells.

Other valuable methods for free surface simulations are the Eulerian *level-set* method and the arbitrary Langrarian-Eulerian *marker and cells* (MAC) particle method [Caboussat 2005]. Arbitrary Langrarian-Eulerian (ALE) methods are a combination of both approaches eliminating many disadvantages. The ALE surface-tracking method used in this work is discussed in detail in the section below.

2.1.3 Surface-Tracking Method

The procedure used in this work to simulate a free rising bubble contains a surface-tracking method (or interface-tracking method) based on a finite volume discretisation with a moving computational mesh and automatic mesh motion [Tuković and Jasak 2008]. A key characteristics of free surface problems are moving contact lines for the fluid parts. This leads to problems when using a traditional CFD approach with a fixed (Eulerian) grid. The ALE method is an approach to overcome this difficulty. The grid is moved arbitrary inside the fluid domain following the movement of the boundary [Kassiotis 2008]. While the transport equations are written in Eulerian coordinates, the

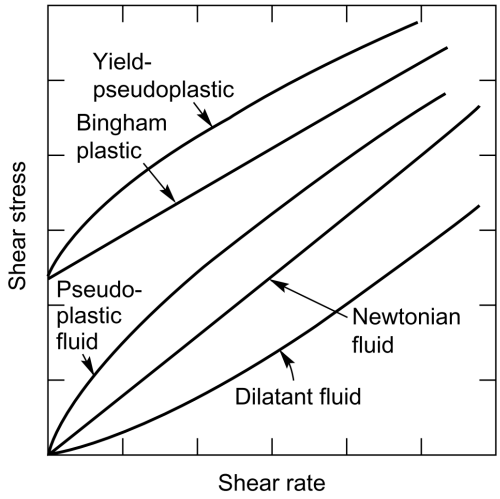


FIGURE 2.2: Types of time-independent flow behaviour [Chhabra and Richardson 1999]

structure dynamics are typically described in a Lagrangian frame of reference [Caboussat 2005]. With a surface-tracking method, a separate mesh is applied for each phase while the motion of the free surface is obtained from the velocity boundary conditions at the interface. The computational mesh is adjusted to the shape of the boundary, which is updated every step of the transient simulation. In order to correctly predict the motion of the boundary vertices, an extra partial differential equation (see Section 3.2.3: Space conservation law) is solved [OpenFOAMWiki 2020a]. The advantages of this approach over interface capturing methods like the VOF method are a precise modelling of the free surface and a sharp interface. In addition, numerical smearing is avoided and problems with extremely high surface tensions can be solved. Drawbacks are that only small changes in topology can be handled if no remeshing is applied. Problems can also occur for small density ratios of the phases. Maintaining the mesh quality is one of the main difficulties in tackling cases with variable geometry [OpenFOAMWiki 2020a; H. Jasak 2005; Jasak and Tuković 2006].

2.2 Viscoelasticity

Fluids can be differentiated into *Newtonian* and *non-Newtonian* fluids. Newtonian fluids are characterised by a constant ratio of shear stress to shear rate, whereas non-Newtonian fluids are those whose flow curves (shear stress versus shear rate) are non-linear or do not pass through the origin [Wang 2017]. Some of the most common time-independent non-Newtonian, as well as Newtonian, behaviours are shown in Figure 2.2. While Newtonian fluid show a constant viscosity, non-Newtonian fluid show a shear rate dependent viscosity.

(A) Elastic materials (B) Viscous materials (C) Viscoelastic materials

FIGURE 2.3: Material response to mechanical deformation [Barriga and Mayor 2019]

Viscoelasticity belongs to the category of time-dependent non-Newtonian behaviour and is exhibited by many materials of interest such as polymer melts or polymer solutions. The term viscoelastic refers to the partially viscous and partially elastic characteristics of the material when undergoing deformation. If an ideal solid is deformed elastically, it regains its original form when the stress is removed by releasing the stored energy. The other extreme: If an ideal viscous fluid is deformed, the fluid will flow irreversibly due to its viscosity. A viscoelastic fluid combines both characteristics of an elastic solid an a viscous fluid (Figure 2.3) [Chhabra and Richardson 1999].

2.3 OpenFOAM

Open-source **F**ield **O**peration **A**nd **M**anipulation (OpenFOAM) an open-source library in the object-oriented C++ language for simulations in continuum mechanics. Providing second-order accurate FV discretisation with polyhedral mesh support, second-order discretisation in time, efficient linear system solvers and support for massively parallel computing [Tuković and Jasak 2012]. OpenFOAM has a large user base across most areas of engineering and science, where it is used in both academic and industrial settings. The features of the library range from complex fluid flows involving chemical reactions, turbulence and heat transfer, to acoustics, solid mechanics and electromagnetics. Its unique selling is the integration of pre-/ post-processing utilities and the possibility to develop customised numerical solvers and modifications due to the open access source code [OpenFOAM® 2020a]. The *bubbleInterTrackFoam* solver used in this work is part of foam-extend-4.0, a community edition based on OpenFOAM 1.6.

2.4 Solver: bubbleInterTrackFoam

In this section will discuss the background of the *bubbleInterTrackFoam* solver of foam-extend-4.0 used in this work. Explanations about the solver directory and remarks to the most important files and folder will be given. TUKOVIĆ [2005] developed this solver against the background of surfactant influences on the bubble motion. Since this will not

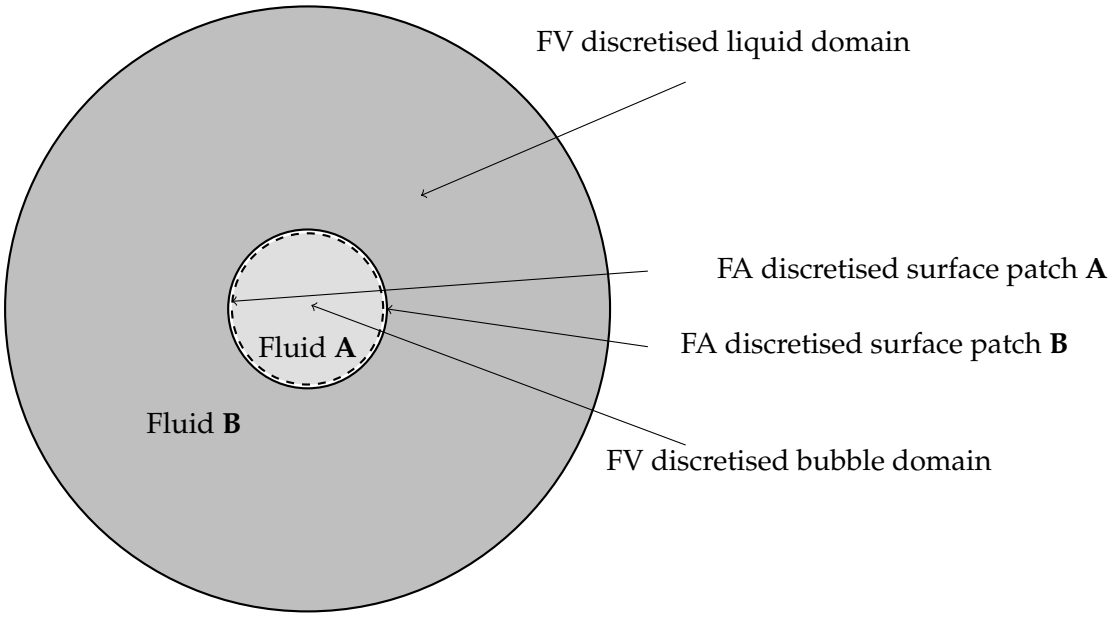


FIGURE 2.4: Domain sketch

be considered in this work, some parts and files of the solver remain unused and will not be further discussed.

2.4.1 Solver properties

The *bubbleInterTrackFoam* solver is using an incompressible, transient FVM for the simulation of a free rising bubble. The second-order accurate 3D moving mesh interface-tracking method is using a vertex-based mesh motion solver. The calculation of proper conditions at the free surface is essential and requires special attention. The preservation of the kinematic and dynamic condition (see Equation 3.27 and Equation 3.14 in Section 3.4.1) on the surface are key steps in the calculation of interfacial properties. This is carried out by a semi-implicit iterative solution procedure. In order to solve the force balance at the interface, which follows from the dynamic condition, it is necessary to calculate derivatives of the velocity field at the interface. This is obtained through the Finite Area Method (FAM) and a curved surface mesh that spans the bubble. The FA mesh consists of two congruent meshes with a zero thickness interface in between, as it is shown in Figure 2.4 [Tuković and Jasak 2012; Tuković and Jasak 2008]. A Moving Reference Frame (MRF) is used, where the space domain is moved along with the rising bubble. The velocity of the MRF is adjusted so that the bubble is kept centred within the domain. This allows a minimal domain size and the application of local mesh refinements in the vicinity of the bubble [Rusche 2002].

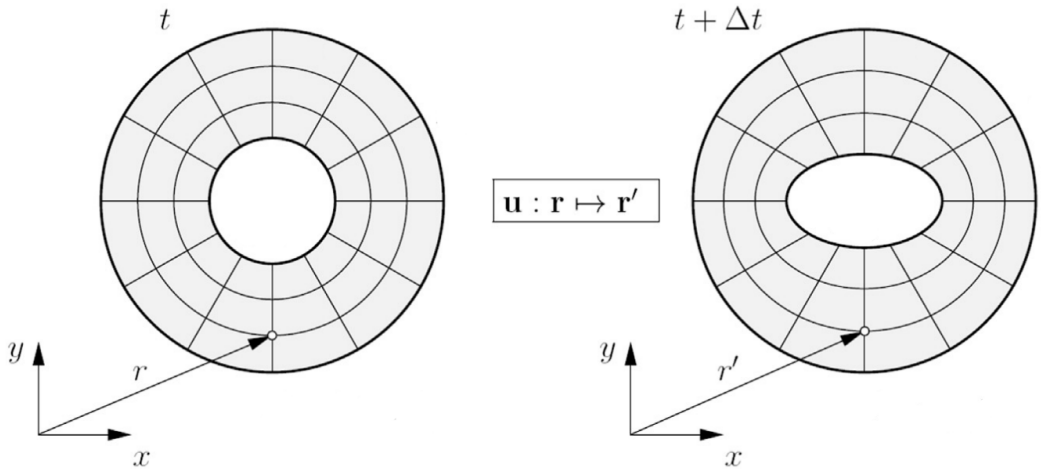


FIGURE 2.5: Mesh deformation [Tuković 2005]

Solution algorithm of the flow field

The evaluation of the transport equations is not straightforward as there is no explicit solution for the pressure [OpenFOAMWiki 2020c]. To overcome this problem, the equations are discretised semi-implicitly with the coupling being handled by a combination of the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) and the Pressure-Implicit with Splitting of Operators (PISO) method. The combination of SIMPLE and PISO method is often referred as PIMPLE method. First, the momentum equation is solved yielding an approximation of the velocity field. The pressure gradient term is then calculated using the pressure distribution from the previous iteration or an initial guess. The pressure equation is formulated and solved in order to obtain the new pressure distribution. Finally, the velocities are corrected and a new set of conservative fluxes is calculated. The method is called semi-implicit as the discretised momentum equation and pressure correction equation are solved implicitly, while the velocity correction is solved explicitly [Tuković and Jasak 2008; OpenFOAM® 2020a; Issa 1986].

Automatic mesh motion

The vertex-based mesh motion solver for polyhedral meshes calculates the motion of internal points based on the prescribed motion of interface points [Tuković and Jasak 2012]. During the simulation, maintaining the mesh quality defined by boundary mesh faces and control points is crucial. In order to preserve the smoothness of the interface during the calculation, the approach proposed by MUZAFERIJA and PERIĆ [1997] is used. The displacement of the free surface is embedded in the PIMPLE procedure, so that at the end of each external iteration, the displacement of the control points is calculated (schematic in Figure 2.5) [Tuković 2005; H. Jasak 2005].

2.4.2 Solution procedure

TUKOVĆ and JASAK [2012] are giving an overview of the solution procedure of the interface tracking method for a Navier-Stokes system on a moving mesh. According to them, the procedure of a two-phase fluid flow simulation consists of the following steps:

1. *For the new time instance $t = t^n$, initialise the values of velocity, pressure and mass fluxes with the corresponding values from the previous time instance. The initial net mass fluxes through the faces at the interface are calculated with the corresponding face volume fluxes obtained using positions of the interface mesh points from the previous time step.*
2. *Define displacement directions for the interfacial mesh points and the control points.*
3. *Start of the outer iteration loop:*
   (a) *In order to compensate the net mass flux through the interface, calculate displacement of the interface mesh points* [...].
   (b) *Displacement of the interface mesh points is used as a boundary condition for the solution of the mesh motion problem. After mesh movement, the new face volume fluxes are calculated* [...]
   (c) *Update pressure and velocity boundary conditions at the interface.*
   (d) *Assemble and solve the discretised momentum* [...] *on the mesh with the current shape of the interface. The pressure field and face mass fluxes are used from the previous (outer) iteration.*
   (e) *Start of the PISO iteration loop:*
      (i) *Assemble and solve the discretised pressure Eq*[uation] *using velocity field obtained in the previous step.*
      (ii) *Calculate new absolute mass fluxes through the cell faces* [...] *with the new pressure field*
      (iii) *Calculate new cell-centre velocity* [...] *with new pressure field*
      (iv) *If specified number of PISO iterations (usually 2) is not reached return to step (i).*
   (f) *Calculate the net mass fluxes through the faces at the interface.*
   (g) *Convergence is checked and if the residual levels and the net mass flux through the interface do not satisfy the prescribed accuracy, the procedure returns to step (a).*
4. *If the final time instance is not reached, return to step 1.*

[Tuković and Jasak 2012]

For the purpose of better visualisation, Figure 2.6 is giving a schematically representaion of the solution procedure quoted above in form of a flow chart. The choice of the

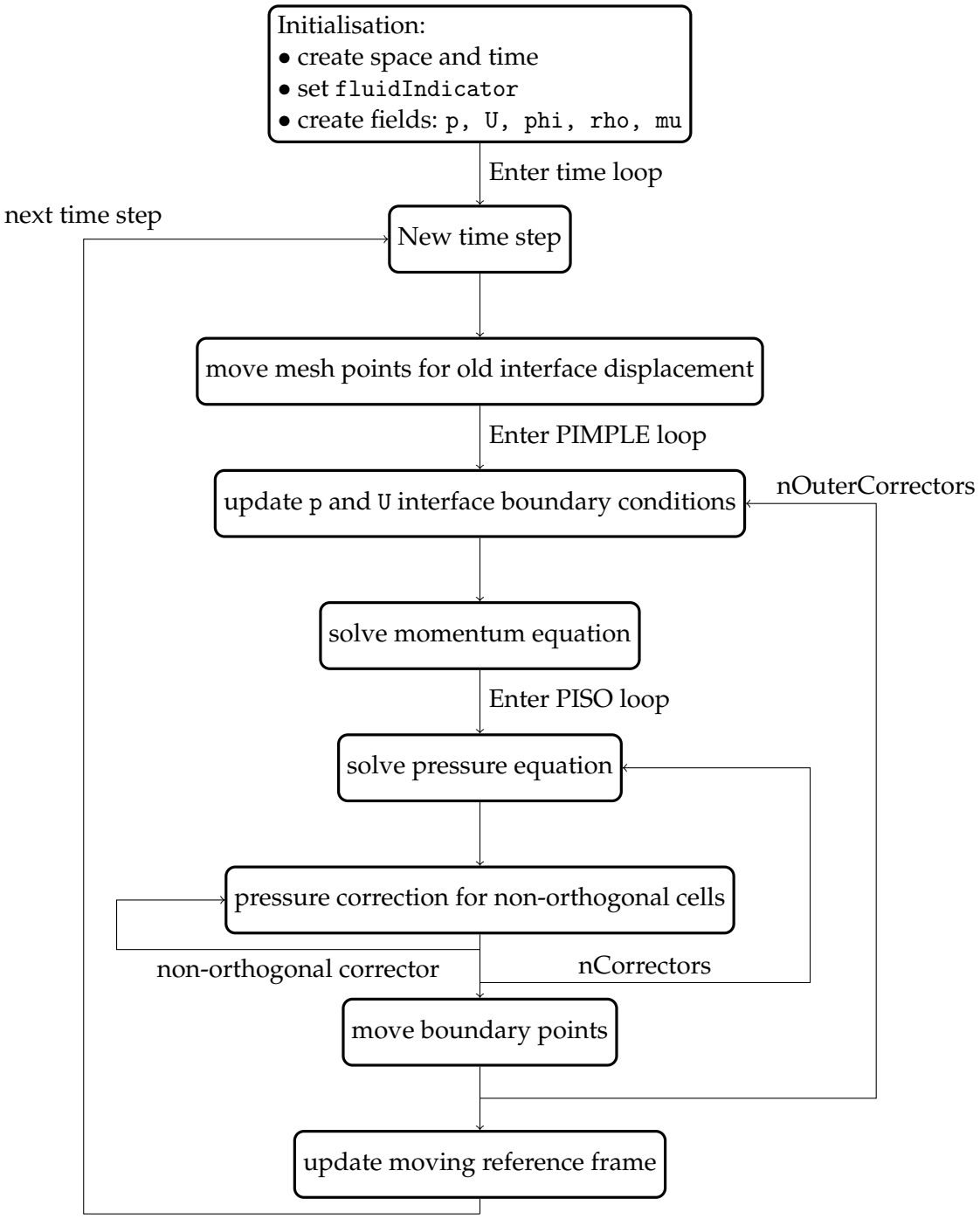


FIGURE 2.6: Flowchart: *bubbleInterTrackFoam* solution procedure

convergence conditions as well as the number of iterations has a large impact on the computation time. A small time step size for example, yields in a higher stability and less iteration steps for the time step loop to achieve convergence. However, longer computational time is the result as more steps are to be calculated. A performance increase can be achieved by limiting the movement of mesh points to only move interfacial points instead of all mesh points [Tuković and Jasak 2008]. Figure 2.5 shows the mesh deformation and movement of points schematically. The nOuterCorretors in Figure 2.6 should always have high numbers and range between 50 – 1000 iterations. Whereas the inner

nCorrector has a small number around 2 – 5 repetitions. The non-orthogonal Correctors only need to be used for highly non-orthogonal meshes [OpenFOAMWiki 2020b].

2.4.3 Boundary conditions

Since interface control volumes do not have adjacent control volumes, the application of boundary conditions is necessary. There are two basic types of boundary conditions: *Dirichelt* and *Neumann* boundary conditions. Most physical boundary behaviour is described by these two boundary conditions or through a combination of both. Dirichlet boundary conditions specify a fixed value of the variable at the boundary faces, while Neumann boundary conditions determine the normal gradient value [Charin et al. 2017].

TUKOVIC̀ and JASAK [2012] presented an assignment scheme for velocity and pressure boundary conditions at the interface through classic Dirichlet and Neumann conditions. When a variable on one side of the interface is assigned with a Dirichlet condition, the other side is assigned with a Neumann condition. For the velocity a fixed value Dirichlet condition is imposed for the **B** side (air bubble) of the interface ($\mathbf{v}_{Bf}$), while a the normal component of the velocity gradient ($\mathbf{n}_f \boldsymbol{\nabla} \mathbf{v}_{Af}$) is specified on the **A** side (fluid outside). For the pressure this procedure is switched. A fixed gradient Neumann condition is imposed at **B** side (air bubble) of the interface ($\mathbf{n}_f \boldsymbol{\nabla} p_{Bf}$) and a fixed value Dirichlet condition (p_{Af}) at the **A** side (fluid outside).

2.4.4 Program files and libraries

In the following section, some of the most important files and libraries of the *bubbleInterTrackFoam* solver will be shortly discussed. The solver directory is shown in the figure below (Figure 2.7). The main program, as it is outlined in Section 2.4.2, is located in the `bubbleInterTrackFoam` directory. The folder `freeSurface/` contains the key library for the surface-tracking procedure. The library is responsible for the calculation of the interface properties as well es the displacement of the interface points and will be discussed further below. The `utilities` folder contains a utility to set the so-called `fluidIndicator`. The `fluidIndicator` is a variable assigned to every cell, either being 1 or 0, to distinguish whether a cell is gas or liquid. The cells are explicitly assigned to one phase and cannot have mixed properties as in the VOF method (see Figure 2.1). Due to the continuous mesh, the `fluidIndicator` is used within the program to differentiate between the phases, for example to assign fluid properties correctly (schematic in Figure 2.8).

bubbleInterTrackFoam directory

In OpenFOAM, the `Make` directory contains files with information for compiling the software. If custom libraries are used, the path information has to be included here. The file `bubbleInterTrackFoam.C` contains the main program. Apart from the main loop, which involves solving the equations describing the continuum, the initialisation is performed. This includes reading mesh and fluid information, creating fields for initial pressure and

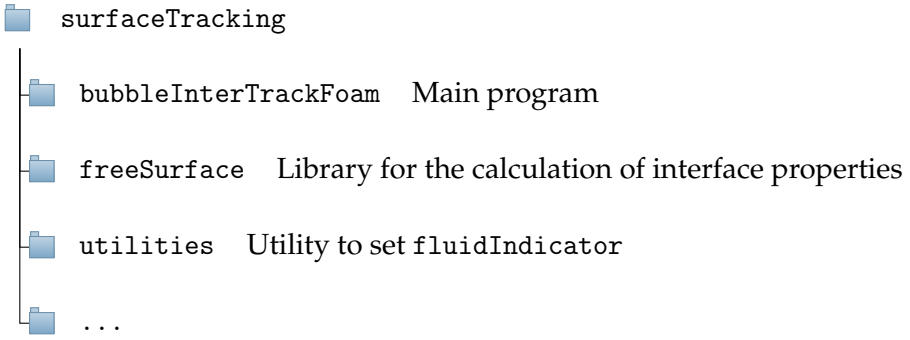


FIGURE 2.7: Directory of `surfaceTracking`

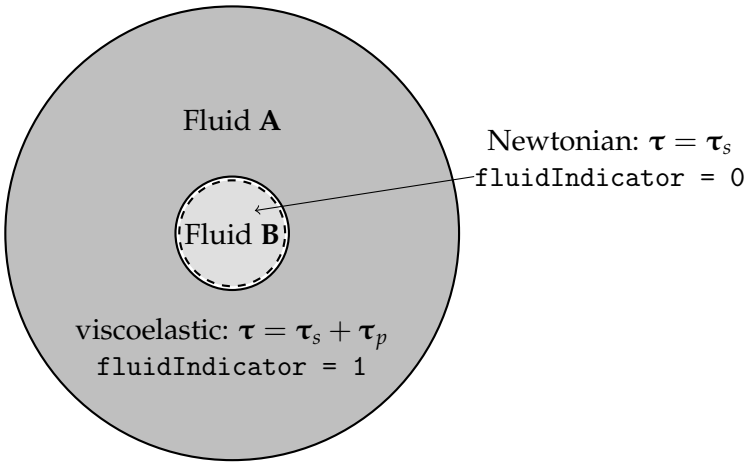


FIGURE 2.8: Fluid domains

velocity values as well as assigning the correct fluid properties the to domain cells. Instruction for the field created are located in a separate file, `createFields.H`, and are included in the main program.

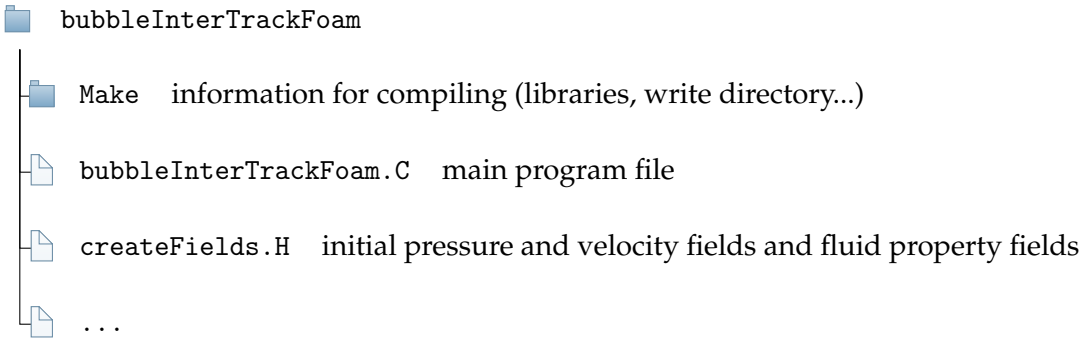


FIGURE 2.9: Directory of `bubbleInterTrackFoam`

freeSurface directory

The `freeSurface` directory contains library files to compute precise interface data. In the header and C-file a new C++ class in the OpenFOAM environment is created: *freeSurface*.

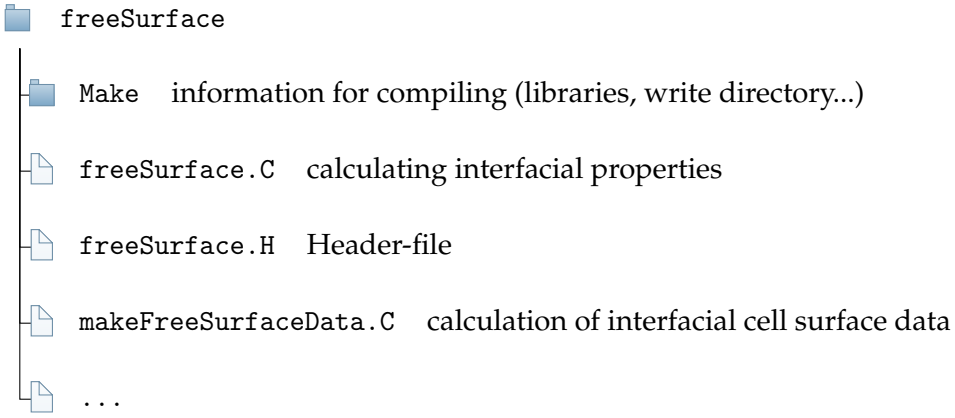


FIGURE 2.10: Directory of `freeSurface`

Among others, this class contains methods for the conversion of FV to FA data structures. Data structures associated with the area mesh are needed for the interface calculations. Interpolating from cell centred values of the boundary cells to values at the cell faces is an important step of the representation of the interface. The methods required for the surface-tracking procedure are part of this class.

Chapter 3

Mathematical Model

3.1 Introduction

In the following chapter, the fundamentals of a viscoelastic simulation discussed previously are complemented with a mathematical basis. The conservation laws for mass and momentum are presented to describe the continuum. Furthermore, the space conservation law is discussed, which is the additional equation required to prevent error in the moving mesh procedure. Afterwards, the constitutive equation for a correct representation of the viscoelastic behaviour is discussed. The last two sections discuss the derivation of a force balance on the bubble surface from the which jump conditions at the interface are derived. These are necessary to find the corresponding boundary conditions for pressure and velocity at the interface.

3.2 Governing continuum equations

Fluid flow equations are solved under the assumption of a continuous fluid. Mathematically, flow problems are described by the conservation laws of mass, momentum and energy. These partial differential equations are called the *Navier-Stokes-Equations*. Since the flow is considered to be isothermal and incompressible, only the **Conversation of Mass** and the **Conservation of Momentum** have to be taken into account [Schwarze 2013]. The fluid flow inside an arbitrary volume V is bounded by a closed moving surface S [Tuković and Jasak 2012]. In this case of a free rising bubble, the two phases are divided by an interface. Hence, conservation laws can be applied individually. The interface is a surface of zero thickness. At the boundary of the fluid, the physical properties change discontinuously and appropriate boundary conditions must be applied [Tuković 2005; Charin et al. 2017].

The formulation of the equations below is known as the ALE formulation which allows the mesh to present its own velocity $\mathbf{v}_S$. The mesh velocity may differ from the fluid velocity. In order to preserve the moving mesh from violating the balance of the conserved transport properties and causing artificial mass sources, an additional conservation law is formulated. This **space conservation law** has to be fulfilled simultaneously with the other conservation laws [Tuković 2005; Demirdzic and Peric 1988; Charin et al. 2017].

3.2.1 Conservation of Mass

The behaviour of a continuous fluid is governed by the conservation of mass. The main principle of this law comes from the statement that there is neither loss nor creation of material [Nikrityuk 2011]. Without external or internal sources this is described by the continuity equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \rho \,\mathrm{d}V + \oint_S \mathbf{n}\cdot\rho(\mathbf{v}-\mathbf{v}_S)\,\mathrm{d}S = 0, \tag{3.1}$$

where t is the time, $\mathbf{n}$ is the outward pointing unit normal vector on the surface S, ρ is the fluid density, $\mathbf{v}$ is the fluid velocity and $\mathbf{v}_S$ is the velocity of the surface. In case of constant density or negligible compressibility, respectively, the equation simplifies to [Tuković 2005; Tuković and Jasak 2008]:

$$\oint_S \mathbf{n}\cdot(\mathbf{v}-\mathbf{v}_S)\,\mathrm{d}S = 0. \tag{3.2}$$

3.2.2 Conservation of Momentum

The equation for the Conservation of Momentum in a fluid is derived by applying Newton's second law of motion to fluid motion [Nikrityuk 2011]. The following equation in ALE form is achieved by applying this law to a fluid passing through an infinitesimal, fixed control volume:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \rho\mathbf{v}\,\mathrm{d}V \oint_S \mathbf{n}\cdot\rho(\mathbf{v}-\mathbf{v}_S)\mathbf{v}\,\mathrm{d}S = \int_V \rho\mathbf{g}\,\mathrm{d}V + \oint_S \mathbf{n}\cdot\mathbf{\Pi}\,\mathrm{d}S, \tag{3.3}$$

where $\mathbf{g}$ is the gravitational acceleration and $\mathbf{\Pi}$ represents the total stress tensor. The left hand side of the equation represents the rate of change of momentum in the control volume and the rate of momentum change caused by convection through the control surface. The terms on the right hand side of the equation represent surface forces acting on the control volume. This includes the pressure term and a term for normal and shear stress represented by the *total viscous stress tensor*. Hence, the total stress tensor $\mathbf{\Pi}$ in a viscous fluid is decomposed into hydrostatic and viscous parts which has the following form [Tuković 2005]:

$$\mathbf{\Pi} = -P\mathbf{I} + \boldsymbol{\tau}. \tag{3.4}$$

Here, $\mathbf{I}$ is the unit tensor, P is the total pressure and $\boldsymbol{\tau}$ is the total viscous stress tensor. For Newtonian fluids, the total viscous stress tensor can be expressed as a direct function of the velocity. For the viscoelastic fluids, as treated in this work, an additional differential equation must be solved [Morrison 1998; Schwarze 2013]. (Section 3.3: Constitutive equations).

3.2.3 Space conservation law

To ensure that the mesh motion does not interfere with the balance of conserved transport properties, the space conservation law is introduced. This law expresses the relationship between the rate of volume change of a spatial domain and the velocity of the points of its boundary [Demirdzic and Peric 1988; Charin et al. 2017]:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \mathrm{d}V - \oint_S \mathbf{n}\cdot\mathbf{v}_S \,\mathrm{d}S = 0. \tag{3.5}$$

The temporal volume variation of the corresponding control volume is equal to the motion of the bounding surface S with arbitrary velocity $\mathbf{v}_S$.

3.3 Constitutive equations

Along with the conservation laws, the mechanical constitute equation describes the relationship between stress and the deformation rate of the fluid. For viscoelastic fluid behaviour the total viscous stress tensor $\boldsymbol{\tau}$ can be additively assembled from a solvent related *viscous* stress tensor $\boldsymbol{\tau}_s$ and an *elastic*, polymeric contribution to the stress tensor $\boldsymbol{\tau}_p$ [Favero et al. 2010]:

$$\boldsymbol{\tau} = \boldsymbol{\tau}_s + \boldsymbol{\tau}_p. \tag{3.6}$$

The viscous stress tensor is given by the definition of viscous stress in a Newtonian fluid

$$\boldsymbol{\tau}_s = 2\mu_s \mathbf{D}, \tag{3.7}$$

where μ_s is the dynamic solvent viscosity and $\mathbf{D}$ the deformation gradient tensor given by

$$\mathbf{D} = \frac{1}{2}\Big[\boldsymbol{\nabla}\mathbf{v} + (\boldsymbol{\nabla}\mathbf{v})^{\mathrm{T}}\Big]. \tag{3.8}$$

The symmetric tensor $\boldsymbol{\tau}_p$ corresponds to the elastic, polymeric part of the fluid. The elastic stress tensor is often obtained as a sum of partial elastic stress tensors $\boldsymbol{\tau}_{pk}$ calculated from different properties with constitutive differential models. This leads to higher accuracy of the elastic stress tensor [Favero et al. 2010; Morrison 1998]:

$$\boldsymbol{\tau}_p = \sum_{k=1}^{N} \boldsymbol{\tau}_{pk}. \tag{3.9}$$

A variety of different viscoelastic models exists for the calculation of the elastic stress tensor, which is available for the most versatile fields of application. Presented below is the Phan-Thien–Tanner differential constitutive model for the computation of polymeric solutions, which is used in this work.

3.3.1 Phan-Thien–Tanner model

The constitutive equation presented by PHAN-THIEN and TANNER [1977] is in common use for the simulation of polymer solutions and melts [Oliveira and Pinho 1999]. In its general form it can be written as:

$$f(\mathrm{tr}(\boldsymbol{\tau}_p))\boldsymbol{\tau}_p + \lambda \overset{\triangledown}{\boldsymbol{\tau}}_p + \zeta(\mathbf{D}\cdot\boldsymbol{\tau}_p - \boldsymbol{\tau}_p\cdot\mathbf{D}^{\mathrm{T}}) = 2\mu_p\mathbf{D}, \tag{3.10}$$

where λ is the relaxation time, μ_p is a constant dynamic polymer viscosity coefficient and ζ is a constant model parameter that controls the amount of movement between the fluid polymeric chains. Oldroyd's upper-convected derivative $\overset{\triangledown}{\boldsymbol{\tau}}_p$ is defined as:

$$\overset{\triangledown}{\boldsymbol{\tau}}_p = \frac{\mathrm{D}\mathbf{v}}{\mathrm{D}t} - \boldsymbol{\tau}_p\cdot\boldsymbol{\nabla}\mathbf{v} - \boldsymbol{\nabla}\mathbf{v}^{\mathrm{T}}\cdot\boldsymbol{\tau}_p. \tag{3.11}$$

In the original paper the linearised from of the function f was given as:

$$f(\mathrm{tr}(\boldsymbol{\tau}_p)) = 1 + \frac{\epsilon\lambda}{\mu_p}\mathrm{tr}(\boldsymbol{\tau}_p) \tag{3.12}$$

and later the exponential form was released [N. Phan-Thien 1978]:

$$f(\mathrm{tr}(\boldsymbol{\tau}_p)) = \exp\Big(\frac{\epsilon\lambda}{\mu_p}\mathrm{tr}(\boldsymbol{\tau}_p)\Big). \tag{3.13}$$

with ϵ representing a parameter related to the elongational behaviour of the model.

3.4 Free surface interface conditions

The fundamental idea of the surface-tracking method is the exact representation of the conditions at the free surface. By introducing an interface, the continuum equations can be applied separately for each phase of the continuous domain. Since the fluid properties change discontinuously through the interface, great emphasis must be given to the calculation of the associated boundary conditions. For an exact description of the conditions at the phase boundary it is inevitable to find relations between the velocities and pressures on either side of the free surface. TUKOVIĆ [2005] presented a method which calculates the boundary conditions at the interface derived from a force balance at the free surface. The following sections present this approach and describe the formulation of the force balance.

3.4.1 Force balance

Figure 3.1 shows a schematic graphic of the boundary between the two incompressible fluids. As pointed out before, the goal is to acquire a **velocity jump condition** and a **pressure jump condition** to determine the conditions at the two sides of the free surface.

From the momentum conservation law TUKOVIĆ [2005] derives the **dynamic condition** which states that forces acting on the fluid at the free surface are in equilibrium. The

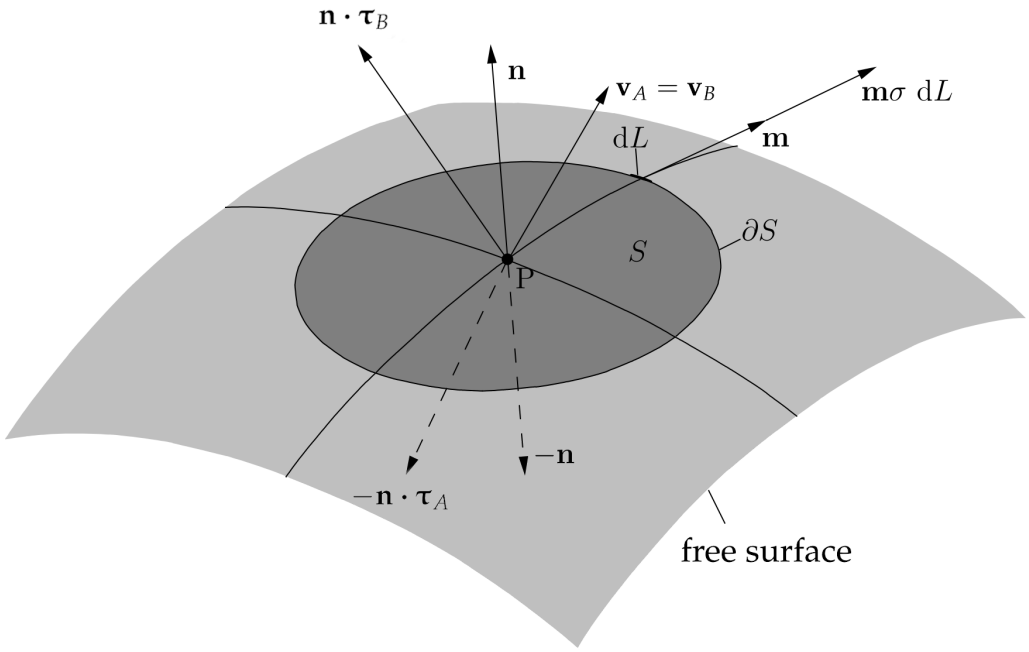


FIGURE 3.1: Forces acting on the free surface [Tuković 2005]

balance is derived from a part of the free surface S bounded by the closed curve ∂S. See Figure 3.1 for illustration. The curve integral equation of the equilibrium is expressed as:

$$\int_S \mathbf{n} \cdot \boldsymbol{\Pi}_B \, \mathrm{d}S - \int_S \mathbf{n} \cdot \boldsymbol{\Pi}_A \, \mathrm{d}S + \oint_{\partial S} \mathbf{m}\sigma \, \mathrm{d}L = 0, \tag{3.14}$$

where $\boldsymbol{\Pi}_A$ and $\boldsymbol{\Pi}_B$ are total stress tensors on both sides of the free surface, σ is the surface tension coefficient, $\mathbf{m}$ is the outer unit binormal on ∂S, which is tangential to S and L represents the arc length along the curve ∂S. By applying the divergence theorem for curved surfaces to the equation above, a surface integral equation is obtained:

$$\int_S \mathbf{n} \cdot \boldsymbol{\Pi}_B \, \mathrm{d}S - \int_S \mathbf{n} \cdot \boldsymbol{\Pi}_A \, \mathrm{d}S + \int_S \boldsymbol{\nabla}_S \sigma \, \mathrm{d}S + \int_S \kappa \mathbf{n} \sigma \, \mathrm{d}S = 0, \tag{3.15}$$

where $\kappa = -\boldsymbol{\nabla}_S \mathbf{n}$ describing twice the mean curvature of the free surface. With $\boldsymbol{\nabla}_S$ being the surface gradient operator only accounting the normal part of the gradient. A formulation proposed by CHEN, SARIC and STONE [2000] where the surface gradient operator is defined as:

$$\boldsymbol{\nabla} = (\mathbf{I} - \mathbf{nn}) \cdot \boldsymbol{\nabla} + \mathbf{nn} \cdot \boldsymbol{\nabla} = \boldsymbol{\nabla}_S + \mathbf{nn} \cdot \boldsymbol{\nabla}. \tag{3.16}$$

When Equation 3.15 is weighted for one point of the surface S, the following differential equation is obtained:

$$\mathbf{n} \cdot \boldsymbol{\Pi}_B - \mathbf{n} \cdot \boldsymbol{\Pi}_A = -\sigma \kappa \mathbf{n} - \boldsymbol{\nabla}_S \sigma. \tag{3.17}$$

Considering that the total stress tensor $\mathbf{\Pi}$ can be split into the sum of an isotropic pressure part $-P\mathbf{I}$ and total viscous stress tensor $\boldsymbol{\tau}$ (see Equation 3.4), Equation 3.17 becomes:

$$\mathbf{n} \cdot (-P_B \mathbf{I}) + \mathbf{n} \cdot \boldsymbol{\tau}_B + \mathbf{n} \cdot (P_A \mathbf{I}) - \mathbf{n} \cdot \boldsymbol{\tau}_A = -\sigma\kappa\mathbf{n} - \boldsymbol{\nabla}_S \sigma \tag{3.18}$$

Rearrangement leads to the *total force balance* for both sides of the free surface:

$$(P_B - P_A)\mathbf{n} - \mathbf{n} \cdot (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A) = \sigma\kappa\mathbf{n} + \boldsymbol{\nabla}_S \sigma. \tag{3.19}$$

During the course of the next sections, the total force balance is divided into a normal and a tangential part. This allows the development of separate equations for pressure and velocity jump later on.

3.4.2 Normal force balance

Deriving the normal part of the force balance starts with a scalar multiplication of the Equation 3.19 with the unit normal $\mathbf{n}$:

$$\mathbf{n} \cdot (P_B - P_A)\mathbf{n} - \mathbf{n} \cdot (\mathbf{n} \cdot (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A)) = \mathbf{n} \cdot (\sigma\kappa\mathbf{n}) + \mathbf{n} \cdot (\boldsymbol{\nabla}_S \sigma). \tag{3.20}$$

Rearrangement leads to:

$$(P_B - P_A) \underbrace{(\mathbf{n} \cdot \mathbf{n})}_{=1} - \mathbf{n} \cdot (\mathbf{n} \cdot (\boldsymbol{\tau_B} - \boldsymbol{\tau_A}) = \sigma\kappa \underbrace{(\mathbf{n} \cdot \mathbf{n})}_{=1} + \underbrace{\mathbf{n} \cdot (\boldsymbol{\nabla}_S \sigma)}_{=0}. \tag{3.21}$$

Since the scalar product of two unitary vectors pointing in the same direction equals 1 and normal component of the surface gradient does not exist, the following equation for the **normal force balance** is obtained:

$$P_B - P_A - \underbrace{\mathbf{nn} : (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A)}_{\mathbf{n}\cdot(\mathbf{n}\cdot(\boldsymbol{\tau}_B - \boldsymbol{\tau}_A))} = \sigma\kappa \tag{3.22}$$

For visual convenience the double dot product $\mathbf{nn}:$ with the unit normal is introduced, whereas $\mathbf{nn}\cdot$ refers in the following to the scalar-vector product.

3.4.3 Tangential Force Balance

The tangential force balance on the free surface is obtained from the difference of the normal part of the normal force balance (Equation 3.22) and the total force balance (Equation 3.19). Scaling the normal force balance with $\mathbf{n}$ gives the normal part of the normal force balance:

$$\mathbf{n} \cdot \Big[P_B - P_A - \mathbf{nn} : (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A) - \sigma\kappa \Big] = 0. \tag{3.23}$$

which rewrites to:

$$(P_B - P_A)\mathbf{n} - \mathbf{n} \cdot [\mathbf{nn} : (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A)] - \sigma\kappa\mathbf{n} = 0. \tag{3.24}$$

Subtraction of the normal components of the normal force balance (Equation 3.24) from the total force balance (Equation 3.19) gives:

$$\begin{aligned}&(P_B - P_A)\mathbf{n} - \mathbf{n}\cdot(\boldsymbol{\tau}_B - \boldsymbol{\tau}_A) - \sigma\kappa\mathbf{n} - \boldsymbol{\nabla}_S\sigma \\ &- \Big((P_B - P_A)\mathbf{n} - \mathbf{n}\cdot[\mathbf{nn} : (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A)] - \sigma\kappa\mathbf{n}\Big) = 0.\end{aligned} \tag{3.25}$$

Simplification yields the **tangential force balance**:

$$\mathbf{n}\cdot(\boldsymbol{\tau}_B - \boldsymbol{\tau}_A) - \mathbf{n}[\mathbf{nn} : (\boldsymbol{\tau}_B - \boldsymbol{\tau}_A)] = -\boldsymbol{\nabla}_S\sigma. \tag{3.26}$$

3.5 Interfacial boundary conditions

For the definition of adequate boundary conditions at the interface is essential to acquire dependencies for velocity and pressure across the interface. BATCHELOR [2000] defines the relation between the velocities on both sides of the interface by the **kinematic condition**. This first boundary condition states the velocity must be continuous across the interface. The condition is derived from a mass conservation principle, where the interface is treated as a two-dimensional surface that does not retain mass. In other words, the velocities on both sides of the free surface are equal:

$$\mathbf{v}_A = \mathbf{v}_B, \tag{3.27}$$

where $\mathbf{v}_A$ and $\mathbf{v}_B$ are the fluid velocities on both sides of the free surface.

From the previously obtained force balances, additional boundary conditions for pressure and velocity at the interface are derived. The proper coupling between the two phases at the sharp interface is performed by so-called jump conditions.

The **normal** and **tangential force balances** lead to the equations for the pressure variation across the interface, and a relationship between the normal derivative of tangential velocity on the two sides of the interface [Tuković and Jasak 2012]. As part of the adaptation process for the simulation of viscoelastic fluids, these jump conditions differ from the equations initially proposed by TUKOVIĆ [2005].

3.5.1 Pressure jump condition

Starting of from the **normal force balance**, the equation elements will be converted into parameters available during computation. In order to convert the normal force balance into an equation representing the absolute pressure variation at the free surface some identities have to be pointed out before. As described in Section 3.3, the total viscous stress tensor is broken down into a solvent contribution and a polymeric contribution

(Equation 3.6), $\boldsymbol{\tau} = \boldsymbol{\tau}_s + \boldsymbol{\tau}_p$. Through inserting the definition of the stress tensor into the normal force balance (Equation 3.22) the following is obtained:

$$P_B - P_A - \mathbf{nn} : (\boldsymbol{\tau}_{Bs} - \boldsymbol{\tau}_{As}) - \mathbf{nn} : (\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap}) = \sigma\kappa. \tag{3.28}$$

Here, $\boldsymbol{\tau}_{As}$ and $\boldsymbol{\tau}_{Bs}$ are the viscous stress tensors on either sides of the free surface, whereas $\boldsymbol{\tau}_{Ap}$ and $\boldsymbol{\tau}_{Bp}$ represent the elastic stress tensors. While the polymeric contribution is obtained by solving an extra differential equation, the viscous stress tensor is calculated directly from the fluid velocity $\mathbf{v}$. In order to find an expression for $\mathbf{nn} : \boldsymbol{\tau}_s$ in the equation above, some transformations with the deformation gradient tensor $\mathbf{D}$ (see Equation 3.8) are performed. But first, some identities according to CHEN, SARIC and STONE [2000] must be demonstrated.

The velocity gradient $\boldsymbol{\nabla}\mathbf{v}$ expressed with the surface gradient $\boldsymbol{\nabla}_S$ (see Equation 3.16) is:

$$\boldsymbol{\nabla}\mathbf{v} = \boldsymbol{\nabla}_S\mathbf{v} + \mathbf{nn} \cdot \boldsymbol{\nabla}\mathbf{v}, \tag{3.29}$$

where the trace results in

$$\boldsymbol{\nabla} \cdot \mathbf{v} = \boldsymbol{\nabla}_S \cdot \mathbf{v} + \mathbf{nn} : \boldsymbol{\nabla}\mathbf{v}. \tag{3.30}$$

With $\mathbf{nn} : \mathbf{D} = \mathbf{nn} : \boldsymbol{\nabla}\mathbf{v}$ one can write

$$\mathbf{nn} : \mathbf{D} = \boldsymbol{\nabla} \cdot \mathbf{u} - \boldsymbol{\nabla}_S : \mathbf{v}. \tag{3.31}$$

For an incompressible flow, with $\boldsymbol{\nabla}\mathbf{v} = 0$, this breaks down to

$$\mathbf{nn} : \mathbf{D} = -\boldsymbol{\nabla}_S \cdot \mathbf{v}. \tag{3.32}$$

Using the definition of the viscous stress tensor (Equation 3.7) leads to

$$\mathbf{nn} : \boldsymbol{\tau}_s = -2\mu_s \boldsymbol{\nabla}_S \cdot \mathbf{v}. \tag{3.33}$$

From Equation 3.28 and 3.33, a dependency of the total pressure on either sides of the interface is obtained:

$$P_B - P_A = \sigma\kappa - 2(\mu_{Bs} - \mu_{As})\boldsymbol{\nabla}_S \cdot \mathbf{v} + \mathbf{nn} : (\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap}). \tag{3.34}$$

The dynamic pressure p is introduces through the definition of the total pressure as:

$$P = p + \rho\mathbf{g} \cdot \mathbf{r}, \tag{3.35}$$

where $\mathbf{r}$ is the position vector. Hence, in combination with Equation 3.34 follows:

$$p_A = p_B - (\rho_A - \rho_B)\mathbf{g} \cdot \mathbf{r} - (\sigma\kappa) + 2(\mu_{Bs} - \mu_{As})\boldsymbol{\nabla}_S \cdot \mathbf{v} - \mathbf{nn} : (\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap}). \tag{3.36}$$

With the assumption of only viscoelastic behaviour in the surrounding fluid (see Figure 2.8), the final **pressure jump condition** is acquired:

$$p_A = p_B - (\rho_A - \rho_B)\mathbf{g}\cdot\mathbf{r} - (\sigma\kappa) + 2(\mu_{Bs} - \mu_{As})\boldsymbol{\nabla}_S\cdot\mathbf{v} + \mathbf{nn}:(\boldsymbol{\tau}_{Ap}). \tag{3.37}$$

Resulting in an expression for the pressure on both sides of the interface, depending on the fluid velocities and the position vector.

3.5.2 Velocity jump condition

The velocity jump condition shall update the velocity at the interface and the gradient of the interface velocity at the interface. This time starting off from the **tangential force balance** (Equation 3.26) in combination with the definition of the total viscous stress tensor (Equation 3.6), the following expression is obtained:

$$\begin{aligned}&\mathbf{n}\cdot(\boldsymbol{\tau}_{Bs} - \boldsymbol{\tau}_{As}) + \mathbf{n}\cdot(\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap})\\ &-\mathbf{n}\Big[\mathbf{nn}:(\boldsymbol{\tau}_{Bs} - \boldsymbol{\tau}_{As})\Big] - \mathbf{n}\Big[\mathbf{nn}:(\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap})\Big] = -\boldsymbol{\nabla}_S\sigma.\end{aligned} \tag{3.38}$$

In order to create a velocity depending equation, the viscous stress tensor $\boldsymbol{\tau}_s$ is replaced once again. An expression for $\mathbf{nn}:\boldsymbol{\tau}_s$ (Equation 3.33) has been derived in the section above and will be used further down the line. To find an expression for $\mathbf{n}\cdot\boldsymbol{\tau}_s$, TUKOVIĆ [2005] established the following identities from the definition of the surface gradient operator (Equation 3.16):

$$\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v} + \mathbf{n}(\boldsymbol{\nabla}_S\cdot\mathbf{v}) = (\mathbf{I} - \mathbf{nn})\cdot(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) \tag{3.39a}$$

$$\mathbf{n}\cdot\boldsymbol{\nabla}_S\mathbf{v} = 0 \tag{3.39b}$$

$$\mathbf{n}\cdot(\mathbf{nn})\cdot\boldsymbol{\nabla}\mathbf{v} = \mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v} \tag{3.39c}$$

$$\mathbf{nn}\cdot(\boldsymbol{\nabla}\mathbf{v})\cdot\mathbf{n} = \mathbf{nn}\cdot(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) \tag{3.39d}$$

Using Equations 3.39a – 3.39d and the definition of the viscous stress tensor in Equation 3.7, an expression for $\mathbf{n}\cdot\boldsymbol{\tau}_s$ as a function of the velocity is obtained:

$$\mathbf{n}\cdot\boldsymbol{\tau}_S = \mathbf{n}\cdot\Big[\mu_S\Big[\boldsymbol{\nabla}\mathbf{v} + (\boldsymbol{\nabla}\mathbf{v})^T\Big]\Big] \tag{3.40a}$$

$$= \mu_S\Big[\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v} + (\boldsymbol{\nabla}\mathbf{v})\cdot\mathbf{n}\Big] \tag{3.40b}$$

$$= \mu_S\Big[\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v} + (\boldsymbol{\nabla}_S\mathbf{v})\cdot\mathbf{n} + \mathbf{nn}\cdot\boldsymbol{\nabla}\mathbf{v}\cdot\mathbf{n}\Big] \tag{3.40c}$$

$$= \mu_S\Big[\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v} + (\boldsymbol{\nabla}_S\mathbf{v})\cdot\mathbf{n} - \mathbf{n}(\boldsymbol{\nabla}_S\cdot\mathbf{v})\Big] \tag{3.40d}$$

Replacing $\mathbf{n}\cdot\boldsymbol{\tau}_s$ and $\mathbf{nn}:\boldsymbol{\tau}_s$ in the tangential stress balance (Equation 3.38) with the expressions from Equations 3.40d and 3.33 results in an expression for the relationship of

the velocity gradients at the two sides of the interfac:

$$\begin{aligned}
&\mu_{Bs}\Big[\mathbf{n}\cdot(\boldsymbol{\nabla}\mathbf{v})_B + (\boldsymbol{\nabla}_S\mathbf{v})\cdot\mathbf{n} - \mathbf{n}(\boldsymbol{\nabla}_S\cdot\mathbf{v})\Big] \\
&-\mu_{As}\Big[\mathbf{n}\cdot(\boldsymbol{\nabla}\mathbf{v})_A + (\boldsymbol{\nabla}_S\mathbf{v})\cdot\mathbf{n} - \mathbf{n}(\boldsymbol{\nabla}_S\cdot\mathbf{v})\Big] \\
&+\mathbf{n}\cdot(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap}) - \mathbf{n}\Big[\mathbf{nn}:(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap})\Big] \\
&= -\boldsymbol{\nabla}_S\sigma + \mathbf{n}\Big[[-2\mu_{Bs}\boldsymbol{\nabla}_S\cdot\mathbf{v}] - [-2\mu_{As}\boldsymbol{\nabla}_S\cdot\mathbf{v}]\Big],
\end{aligned} \tag{3.41}$$

where simplification is giving

$$\begin{aligned}
&\mu_{Bs}(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) + \mu_{Bs}\boldsymbol{\nabla}_S\mathbf{v}\cdot\mathbf{n} - \cancel{\mathbf{n}\mu_{Bs}(\boldsymbol{\nabla}_S\cdot\mathbf{v})} \\
&-\mu_{As}(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) - \mu_{As}\boldsymbol{\nabla}_S\mathbf{v}\cdot\mathbf{n} + \cancel{\mathbf{n}\mu_{As}(\boldsymbol{\nabla}_S\cdot\mathbf{v})} \\
&+\mathbf{n}\cdot(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap}) - \mathbf{n}\Big[\mathbf{nn}:(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap})\Big] \\
&= -\boldsymbol{\nabla}_S\sigma - 2\mathbf{n}(\mu_{Bs}-\mu_{As})\boldsymbol{\nabla}_S\cdot\mathbf{v}
\end{aligned} \tag{3.42}$$

and further rearrangement leads to the **velocity jump condition**:

$$\begin{aligned}
&\mu_{Bs}\Big[(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) + \mathbf{n}\boldsymbol{\nabla}_S\cdot\mathbf{v}\Big]_B - \mu_{As}\Big[(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) + \mathbf{n}\boldsymbol{\nabla}_S\cdot\mathbf{v}\Big]_A \\
&= -\boldsymbol{\nabla}_S\sigma - (\mu_{Bs}-\mu_{As})\boldsymbol{\nabla}_S\mathbf{v}\cdot\mathbf{n} - \mathbf{n}\cdot(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap}) + \mathbf{n}\Big[\mathbf{nn}:(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap})\Big].
\end{aligned} \tag{3.43}$$

Giving an expression for a relationship between normal derivative of velocity at the two sides of the interface.

Chapter 4

Numerical Method and Computational Implementation

4.1 Introduction

This chapter describes the implementation of the mathematical model established in Chapter 3. Section 2.4 discussed the principle of the *bubbleInterTrackFoam* solver, which is the basis for the implementation of the viscoelastic model in this work. As it was outlined in Section 2.4.4, the class *freeSurface* is essential for the surface-tracking method. It contains methods with finite area calculations to update the boundary conditions at the interface and providing the moving mesh functionality. The *freeSurface* class is the basis of all surface-tracking related processes.

Updating the boundary conditions at the free surface is one of the key steps for a correct description of the behaviour of a bubble and requires special attention during viscoelastic implementation. The mathematical groundwork was established by the derivation of the pressure jump and velocity jump conditions.

While the implementation of the viscoelastic stress tensor into the equations for the description of the boundary conditions requires a rather large effort, the implementation into the momentum equation is relatively straightforward. For the calculation of the boundary conditions as well as for the solution of the momentum equation the viscoelastic stress term has to be calculated. This is done by solving a separate differential equation (Section 3.3: Constitutive equations). The computation is performed by a foam-extend library called *viscoelasticModel*. The library has a number of features required for the calculation of viscoelastic stress, which could be used without further ado for the implementation into the momentum equation. For the update of the boundary conditions at the interface however, finite area stress tensor fields are required, which is not part of the original library.

In order use the FAM for the calculation of stress tensors at the free surface, some adaption in the *finiteArea* library must be made. This is necessary to enable computations with the data structures of the viscoelastic surface stress tensor.

The following sections build the bridge between the functionality of the solver and the mathematical model. Along with pieces of code, the viscoelastic implementation is explained in further course of this chapter. Figure 4.1 shows the solution procedure of the newly developed *bubbleInterTrackModel* solver. The grey highlighted boxes show the

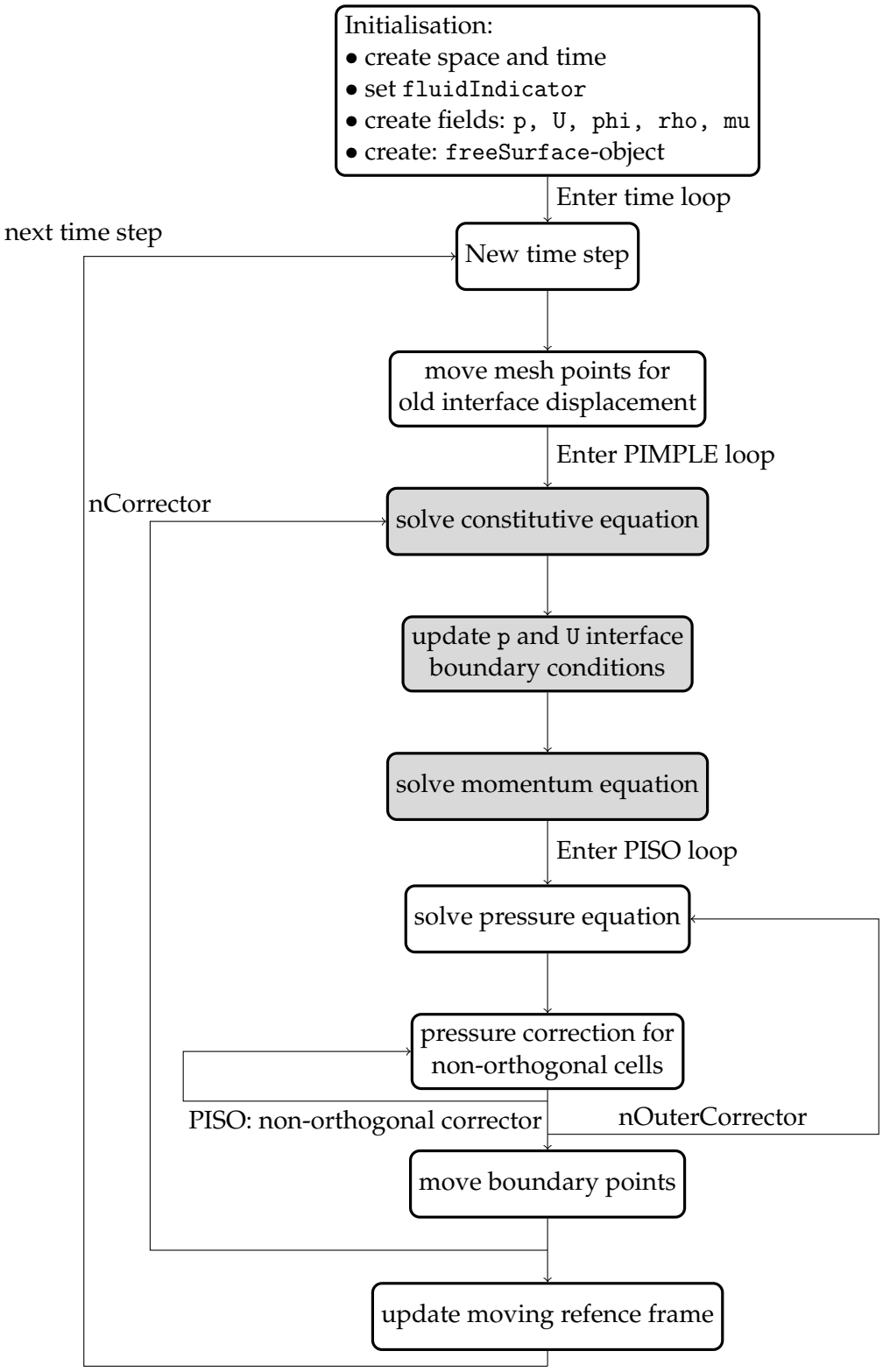


FIGURE 4.1: *bubbleInterTrackModel* solution procedure with viscoelastic adaption in the highlighted steps

calculation of the viscoelastic stress tensor through the constitutive equation with the *viscoelasticModel* library, the implementation of the viscoelastic stress tensor methods to update the interface boundary conditions and the implementation into the momentum

equation. The original *viscoelasticModel* library is intended to calculate the stress tensor for equations solved by the FVM. As the momentum equation is solved for the whole fluid domain with the FVM, the standard return variables of the library are used. The implementation is described in the following section.

4.2 Implementation into the momentum equation

Both bubble and liquid are described by the Law of Conservation of Momentum (Equation 3.3). Shown below is the law conservation of momentum equation in Eulerian form:

$$\rho\frac{\partial \mathbf{v}}{\partial t} + \rho(\mathbf{v}\cdot\nabla\mathbf{v}) = -\nabla p + \nabla\cdot\boldsymbol{\tau} + \rho\mathbf{g}. \tag{4.1}$$

The calculation of the divergence of the total stress tensor, $\boldsymbol{\tau} = \boldsymbol{\tau}_s + \boldsymbol{\tau}_p$, is carried out through a method of the *viscoelasticModel* library (discussed in Section 4.4). This method returns the divergence of the total stress tensor, including both viscous and extra stress. As shown in Figure 2.8, only the surrounding fluid has viscoelastic properties, while the bubble is calculated entirely Newtonian. In order to distinguish the two phases in the continuous mesh, the `fluidIndicator` discussed in Section 2.4.4 is used. From Equation 3.7 follows that the divergence of the viscous stress tensor is:

$$\nabla\cdot\boldsymbol{\tau}_s = \mu_s\nabla^2\mathbf{v}. \tag{4.2}$$

In the main solver file `bubbleInterTrackModel.C`, a finite volume vector matrix is set up to solve the momentum equation. Code Sample 4.1 below shows changes in the code to adapt viscoelastic behaviour for the surrounding fluid **A** and represent the bubble **B** with Newtonian behaviour.

The `fluidIndicator` is used to distinguish between the phases: For the bubble **B**, the `fluidIndicator` is `0`, line `108` disappears and a calculation according to Equation 4.2 is performed. For the surrounding fluid **A**, `fluidIndicator` is `1`, the term `fvm::laplacian(interface.muFluidB(), U)` disappears from the matrix leaving only `interface.divTau(U)`, which represents $\nabla\cdot\boldsymbol{\tau}$. In the `createFields.C`-file, the `fluidIndicator` is used in the same manner in order to calculate a `scalarField rho` that already includes the differentiation between the fluid domains. With the aim of improving stability an under-relaxation step is added in line `111` before the matrix is solved.

```
100 fvVectorMatrix UEqn
101 (
102     fvm::ddt(rho, U)
103     + rho*aF
104     + fvm::div(fvc::interpolate(rho)*phi, U, "div(phi,U)")
105
106      // edited
107     - fvm::laplacian(interface.muFluidB(), U)
108     - (fluidIndicator * (interface.divTau(U) - fvm::laplacian(interface.
        muFluidB(), U)))
```

```
109 );
110 // added: under-relaxation for momentum equation
111 UEquationrelax();
112
113 solve(UEqn == -fvc::grad(p));
114
115 Info<< "UEqn␣solved" << endl << endl;
```

CODE SAMPLE 4.1: Setting up momentum equation in `bubbleInterTrackModel.C`

4.3 Adaption of the *finiteArea* library

To enable the viscoelastic computation of surface stress tensors with the *viscoelasticModel* library, some changes in the *finiteArea* library must be made. This library contains utilities and algorithms to perform the FA operations. Since the *finiteArea* library is only part of foam-extend, which is a development version, the library is work in progress. As a result of this, the definition of matrices with more complex data structures than *scalar*, *vector* and *tensor* is not yet implemented. Additional types of matrices are added to the files `faMatrices.C` and `faMatrices.H` (Code Sample 4.3 and Code Sample 4.2) analogous to the corresponding files in the more developed *finiteVolume* library. For the intended calculation of the viscous stress tensor at the free surface (type: `areaSymmTensorField`), only the `faSymmTensorMatrix` is required for now. Also, it was not possible to perform divergence matrix operations of the viscous stress tensor, which is required during the calculation of the viscoelastic stress tensor. HJ (asm. HRVOJE JASAK) implemented a test in the `famDiv.C`-file to check validity of the processing data. The step does not process any data it is just a security barrier to process correct data. In order do computations with the viscoelastic stress tensor in *fam* matrices, this security barrier is bypassed and returned directly. The folder structure and the implementation are shown below.

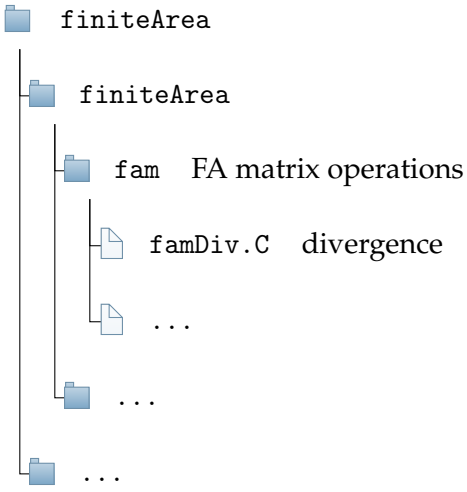


FIGURE 4.2: Directory of the finiteArea library

```
47 typedef faMatrix<scalar> faScalarMatrix;
48 typedef faMatrix<vector> faVectorMatrix;
49 typedef faMatrix<tensor> faTensorMatrix;
50 typedef faMatrix<symmTensor> faSymmTensorMatrix; // added
51 typedef faMatrix<sphericalTensor> faSphericalTensorMatrix; // added
52 typedef faMatrix<symmTensor4thOrder> faSymmTensor4thOrderMatrix; // added
53 typedef faMatrix<diagTensor> faDiagTensorMatrix; // added
```

CODE SAMPLE 4.2: Adapting the finiteArea library in `faMatrices.H`

```
38 defineTemplateTypeNameAndDebug(faScalarMatrix, 0);
39 defineTemplateTypeNameAndDebug(faVectorMatrix, 0);
40 defineTemplateTypeNameAndDebug(faTensorMatrix, 0);
41 defineTemplateTypeNameAndDebug(faSymmTensorMatrix, 0); // added
42 defineTemplateTypeNameAndDebug(faSphericalTensorMatrix, 0);  // added
43 defineTemplateTypeNameAndDebug(faSymmTensor4thOrderMatrix, 0);  // added
44 defineTemplateTypeNameAndDebug(faDiagTensorMatrix, 0); // added
```

CODE SAMPLE 4.3: Adapting the finiteArea library in `faMatrices.C`

```
43 template<class Type>
44 tmp<faMatrix<Type> >
45 div
46 (
47     const edgeScalarField& flux,
48     const GeometricField<Type, faPatchField, areaMesh>& vf,
49     const word& name
50 )
51 {/*
52     const areaVectorField& n = vf.mesh().faceAreaNormals();
53
54     tmp<faMatrix<Type> > tM
55     (
56         fa::convectionScheme<Type>::New
57         (
58             vf.mesh(),
59             flux,
60             vf.mesh().schemesDict().divScheme(name)
61         )().famDiv(flux, vf)
62     );
63     faMatrix<Type>& M = tM();
64
65     GeometricField<Type, faPatchField, areaMesh> v
66     (
67         fa::convectionScheme<Type>::New
68         (
69             vf.mesh(),
70             flux,
71             vf.mesh().schemesDict().divScheme(name)
```

```
72          )().facDiv(flux, vf)
73      );
74
75      //HJ Check if the product is from left or right.  HJ, 6/Dec/2016
76      M -= n*(n & v);
77
78      return tM;
79      }
80     */
81
82          return fa::convectionScheme<Type>::New
83          (
84              vf.mesh(),
85              flux,
86              vf.mesh().schemesDict().divScheme(name)
87          )().famDiv(flux, vf);
88      }
89  }
```

CODE SAMPLE 4.4: Adapting the finiteArea library in `famDiv.C`

4.4 Viscoelastic modelling

In order to provide the solver with the ability to calculate the stress tensor from viscoelastic constitutive equations as it is described in Section 3.3, the foam-extend library *viscoelasticTransportModels* is taken as a basis. This library offers several classes with models for the calculation of the extra, polymeric stress tensor $\boldsymbol{\tau}_p$ and the derivative of the total stress tensor $\boldsymbol{\nabla} \cdot \boldsymbol{\tau}$, which was used already in the momentum equation (Section 4.2). The library also contains a class for a run-time selection mechanism of the constitutive models and provides class with a multi-mode feature to calculate partial stress tensors and form the sum for a higher accuracy as it is shown in Equation 3.9.

The different viscosity models of the original library have three methods for the calculation of the viscoelastic stress tensor. In two methods, the viscoelastic stress tensor is calculated as a FV associated data structure through solving a finite volume matrix and returned. A third method calculates and returns the coupling term for the momentum. Again, methods of the *finiteVolume* library are used. The calculation of FA associated `area`-tensor field data structures instead of FV associated `vol`-tensor field data structures still has to be implemented. The groundwork for this is laid by adapting the *finiteArea* library in the previous section.

As mentioned above, the *viscoelasticTransportModels* library consists of more than one class for the actual calculation of the stress tensor using a viscosity model. There is the feature of run-time selection of the viscosity model and the calculation of the viscoelastic stress tensor as sum of partial elements. Each of these operations is defined in a separate class through whose the flow information is passed and processed. Figure 4.3 shows the different classes of the library in a flow chart and illustrates their connection. In the

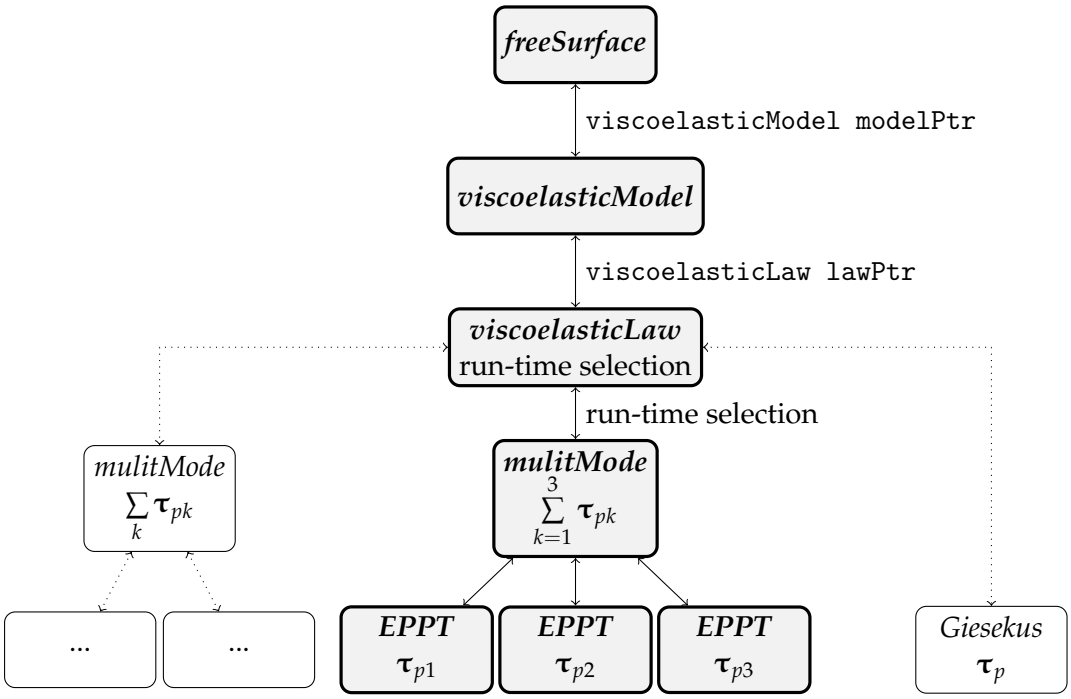


FIGURE 4.3: Flow chart of the calculation of viscoelastic data

following sections, the files shown in Figure 4.4 are edited to enable the calculation of the viscoelastic surface stress tensors.

4.4.1 Adaptions in the *EPPT* class

The EPPT viscoelastic model reflects the Phan-Thien–Tanner constitutive model discussed in Section 3.3.1 in its exponential form. The original class *EPPT* has three member functions. A method named `correctTau()` to calculate the current stress tensor field ($\boldsymbol{\tau}_p$) and a method `tau()` to return the calculated stress field. In addition, there is a method `divTau(volVectorField& U)`, which calculates and returns the coupling term $\boldsymbol{\nabla} \cdot \boldsymbol{\tau}$ for the momentum equation. For the calculation of the viscous stress tensor at the free surface ($\boldsymbol{\tau}_{pf}$) via FAM two additional methods are created. The methods `tauSurfaceCorrect()` and `tauSurface()` compute and return the viscoelastic stress tensor at the finite area patch at the interface. The index f corresponds to values at the cell faces. The concept of cell centred data and data related to the cell faces is discussed in further detail in Section 4.5.

Adaptions in the EPPT.H file

Within the header file the `areaSymmTensorField tauSurface_` variable must be declared. Furthermore, the constructor components must be extended by adding the area fields `Us` (free surface velocity) and `phis` (free surface flux). These area fields are required for the calculating of the viscoelastic stress tensor at the free surface through the FAM. Also, the return function of the surface stress field is defined and the method for the calculation of the surface stress field is declared.

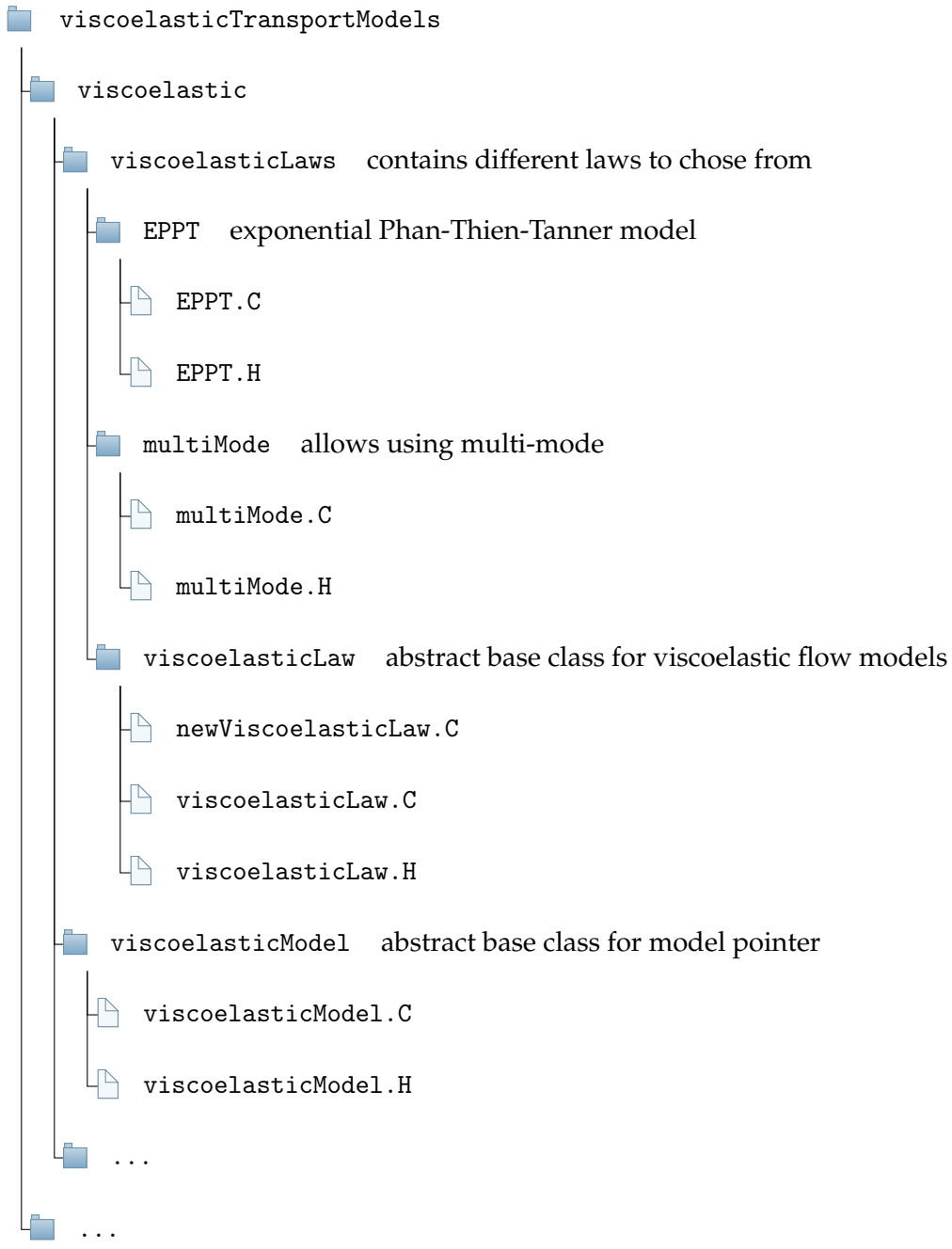


FIGURE 4.4: Directory of the viscoelasticTransportModels library

```
53 class EPTT
54 :
55     public viscoelasticLaw
56 {
57     // Private data
58
59         //- Transported viscoelastic stress
60         volSymmTensorField tau_;
61
62         //- added: Transported viscoelastic surface stress
63         areaSymmTensorField tauSurface_;
...

102     // Constructors
103
104         //- Construct from components
105         EPTT
106         (
107             const word& name,
108             const volVectorField& U,
109             const surfaceScalarField& phi,
110             const areaVectorField& Us, // added
111             const edgeScalarField& phis, // added
112             const dictionary& dict
113         );
...

122     // Member Functions
123
124         //- added: Return the viscoelastic surface stress
125         virtual tmp<areaSymmTensorField> tauSurface() const
126         {
127             return tauSurface_;
128         }
129
130         //- added: Correct the viscoelastic surface stress
131         virtual void tauSurfaceCorrect();
...

145         //- edited: Correct the viscoelastic stress
146         virtual void tauCorrect(); //edited name from correct() to
    tauCorrect()
```

CODE SAMPLE 4.5: Adapting the exponential Phan-Thien–Tanner model for free-surface calculations in `EPPT.H`

Adaptions in the EPPT.C file

As in the header file, the new free surface variables must first be added to the constructor. The `tauSurface` field declared in the header file, is now defined as an input-output object which allows reading and writing various partial stress fields when calculating with *multiMode*.

Code lines `86` to `119` in Code Sample 4.6 represent calculations of the `tauSurface` field according to the Constitutive equations (see Equation 3.10, 3.11 and 3.13).

The code for the calculation of the viscous stress tensor at the free surface through solving a matrix is established analogous to the calculation of the viscoelastic stress tensor associated to the volume field. Methods from the *finiteArea* instead of the *finiteVolume* library must be utilised to process the data. To calculate the gradient of the free surface velocity, the finite area method `fac::grad()` now has to be applied instead of `fvc::grad()`. Furthermore `fam` methods are now applied to the `faSymmTensorMatrix`. The matrix is solved through the `solve()` command (line `118`). An additional dictionary with the selected solution algorithm must be passed here.

```
41 Foam::EPTT::EPTT
42 (
43     const word& name,
44     const volVectorField& U,
45     const surfaceScalarField& phi,
46     const areaVectorField& Us, // added
47     const edgeScalarField& phis, // added
48     const dictionary& dict
49 )
50 :
51     viscoelasticLaw(name, U, phi, Us, phis), // edited
...

62     tauSurface_ // added
63     (
64         IOobject
65         (
66             "tauSurface" + name,
67             U.time().timeName(),
68             U.mesh(),
69             IOobject::MUST_READ,
70             IOobject::AUTO_WRITE
71         ),
72         Us.mesh()
73     ),
...

85 // added
86 void Foam::EPTT::tauSurfaceCorrect()
87 {
88
```

```
89      const tmp<areaTensorField> tL = fac::grad(Us());
90      const areaTensorField& L = tL();
91
92      // Convected derivate term
93      areaTensorField C = tauSurface_ & L;
94
95      // Twice the rate of deformation tensor
96      areaSymmTensorField twoD = twoSymm(L);
97
98       // Stress transport equation
99      Info<< "creating␣faSymmTensorMatrix"<< endl;
100
101     faSymmTensorMatrix tauSurfaceEqn
102     (
103         lambda_*fam::ddt(tauSurface_)
104       + lambda_*fam::div(phis(), tauSurface_)
105      ==
106         etaP_*twoD
107       + lambda_*twoSymm(C)
108       - lambda_*zeta_*symm(tauSurface_ & twoD)
109       - fam::Sp
110         (
111             Foam::exp(epsilon_*lambda_/etaP_*tr(tauSurface_)),
112             tauSurface_
113         )
114
115     );
116
117     tauSurfaceEquationrelax();
118     tauSurfaceEquationsolve(Us().mesh().solutionDict().subDict(tauSurface_.
        name()));
119 }
```

CODE SAMPLE 4.6: Adapting the exponential Phan-Thien–Tanner model for free-surface calculations in `EPPT.C`

4.4.2 Adaptions in the *multiMode* class

The *multiMode* class is inserted upstream to the actual viscoelastic model (Figure 4.3). This class provides the utility of calculating partial elastic stress tensors $\boldsymbol{\tau}_{pk}$ using different model parameters (Equation 3.9). A pointer list `models_` contains a list with the amount of model modes k.

Adaptions in the multiMode.H file

The private class data as well as the constructor is edited according to the header file of the *EPPT* class in Code Sample 4.5. The methods are also declared in accordance with the *EPPT* class.

```
52 class multiMode
53 :
54     public viscoelasticLaw
55 {
56     // Private data
57
58         //- Transported viscoelastic stress
59         mutable volSymmTensorField tau_;
60
61         //- added: Transported viscoelastic surface stress
62         mutable areaSymmTensorField tauSurface_;
63
64         //- List of models
65         PtrList<viscoelasticLaw> models_;
...

82     // Constructors
83
84         //- Construct from components
85         multiMode
86         (
87             const word& name,
88             const volVectorField& U,
89             const surfaceScalarField& phi,
90             const areaVectorField& Us,      // added
91             const edgeScalarField& phis,    // added
92             const dictionary& dict
93         );
...

104         //- added: Return the viscoelastic surface stress
105         virtual tmp<areaSymmTensorField> tauSurface() const;
106
107         //- added: Correct the viscoelastic surface stress
108         virtual void tauSurfaceCorrect();
...

116         //- edited: Correct the viscoelastic stress
117         virtual void tauCorrect();
```

CODE SAMPLE 4.7: Adapting the multi-mode workaround in `multiMode.H`

Adaptions in the multiMode.C file

The changes made in the constructor are similar to the changes made in `EPPT.C` file (Code Sample 4.6) until code line `101`.

A pointer list `models_` is defined to get entries of the model modes for the calculation of the partial elastic stress tensors. Each list entry contains information about the selected

viscoelastic model (line 113: `modelEntries[modelI].keyword()`), flow information that is updated for the free surface calculations with the FA variables (line 114 to 117) and a dictionary containing viscoelastic modelling parameters for each mode individually.

The member functions `multiMode::tauSurfaceCorrect()` in line 140 loops through the model entries calling the actual viscoelastic function, i.e. `EPPT::tauSurfaceCorrect()`, k times with different parameters.

The return function of the tau field `tauSurface()` begins in line 127: After an initialisation by multiplying the whole field with zero, a loop over all partial stress tensors $\boldsymbol{\tau}_{pk}$ calculated to sum up all fields to a total viscoelastic stress tensor and return it.

```
52 Foam::multiMode::multiMode
53 (
54     const word& name,
55     const volVectorField& U,
56     const surfaceScalarField& phi,
57     const areaVectorField& Us,      // added
58     const edgeScalarField& phis,    // added
59     const dictionary& dict
60 )
61 :
62     viscoelasticLaw(name, U, phi, Us, phis), // edited
...

82     // added
83     tauSurface_
84     (
85         IOobject
86         (
87             "tauSurface" + name,
88             U.time().timeName(),
89             U.mesh(),
90             IOobject::NO_READ,
91             IOobject::AUTO_WRITE
92         ),
93         Us.mesh(),
94         dimensionedSymmTensor
95         (
96             "zero",
97             dimensionSet(1, -1, -2, 0, 0, 0, 0),
98             symmTensor::zero
99         )
100    ),
101    models_()
102 {
103    PtrList<entry> modelEntries(dict.lookup("models"));
104    models_.setSize(modelEntries.size());
105
106    forAll (models_, modelI)
107    {
```

```
108         models_.set
109         (
110             modelI,
111             viscoelasticLaw::New
112             (
113                 modelEntries[modelI].keyword(),
114                 U,
115                 phi,
116                 Us,     //added
117                 phis,   //added
118                 modelEntries[modelI].dict()
119             )
120         );
121     }
122 }
123
124
125 // * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
126 // added
127 Foam::tmp<Foam::areaSymmTensorField> Foam::multiMode::tauSurface() const
128 {
129     tauSurface_ *= 0;
130
131     for (label i = 0; i < models_.size(); i++)
132     {
133         tauSurface_ += models_[i].tauSurface();
134     }
135
136     return tauSurface_;
137 }
138
139 // added
140 void Foam::multiMode::tauSurfaceCorrect()
141 {
142         forAll (models_, i)
143     {
144         Info<< "Model mode "  << i+1 << endl;
145         models_[i].tauSurfaceCorrect();
146     }
147
148     tauSurface();
149 }
...

165 // edited: name from correct() to tauCorrect()
166 void Foam::multiMode::tauCorrect()
```

CODE SAMPLE 4.8: Adapting the multi-mode workaround in `multiMode.C`

4.4.3 Adaptions in the *viscoelasticLaw* class

In the `viscoelasticLaw` directory, a class is formulated where the flow information is passed through, prior to *multiMode*, and subsequently to be processed in *EPPT*. The *viscoelasticLaw* class is the base class for the viscoelastic flow models.

Next, the run-time mechanism for the selection of the viscoelastic model is set up. This means viscoelastic model is not hard coded into the solver code but is selected during the execution of the program. The folder `viscoelasticLaw` contains three files: header and C-file are defining the *viscoelastiLaw* class and a third file called `newViscoelasticLaw.C` with a subclass *New* inheriting from *viscoelasticLaw* for the selection of the model.

Adaptions in the viscoelasticLaw.H file

As the header files `multiMode.H` and `EPPT.H` have an include command of this viscoelastic header file, it is sufficient to include the FA methods just in the `viscoelasticLaw.H`-file (line `54` until `58`). In order to make the flow variables available later, they must also be written into the run-time selection table in lines `134` and `135`.

Obviously, constructor and selector are edited as done previously (Code Sample 4.7). In addition to the member functions for surface stress calculations, return functions are added for the surface variables.

```
53 // added: finite area method
54 #include "faCFD.H"
55 #include "fam.H"
56 #include "fac.H"
57 #include "faMatrices.H"
58 #include "faMatrix.H"
...


85
86         //- added: Reference to surface velocity field
87         const areaVectorField& Us_;
88
89         //- added: Reference to edge flux field
90         const edgeScalarField& phis_;
...


107     // Declare run-time constructor selection table
108
109         declareRunTimeSelectionTable
110         (
111             autoPtr,
112             viscoelasticLaw,
113             dictionary,
114             (
115                 const word& name,
116                 const volVectorField& U,
```

```
117                 const surfaceScalarField& phi,
118                 const areaVectorField& Us,      // added
119                 const edgeScalarField& phis,    // added
120                 const dictionary& dict
121             ),
122             (name, U, phi, Us, phis, dict)      // edited
123         );
124
125
126     // Constructors
127
128         //- Construct from components
129         viscoelasticLaw
130         (
131             const word& name,
132             const volVectorField& U,
133             const surfaceScalarField& phi,
134             const areaVectorField& Us,          // added
135             const edgeScalarField& phis         // added
136         );
137
138
139     // Selectors
140
141         //- Return a reference to the selected viscoelastic law
142         static autoPtr<viscoelasticLaw> New
143         (
144             const word& name,
145             const volVectorField& U,
146             const surfaceScalarField& phi,
147             const areaVectorField& Us,          // added
148             const edgeScalarField& phis,        // added
149             const dictionary& dict
150         );
...

179         //- added: Return the surface velocity field
180         const areaVectorField& Us() const
181         {
182             return Us_;
183         }
184
185         //- added: Return the edge flux field
186         const edgeScalarField& phis() const
187         {
188             return phis_;
189         }
190
191         //- added: Return viscoelastic surface stress tensor
192         virtual tmp<areaSymmTensorField> tauSurface() const = 0;
193
```

```
194         //- added: Correct viscoelastic surface stress
195         virtual void tauSurfaceCorrect() = 0;
...


203       //- edited: Correct the viscoelastic stress
204         virtual void tauCorrect() = 0;
```

CODE SAMPLE 4.9: Adapting the class viscoelasticLaw in `viscoelasticLaw.H`

Adaptions in the viscoelasticLaw.C file

Since the *viscoelastiLaw* class is only dedicated to create the run-time table while the run-time selection takes place in a subclass *New*, the reworking of the C-file requires minimal effort. As there are no methods defined, it is necessary to just add the free surface variables to the constructor.

```
39 Foam::viscoelasticLaw::viscoelasticLaw
40 (
41     const word& name,
42     const volVectorField& U,
43     const surfaceScalarField& phi,
44     const areaVectorField& Us,  // added
45     const edgeScalarField& phis // added
46 )
47 :
48     name_(name),
49     U_(U),
50     phi_(phi),
51     Us_(Us),    // added
52     phis_(phis) // added
53 {}
```

CODE SAMPLE 4.10: Adapting the class viscoelasticLaw in `viscoelasticLaw.C`

Adaptions in the newViscoelasticLaw.C file

The subclass *New* searches the dictionary with the rheological parameters for the keyword `type` which relates to the viscoelastic model. If this model is in the previously created run-time table with every available model, a pointer with the model data is created.

```
33 Foam::autoPtr<Foam::viscoelasticLaw> Foam::viscoelasticLaw::New
34 (
35     const word& name,
36     const volVectorField& U,
37     const surfaceScalarField& phi,
38     const areaVectorField& Us,                  // added
```

```
39      const edgeScalarField& phis,                // added
40      const dictionary& dict
41  )
42  {
43      word typeName = dict.lookup("type");
44
45      Info<< "Selecting␣viscoelastic␣model␣" << typeName << endl;
46
47      dictionaryConstructorTable::iterator cstrIter =
48          dictionaryConstructorTablePtr_->find(typeName);
49
50      if (cstrIter == dictionaryConstructorTablePtr_->end())
51      {
52          FatalErrorIn
53          (   // edited
54              "viscoelasticLaw::New(const␣word&␣name,␣const␣volVectorField&,␣
    "
55              "const␣surfaceScalarField&,␣const␣areaVectorField&,␣const␣
    edgeScalarField&)"
56          )   << "Unknown␣viscoelasticLaw␣type␣" << typeName
57              << endl << endl
58              << "Valid␣viscoelasticLaw␣types␣are␣:" << endl
59              << dictionaryConstructorTablePtr_->sortedToc()
60              << exit(FatalError);
61      }
62
63      return autoPtr<viscoelasticLaw>(cstrIter()(name, U, phi, Us, phis, dict
    )); // edited
64  }
```

CODE SAMPLE 4.11: Adapting the subclass *New* of *viscoelasticLaw* in `newViscoelasticLaw.C`

4.4.4 Adaptions in the *viscoelasticModel* class

The class *viscoelasticModel* is the starting point of the chain to select, calculate and return the extra viscoelastic stress throughout the classes discussed in Sections 4.4.1 to 4.4.3. In this class, the dictionary `rheology` is read, which contains the individual modelling parameters for each mode. The objects created with the *viscoelasticModel* class forms the docking point between the main program and the calculation of the stress tensors through a constitutive equation. The implementation of the stress tensors in the main program is topic of the following Section 4.5.

Adaptions in the viscoelasticModel.H file

Constructor components as well as methods are edited as done in the previous header-files (Code Sample 4.7).

```
81      // Constructors
82
83          //- Construct from components
84          viscoelasticModel
85          (
86              const volVectorField& U,
87              const surfaceScalarField& phi,
88              const areaVectorField& Us,      // added
89              const edgeScalarField& phis     // added
90          );
...

101         //- added: Return viscoelastic surface stress tensor
102         virtual tmp<areaSymmTensorField> tauSurface() const;
103
104         //- added: Correct viscoelastic surface stress
105         virtual void tauSurfaceCorrect();
...

104         //- edited: Correct the viscoelastic stress
105         virtual void tauCorrect();
```

CODE SAMPLE 4.12: Adapting *viscoelasticModel* in `viscoelasticModel.H`

Adaptions in the viscoelasticModel.C file

The free surface variables are added to the constructor and the *viscoelasticLaw::New* pointer gets passed the required variables. The free surface methods are added and use the arrow operator to access member functions through the pointer.

```
40 viscoelasticModel::viscoelasticModel
41 (
42     const volVectorField& U,
43     const surfaceScalarField& phi,
44     const areaVectorField& Us,      // added
45     const edgeScalarField& phis     // added
46 )
47 :
48     IOdictionary
49     (
50         IOobject
51         (
52             "viscoelasticProperties",
53             U.time().constant(),
54             U.db(),
55             IOobject::MUST_READ,
56             IOobject::NO_WRITE
57         )
58     ),
```

```
59      lawPtr_(viscoelasticLaw::New(word::null, U, phi, Us, phis, subDict("
        rheology")))    // edited
60  {}
61
62
63  // * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
64
65
66  // added
67  tmp<areaSymmTensorField> viscoelasticModel::tauSurface() const
68  {
69      return lawPtr_->tauSurface();
70  }
71
72  // added
73  void viscoelasticModel::tauSurfaceCorrect()
74  {
75      lawPtr_->tauSurfaceCorrect();
76  }
...

89  // edited
90  void viscoelasticModel::tauCorrect()
91  {
92      lawPtr_->tauCorrect();
93  }
```

CODE SAMPLE 4.13: Adapting *viscoelasticModel* in `viscoelasticModel.C`

4.5 Adaption in the *freeSurface* class

Figure 4.1 shows the solution procedure of the new developed solver named *bubbleInterTrackModel*. The first grey highlighted box represents the calculation of the viscous stress tensor performed through classes presented in the previous section. Updating the interface boundary conditions through methods of the *freeSurface* class is the next step. The *freeSurface* class is the hub of all surface-tracking related procedures and methods. As a results of the implementation of viscoelastic fluid behaviour in the surface-tracking method, several changes have to be made.

The special feature of the surface-tracking method is the presentation of a sharp interface at the phase boundary between the bubble and the liquid. The interface is represented by a finite area mesh that spans the bubble. In classical CFD, the transport equations are numerically approximated for each cell by the FVM. The transport properties, such as pressure and velocity, which are required for the solution of the partial differential equations, refer to values in the cell centroid and are uniform across the cell.

The interface is located exactly at the cell faces of the corresponding gas/liquid cells at the phase boundary. The *freeSurface* class provides methods to transform data stored in those boundary cells of the finite volume mesh into data related to the finite area mesh.

For the purpose of updating the boundary conditions at the phase boundary, an approximation must be made from the cell centred values of the boundary cells to the interface. This is discussed in the following Section 4.5.1. Figure 4.5 shows the phase boundary between the boundary cells and the location of cell centred data.

4.5.1 Update interface boundary conditions

Updating the boundary conditions of velocity and pressure at the interface is an essential step during the surface-tracking procedure. The discretised computational mesh consists of two domains. A surrounding fluid (Fluid A) and the air bubble in the centre (Fluid B). As already shown in Figure 2.4, the two mesh parts are in contact via two geometrically identical surfaces. These two congruent surfaces, S_A and S_B, form the boundary between the two phases and thus the interface. These two interfaces are defined by a set of boundary surfaces. Figure 4.5 illustrates that each face Af on surface S_A has a corresponding and geometrically equal face Bf on surface S_B. Since the fluid properties at the interface change discontinuously, a proper coupling of the transport equations is necessary. This is performed by applying adequate boundary conditions at the interface. At side A of the interface, the dynamic pressure p_A and the normal velocity derivative $\mathbf{n}_A \cdot (\boldsymbol{\nabla}\mathbf{v})_A$ are specified, while at the side B the velocity $\mathbf{v}_B$ and the normal derivative of the dynamic pressure $\mathbf{n}_B \cdot (\boldsymbol{\nabla}p)_B$ are specified. As shown in Figure 4.1, the boundary conditions are updated at the beginning of each iteration [Tuković and Jasak 2012].

The update methods for the interface velocity and pressure boundary conditions are part of the *freeSurface* class. During the initialisation process of the main program, a `freeSurface` object called `interface` is created in the `createFields.H` file. This object is used to call the surface-tracking methods.

```
50 freeSurface interface(mesh, rho, U, p, phi);
```

CODE SAMPLE 4.14: `createFields.H`

From the object created in Code Sample 4.14, the update methods for the boundary conditions are called in the main function. The method `updateBoundaryCondtions()` then

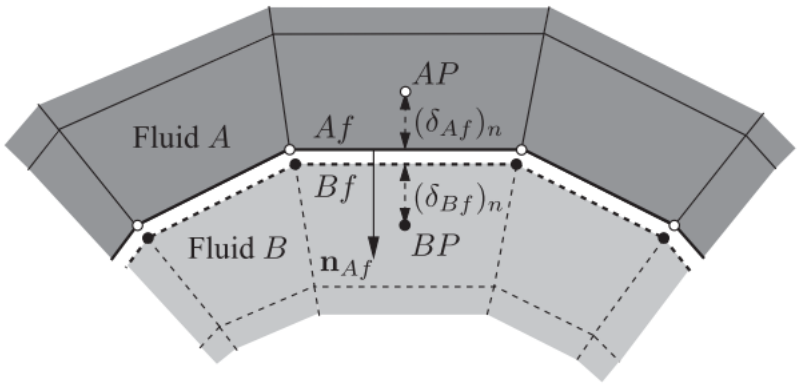


FIGURE 4.5: Representation of the interface with the mesh boundary faces [Tuković and Jasak 2012]

calls individual methods to update pressure and the velocity, as well as methods to calculate the viscoelastic stress field for the volume field and the viscoelastic stress field for the interfacial area field. The following sections describe the modification of the `updateVelocity()` and `updatePressure()` methods, that update the boundary conditions at the interface.

```
77 interface.updateBoundaryConditions();
```

CODE SAMPLE 4.15: `bubbleInterTrackModel.C`

```
1262 void freeSurface::updateBoundaryConditions()
1263 {
1264     tauCorrect();              // added
1265     tauSurfaceCorrect();    // added
1266     updateVelocity();
1267     // updateSurfactantConcentration();
1268     updatePressure();
1269 }
```

CODE SAMPLE 4.16: `freeSurface.C`

4.5.2 Interface velocity via first-order approximation

The calculation of the tangential component of the interface velocity is performed using first-order approximation. As it is shown in Figure 4.5, the cell velocities refer to the velocities at point AP and BP, respectively. Starting with the viscoelastically adapted **velocity jump condition** (Equation 3.43), the tangential component of the velocity vector $\mathbf{v}_t = (\mathbf{I} - \mathbf{nn}) \cdot \mathbf{v}$ and the normal velocity component $\boldsymbol{\nu}_n = \mathbf{n} \cdot \mathbf{v}$ are introduced. A relationship between the normal derivative of tangential velocity on the two sides of the interface is established in a similar manner as TUKOVIĆ and JASAK [2012] proposed.

$$\begin{aligned} &\mu_{Bs}(\mathbf{n} \cdot \boldsymbol{\nabla}\mathbf{v}_t)_B - \mu_{As}(\mathbf{n} \cdot \boldsymbol{\nabla}\mathbf{v}_t)_A = \\ &- \boldsymbol{\nabla}_S\sigma - (\mu_{Bs} - \mu_{As})\boldsymbol{\nabla}_S\boldsymbol{\nu}_n - \mathbf{n} \cdot (\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap}) + \mathbf{n}\Big[\mathbf{nn} : (\boldsymbol{\tau}_{Bp} - \boldsymbol{\tau}_{Ap})\Big] \end{aligned} \tag{4.3}$$

From Equation 4.3, the tangential component of the interface velocity is calculated using first-order approximation from the cell centroids (AP and BP) to the cell faces (Af and Bf):

$$\begin{aligned} &\mu_{Bs}\left(\frac{\mathbf{v}_{BP} - \mathbf{v}_{Bf}}{(\delta_{Bf})_n}\right)_t - \mu_{As}\left(\frac{\mathbf{v}_{Af} - \mathbf{v}_{AP}}{(\delta_{Af})_n}\right)_t = \\ &- \boldsymbol{\nabla}_S\sigma - (\mu_{Bs} - \mu_{As})(\boldsymbol{\nabla}_S\boldsymbol{\nu}_n)_{Af} - \mathbf{n}_f \cdot (\boldsymbol{\tau}_{Bpf} - \boldsymbol{\tau}_{Apf}) + \mathbf{n}_f\Big[\mathbf{n}_f\mathbf{n}_f : (\boldsymbol{\tau}_{Bpf} - \boldsymbol{\tau}_{Apf})\Big], \end{aligned} \tag{4.4}$$

where $(\mathbf{v}_{AP})_t$ and $(\mathbf{v}_{BP})_t$ are the tangential components of velocity in the centroid of cells while $(\delta_{Af})_n$ and $(\delta_{Bf})_n$ are the normal distances from centroids of the cells to the

interface, $\mathbf{n}_f = \mathbf{n}_{Af} = -\mathbf{n}_{Bf}$ is the unitary normal on the face f and $\boldsymbol{\tau}_{pf}$ is the polymeric contribution to the stress tensor at the face. From the **kinematic condition** follows at the faces $(\mathbf{v}_{Af})_t = (\mathbf{v}_{Bf})_t$ and leads to:

$$\begin{aligned}\left(\frac{\mu_{As}}{(\delta_{Af})_n} + \frac{\mu_{Bs}}{(\delta_{Bf})_n}\right)(\mathbf{v}_{Af})_t &= \frac{\mu_{Bs}}{(\delta_{Bf})_n}(\mathbf{v}_{BP})_t + \frac{\mu_{As}}{(\delta_{Af})_n}(\mathbf{v}_{AP})_t \\ + \boldsymbol{\nabla}_s\sigma + (\mu_{Bs} - \mu_{As})(\boldsymbol{\nabla}_S\boldsymbol{\nu}_n)_{Af} &+ \mathbf{n}_f \cdot (\boldsymbol{\tau}_{Bpf} - \boldsymbol{\tau}_{Apf}) - \mathbf{n}_f\Big[\mathbf{n}_f\mathbf{n}_f : (\boldsymbol{\tau}_{Bpf} - \boldsymbol{\tau}_{Apf})\Big].\end{aligned} \tag{4.5}$$

Assuming that the air inside the bubble has newtonian fluid behaviour with $\boldsymbol{\tau}_{Bpf} = 0$ leads the equation below. Due to the additive composition of the total viscoelastic stress tensor (Equation 3.6), the elastic contribution can be identified and is the differentiating part in comparison with the work of TUKOVIĆ and JASAK [2005; 2008; 2012].

$$\begin{aligned}\left(\frac{\mu_{As}}{(\delta_{Af})_n} + \frac{\mu_{Bs}}{(\delta_{Bf})_n}\right)(\mathbf{v}_{Af})_t &= \frac{\mu_{As}}{(\delta_{Bf})_n}(\mathbf{v}_{BP})_t + \frac{\mu_{As}}{(\delta_{Af})_n}(\mathbf{v}_{AP})_t \\ + \boldsymbol{\nabla}_S\sigma + (\mu_{Bs} - \mu_{As})(\boldsymbol{\nabla}_S\boldsymbol{\nu}_n)_{Af} &\underbrace{-\mathbf{n}_f \cdot (\boldsymbol{\tau}_{Apf}) + \mathbf{n}_f\Big[\mathbf{n}_f\mathbf{n}_f : (\boldsymbol{\tau}_{Apf})\Big]}_{\text{elastic contribution}}\end{aligned} \tag{4.6}$$

The following Code Sample 4.17 shows the computational implementation of Equation 4.6 into to `updateVelocity()` method. Although two normal vector fields are already defined at the beginning of the method, an additional vector field is required. In line `1281` in Code Sample 4.17, a normal vector field of type `areaVectorField` is defined based on the FA mesh. This is necessary, because for the calculation of the surface stress tensor an `areaSymmTensorField` is returned and the data types have to be compatible. Line `1359` to `1392` represent the calculations performed in Equation 4.6, with the viscoelastic contribution added in line `1388` and `1390`. `tauSurface()` is the return function of the previously calculated polymeric surface stress tensor through `tauSurfaceCorrect()`. The normal and tangential velocity components add up to the velocity of free surface `Us` in line `1394` and the boundary condition is updated.

```
1273 void freeSurface::updateVelocity()
1274 {
1275     if(twoFluids())
1276     {
1277       // calculate normal vector from boundary patch
1278         vectorField nA = mesh().boundary()[aPatchID()].nf();
1279
1280         //added: face area normal as areaVectorField
1281         areaVectorField nAs = aMesh().faceAreaNormals();
1282
1283       // calculate normal vector from boundary patch
1284         vectorField nB = mesh().boundary()[bPatchID()].nf();
 ...
```

```
1359        vectorField UtFs = muFluidA().value()*DnA*UtPA
1360            + muFluidB().value()*DnB*UtPB;
1361
1362          vectorField UnFs =
1363              nA*phi_.boundaryField()[aPatchID()]
1364             /mesh().boundary()[aPatchID()].magSf();
1365
1366          Us().internalField() += UnFs - nA*(nA & Us().internalField());
1367
1368          correctUsBoundaryConditions();
1369
1370          UtFs -= (muFluidA().value() - muFluidB().value())*
1371              ( fac::grad( Us() ) & aMesh().faceAreaNormals() )().
     internalField();
 ...

1385          UtFs += tangentialSurfaceTensionForce;
1386
1387        // added: viscoelastic stress term
1388          UtFs -= (nAs & tauSurface());
1389
1390          UtFs += (nAs * (nAs & (nAs & tauSurface())));
1391
1392          UtFs /= muFluidA().value()*DnA + muFluidB().value()*DnB + VSMALL;
1393
1394          Us().internalField() = UnFs + UtFs;
1395          correctUsBoundaryConditions();
 ...     }

 ... }
```

CODE SAMPLE 4.17: Changes in `updateVelocity()` in`freeSurface.C` to calculate interface velocity

4.5.3 Interface velocity derivative

Again deriving from the adapted **velocity jump condition** (Equation 3.43). The normal velocity derivative at the face Af is specified using the updated interface velocity from Equation 4.6:

$$\begin{aligned}
&\mu_{As}\Big[(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) + \mathbf{n}\boldsymbol{\nabla}_S\cdot\mathbf{v}\Big]_A \\
&= \mu_{Bs}\underbrace{\Big[(\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}) + \mathbf{n}\boldsymbol{\nabla}_S\cdot\mathbf{v}\Big]_B}_{\mathbf{n}\cdot(\boldsymbol{\nabla}\mathbf{v}_t)_B} \\
&+ \boldsymbol{\nabla}_S\sigma + (\mu_{Bs}-\mu_{As})\boldsymbol{\nabla}_S\mathbf{v}\cdot\mathbf{n} \\
&+ \mathbf{n}\cdot(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap}) - \mathbf{n}\Big[\mathbf{nn}:(\boldsymbol{\tau}_{Bp}-\boldsymbol{\tau}_{Ap})\Big].
\end{aligned} \tag{4.7}$$

Rearrangement leads to the following:

$$\begin{aligned}\mu_{As}\mathbf{n}_f \cdot (\boldsymbol{\nabla}\mathbf{v})_{Af} = \mu_{Bs}\mathbf{n}_f \cdot (\boldsymbol{\nabla}\mathbf{v}_t)_B \\ - \mu_{As}\mathbf{n}_f(\boldsymbol{\nabla}_S \cdot \mathbf{v})_{Af} \\ + \boldsymbol{\nabla}_S\sigma + (\mu_{Bs} - \mu_{As})(\boldsymbol{\nabla}_S\mathbf{v})_{Af} \cdot \mathbf{n}_f \\ + \mathbf{n}_f \cdot (\boldsymbol{\tau}_{Bpf} - \boldsymbol{\tau}_{Apf}) - \mathbf{n}_f\Big[\mathbf{n}_f\mathbf{n}_f : (\boldsymbol{\tau}_{Bpf} - \boldsymbol{\tau}_{Apf})\Big].\end{aligned} \tag{4.8}$$

Applying first-order approximation to $(\boldsymbol{\nabla}\mathbf{v}_t)_B$ and the kinematic condition $\mathbf{v}_{Af} = \mathbf{v}_{Bf}$ results, with the assumption of viscoelasticity only in Fluid A, in

$$\begin{aligned}\mu_{As}\mathbf{n}_f \cdot (\boldsymbol{\nabla}\mathbf{v})_{Af} = \mu_{Bs}\frac{\mathbf{v}_{BP} - \mathbf{v}_{Af}}{(\delta_{Af})_n} \\ - \mu_{As}\mathbf{n}_f(\boldsymbol{\nabla}_S \cdot \mathbf{v})_{Af} \\ + \boldsymbol{\nabla}_S\sigma + (\mu_{Bs} - \mu_{As})(\boldsymbol{\nabla}_S\mathbf{v})_{Af} \cdot \mathbf{n}_f \\ \underbrace{-\mathbf{n}_f \cdot (\boldsymbol{\tau}_{Apf}) + \mathbf{n}_f\Big[\mathbf{n}_f\mathbf{n}_f : (\boldsymbol{\tau}_{Apf})\Big]}_{\text{elastic contribution}}.\end{aligned} \tag{4.9}$$

The implementation of the elastic contribution follows the format presented in Code Sample 4.17. Code lines 1434 and 1436 represent the elastic contribution of the interface velocity derivative.

```
1273 void freeSurface::updateVelocity()
1274 {
 ...

1425         // Update fixedGradient boundary condition on patch A
1426         vectorField nGradU =
1427             muFluidB().value()*(UtPB - UtFs)*DnA
1428           + tangentialSurfaceTensionForce
1429           - muFluidA().value()*nA*fac::div(Us())().internalField()
1430           + (muFluidB().value() - muFluidA().value())
1431            *(fac::grad(Us())().internalField()&nA);
1432
1433       // added: vicsoelastic stress term
1434         nGradU -= (nAs & tauSurface());
1435
1436         nGradU += (nAs * (nAs & (nAs & tauSurface())));
1437
1438         nGradU /= muFluidA().value() + VSMALL;
 ... }
```

CODE SAMPLE 4.18: Changes in `updateVelocity()` in `freeSurface.C` to calculate interface velocity derivative

4.5.4 Update interface pressure boundary condition

Updating the pressure boundary condition at the free surface is done via the **pressure jump condition** (Equation 3.37). Again, a discretisation of the equation is required. The fact that the pressure is a surface force, makes the procedure less extensive. There is no need to approximate from the cell centroid to the cell face, since the values for the pressure already refer to the cell faces. With indication of the cell faces the following equation for the dynamic pressure at face Af (see Figure 4.5) is acquired:

$$p_{Af} = p_{Bf} - (\rho_A - \rho_B)\mathbf{g} \cdot \mathbf{r}_{Af} - (\sigma\kappa)_{Af} + 2(\mu_{Bs} - \mu_{As})(\boldsymbol{\nabla}_S \cdot \mathbf{v})_{Af} + \mathbf{n}_f\mathbf{n}_f : (\boldsymbol{\tau}_{Apf}) \quad (4.10)$$

As it is presented in Code Sample 4.19: At first, a surface based area normal vector is created in line 1587. The pressure pA is then calculated with the discretised pressure jump condition (Equation 4.10) from line 1594 until line 1646 with the elastic contribution added in line 1632.

```
1586 void freeSurface::updatePressure()
1587 {
1588     // calculate normal vector in free surface
1589     vectorField nA = mesh().boundary()[aPatchID()].nf();
1590
1591    // added: face area normal as areaVectorField
1592     areaVectorField nAs = aMesh().faceAreaNormals();
1593
1594     if(twoFluids())
1595     {
1596         scalarField pA =
1597             interpolatorBA().faceInterpolate
1598             (
1599                 p().boundaryField()[bPatchID()]
1600             );
1601       // free surface curvature
1602         const scalarField& K = aMesh().faceCurvatures().internalField();
 ...

1617             pA -= cleanInterfaceSurfTension().value()*K;
 ...


1627
1628         pA -= 2.0*(muFluidA().value() - muFluidB().value())
1629             *fac::div(Us())().internalField();
1630
1631       // added: viscoelastic stress term
1632         pA += (nAs & (nAs & (tauSurface())));
1633
1634 //         vector R0 = gAverage(mesh().C().boundaryField()[aPatchID()]);
1635         vector R0 = vector::zero;
1636
1637         pA -= (rhoFluidA().value() - rhoFluidB().value())*
```

```
1638            (
1639                g_.value()
1640              & (
1641                    mesh().C().boundaryField()[aPatchID()]
1642                  - R0
1643                )
1644            );
1645
1646        p().boundaryField()[aPatchID()] == pA;
 ...    }
 ...}
```

CODE SAMPLE 4.19: Changes in `updatePressure()` in `freeSurface.C` to calculate interface pressure

4.5.5 Further adaption in the *freeSurface* class

The functions of the viscoelastic model are supposed to be called from within the class *freeSurface* via a variable named `modelPtr` of type *viscoelasticModel*. For this purpose, flow information must be passed to the `modelPtr`, like it is done with the `lawPtr` in Code Sample 4.11, when the *freeSurface* object and thus the `modelPtr` is created. If only the volume fields `U` and `phi` were used for calculations, this would not be a problem, because those fields are construction elements of *freeSurface* and are available while the object is created. The fields `Us` and `phis` however, do not belong to the construction elements and correspond to the category demand-driven data. Meaning those fields are defined by an extra method which creates these variable during the construction of the object. Their calculation is part of the class' functionality. These variables are initialised as null pointers and assigned with a value the first time used. The same issue accounts for the `modelPtr` itself and thus requires the same work around. As there is a variety of demand-driven data in *freeSurface*, all these methods are located in an extra file called `makeFreeSurfaceData.C`.

Adaptions in the freeSurface.H file

First of all, the header file of the *viscoelasticModel* must be included in order to access the transport models. As mentioned above, the object `modelPtr_`, through which the viscoelastic methods are called, is initially declared as a pointer and will be dereferenced later via a return function called `modelPtr`. In addition, the method `makeModelPtr` is declared, which is executed the first time when the return `modelPtr` is called and creates the actual *viscoelasticModel* object with flow data. In the nomenclature of OpenFOAM, internal variables are marked by an underscore and are usually made accessible via a return function, as it is also the case here. Subsequently, the functions of the viscoelastic model are declared in *freeSurface*. An additional method named `calculateTauTotalField` is declared. This function calculates a total stress field for the whole fluid domain to have this field available for later flow analysis.

```
52 #include "viscoelasticModel.H"                          // added
...

193     //- Viscoelastic law
194         mutable viscoelasticModel* modelPtr_;       // added
...

198     // Make demand-driven data
199
200             void makeInterpolators();
201             void makeControlPoints();
202             void makeMotionPointsMask();
203             void makeDirections();
204             void makeTotalDisplacement();
205             void readTotalDisplacement();
206             void makeFaMesh() const;
207             void makeUs() const;
208             void makePhis() const;
209             void makeModelPtr() const;              // added
210             void makePhi();
211             void makeDdtPhi();
212             void makeSurfactConc() const;
213             void makeSurfaceTension() const;
214             void makeSurfactant() const;
215             void makeFluidIndicator();
...

265         //- added: Calculate total stress tensor field
266         volSymmTensorField calculateTauTotalField(volVectorField& U_,
    volScalarField& fluidIndicator_) const;
267
268         //- added: Return Surface stress tensor
269         virtual tmp<areaSymmTensorField> tauSurface() const;
270
271         //- added: Correct viscoelastic surface stress
272         virtual void tauSurfaceCorrect();
273
274         //- added: Return the viscoelastic stress tensor
275         virtual tmp<volSymmTensorField> tau();
276
277         //- added: Return the coupling term for the momentum equation
278         virtual tmp<fvVectorMatrix> divTau(volVectorField& U_);
279
280         //- added: Correct the viscoelastic stress
281         virtual void tauCorrect();
282
283         //- added: Read viscoelasticProperties dictionary
284         virtual bool read();
...
```

```
437       //- Return free-surface velocity field
438         areaVectorField& Us();
439
440         //- added: Return free-surface velocity field
441         const areaVectorField& Us() const;
442
443         //- Return free-surface fluid flux field
444         edgeScalarField& Phis();
445
446         //- added: Return free-surface fluid flux field
447         const edgeScalarField& Phis() const;
448
449         //- added: Return viscoity model object
450         viscoelasticModel& modelPtr();
451
452         //- added: Return viscosity model object
453         const viscoelasticModel& modelPtr() const;
```

CODE SAMPLE 4.20: Adapting *freeSurface* in `freeSurface.H`

Adaptions in the makeFreeSurfaceData.C file

The method `makeModelPtr()` declared in the header file to create the variable `modelPtr_` is defined here. The method is created analogous to the other functions for making the demand-driven data and is passing the flow variables to the pointer. As already described, the internal variables marked with an underscore are made accessible via a return function for each variable. First, it is checked if the pointer already exists, otherwise the make function is called, then the pointer is dereferenced and returned as a variable. Although this dereferencing and return function is already available for `phis`, a constant copy is created. The same applies to the `modelPtr()` method. It is necessary to have constant copies of these methods to make them usable in constant and non-constant methods later on.

```
 32 #include "viscoelasticLaw.H"              // added
...

617 // added
618 void freeSurface::makeModelPtr() const
619 {
620     if (debug)
621     {
622         Info<< "freeSurface::makeModelPtr()␣:␣"
623             << "making␣viscosity␣law␣pointer"
624             << endl;
625     }
626
627
628     // It is an error to attempt to recalculate
```

```
629     // if the pointer is already set
630     if (modelPtr_)
631     {
632         FatalErrorIn("freeSurface::makeModelPtr()")
633             << "viscosity␣ModelPtr␣alreay␣exists"
634             << abort(FatalError);
635     }
636    modelPtr_= new viscoelasticModel(U_, phi_, Us(), Phis());
637 }
...

940 // added
941 const edgeScalarField& freeSurface::Phis() const
942 {
943     if (!phisPtr_)
944     {
945         makePhis();
946     }
947
948     return *phisPtr_;
949 }
950
951 // added
952 viscoelasticModel& freeSurface::modelPtr()
953 {
954     if (!modelPtr_)
955     {
956         makeModelPtr();
957     }
958
959     return *modelPtr_;
960 }
961
962 // added
963 const viscoelasticModel& freeSurface::modelPtr() const
964 {
965     if (!modelPtr_)
966     {
967         makeModelPtr();
968     }
969
970     return *modelPtr_;
971 }
```

CODE SAMPLE 4.21: Adapting *freeSurface* in `makeFreeSurfaceData.C`

Adaptions in the freeSurface.C file

The code sample below shows the changes in the *freeSurface* C-file. First, the newly created variable `modelPtr_` is included in the clear method for the demand driven data and

initialised as a null pointer in line 180.

In line 134 the variable `muFluidA_` is edited. The two variables `muFluidA_` and `muFluidB_` represent the solvent viscosities μ_s. Originally, the values were read in from a dictionary, where all fluid properties are listed. Since the parameters for the calculation of the viscosity of Fluid A are now in a separate dictionary, the variable is only initialised with the value 0 and the correct dimension. If the viscous stress is calculated using the *multiMode* functionality, an equivalent part of the solvent viscosity μ_{sk} is assigned to each model mode to calculate the coupling term for the momentum equation $\boldsymbol{\nabla} \cdot \boldsymbol{\tau}$. The sum of the individual solvent viscosities is giving the total viscosity and is calculated between line 184 and 207.

$$\mu_s = \sum_k \mu_{sk} \tag{4.11}$$

From line 392 on, the viscoelastic methods are implemented. The first method `calculateTauTotalField` is supposed to calculate and return the total stress tensor field as a sum of viscous and elastic contribution. The calculation of the viscous contribution is finished but the elastic part is missing. The commented-out code line 407 shows the expression of the elastic contribution. Due to test purposes, it was commented-out and then overlooked to put back. Further below are the methods for accessing the viscoelastic methods via `modelPtr()`.

From line 1991 on, the method for updating the properties has been adapted analogous to the calculation of viscosity from line 184.

```
70 void freeSurface::clearOut()
71 {
72     deleteDemandDrivenData(interpolatorABPtr_);
73     deleteDemandDrivenData(interpolatorBAPtr_);
74     deleteDemandDrivenData(controlPointsPtr_);
75     deleteDemandDrivenData(motionPointsMaskPtr_);
76     deleteDemandDrivenData(pointsDisplacementDirPtr_);
77     deleteDemandDrivenData(facesDisplacementDirPtr_);
78     deleteDemandDrivenData(totalDisplacementPtr_);
79     deleteDemandDrivenData(aMeshPtr_);
80     deleteDemandDrivenData(UsPtr_);
81     deleteDemandDrivenData(phisPtr_);
82     deleteDemandDrivenData(modelPtr_);                    // added
83     deleteDemandDrivenData(surfactConcPtr_);
84     deleteDemandDrivenData(surfaceTensionPtr_);
85     deleteDemandDrivenData(surfactantPtr_);
86     deleteDemandDrivenData(fluidIndicatorPtr_);
87 }
...

132     muFluidA_
133     (
134         "muFluidA",dimensionSet(1,-1,-1,0,0,0,0), scalar(0)       // edited
135     ),
```

```
136     muFluidB_
137     (
138         this->lookup("muFluidB")
139     ),
...

165     smoothing_(false),
166     interpolatorABPtr_(NULL),
167     interpolatorBAPtr_(NULL),
168     controlPointsPtr_(NULL),
169     motionPointsMaskPtr_(NULL),
170     pointsDisplacementDirPtr_(NULL),
171     facesDisplacementDirPtr_(NULL),
172     totalDisplacementPtr_(NULL),
173     aMeshPtr_(NULL),
174     UsPtr_(NULL),
175     phisPtr_(NULL),
176     surfactConcPtr_(NULL),
177     surfaceTensionPtr_(NULL),
178     surfactantPtr_(NULL),
179     fluidIndicatorPtr_(NULL),
180     modelPtr_(NULL)                                          // added
181 {
182
183     // added
184    Info << "Calculating␣muFluidA" << endl;
185
186     IOdictionary viscoelasticProperties
187     (
188         IOobject
189         (
190             "viscoelasticProperties",
191             Ub.mesh().time().constant(),
192             Ub.mesh(),
193             IOobject::MUST_READ,
194             IOobject::NO_WRITE
195         )
196     );
197
198     //Sum up solvent related viscosities
199     PtrList<entry> modelEntries_(viscoelasticProperties.subDict("rheology")
        .lookup("models"));
200
201    Info << "Model␣entry␣list␣created" << endl;
202
203     forAll(modelEntries_, modelI)
204     {
205         muFluidA_ += dimensionedScalar(modelEntries_[modelI].dict().lookup(
        "etaS"));
206     }
207     Info << "muFluidA␣cualculated␣as:" << muFluidA_.value() << endl;
```

```
...

// added
volSymmTensorField freeSurface::calculateTauTotalField(volVectorField& U_,
    volScalarField& fluidIndicator_) const
{


   // Velocity gradient tensor
    const tmp<volTensorField> tL = fvc::grad(U_);
    const volTensorField& L = tL();


    // Twice the rate of deformation tensor
    volSymmTensorField twoD = twoSymm(L);

   volSymmTensorField tau_ = (fluidIndicator_ * (muFluidA_ - muFluidB_) +
    muFluidB_ ) * twoD;

   // tau_ += fluidIndicator_ * tau()();

   return tau_;
}

// added
tmp<areaSymmTensorField> freeSurface::tauSurface() const
{
    return modelPtr().tauSurface();
}

// added
void freeSurface::tauSurfaceCorrect()
{
    modelPtr().tauSurfaceCorrect();
}

// added
tmp<volSymmTensorField> freeSurface::tau()
{
    return modelPtr().tau();
}

// added
tmp<fvVectorMatrix> freeSurface::divTau(volVectorField& U_)
{
    return modelPtr().divTau(U_);
}

// added
void freeSurface::tauCorrect()
{
```

```
439     modelPtr().tauCorrect();
440 }
441
442 // added
443 bool freeSurface::read()
444 {
445     if (regIOobject::read())
446     {
447         return true;
448     }
449     else
450     {
451         return false;
452     }
453 }
454
...

1991 void freeSurface::updateProperties()
1992 {
1993    //added
1994    muFluidA_ = dimensionedScalar("muFluidA",dimensionSet(1,-1,-1,0,0,0,0),
     scalar(0));
1995
1996     Info << "Calculating muFluidA" << endl;
1997
1998     //Sum up solvent related viscosities
1999     PtrList<entry> modelEntries_(subDict("rheology").lookup("models"));
2000
2001    Info << "Moldel entry list created" << endl;
2002
2003     forAll(modelEntries_, modelI)
2004     {
2005         muFluidA_ += dimensionedScalar(modelEntries_[modelI].dict().lookup(
     "etaS"));
2006     }
2007     Info << "muFluidA is calculated as:" << muFluidA_ << endl;
2008
2009    //muFluidA_ = dimensionedScalar(this->lookup("muFluidA"));
2010
2011     muFluidB_ = dimensionedScalar(this->lookup("muFluidB"));
2012
2013     rhoFluidA_ = dimensionedScalar(this->lookup("rhoFluidA"));
2014
2015     rhoFluidB_ = dimensionedScalar(this->lookup("rhoFluidB"));
2016
2017     g_ = dimensionedVector(this->lookup("g"));
2018
2019     cleanInterfaceSurfTension_ =
2020         dimensionedScalar(this->lookup("surfaceTension"));
2021 }
```

CODE SAMPLE 4.22: Adapting *freeSurface* in `makeFreeSurfaceData.C`

One of the last steps in the main program in the file `bubbleInterTrackFoam.C` is the calculation of the total force as the sum of the total viscous force and the pressure force. The calculation of the viscous force requires editing due to the viscoelastic implementation. The elastic contribution of the viscoelastic force is calculated from the extra stress tensor and the face area normal.

```
95 vector freeSurface::totalViscousForce() const
96 {
97     const scalarField& S = aMesh().S();
98     const vectorField& n = aMesh().faceAreaNormals().internalField();
99     const areaVectorField& nAs = aMesh().faceAreaNormals();
100
101     vectorField nGradU =
102         U().boundaryField()[aPatchID()].snGrad();
103
104     vectorField viscousForces =
105       - muFluidA().value()*S
106        *(
107             nGradU
108           + (fac::grad(Us())().internalField()&n)
109           - (n*fac::div(Us())().internalField())
110         );
111
112
113     viscousForces += nAs & tauSurface();// added: viscoelastic stress term
114
115     return gSum(viscousForces);
116 }
```

CODE SAMPLE 4.23: Calculating total viscous force in `freeSurface.C`

Chapter 5

Simulation Procedure

5.1 Introduction

To test the viscoelastic implementation, simulations with the newly developed solver are performed. At first, the simulation of a free-rising single bubble with parameters corresponding to water and a pseudo 2D mesh is carried out and the results are later compared with the results of the tutorial case of the original *bubbleInterTrackFoam* solver. To find out how the implemented viscoelastic model reacts, the viscoelastic model parameters are slowly moved from totally ideal behaviour to a higher degree of viscoelasticity. Besides this, 3D simulations are performed with the original *bubbleInterTrackFoam* and different numerical parameters as a preparation for 3D simulations with a modified solver.

5.2 Pre-processing

The pre-processing is the first step in setting up a new simulation. This includes creating the geometry and meshing it, setting boundary and initial conditions, defining the numerical configuration and providing the physical model parameters. All these parameters are located in files in the case directory (Figure 5.1).

The general OpenFOAM case directory consists of three main folders. A folder `constant` with the mesh data as well as dictionaries with the physical parameters for the calculation. The numerical configuration in located in the `system` folder. Here, dictionaries with for discretisation schemes and solution schemes are found, as well as solver control data. Boundary conditions and initial field values are located in the `0` folder.

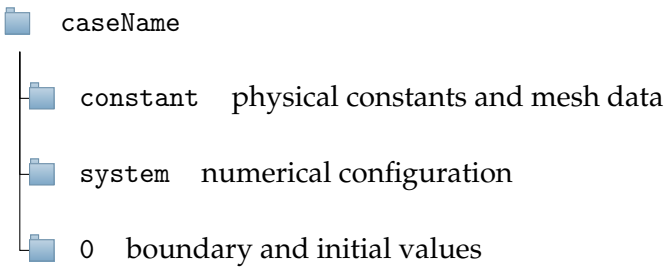
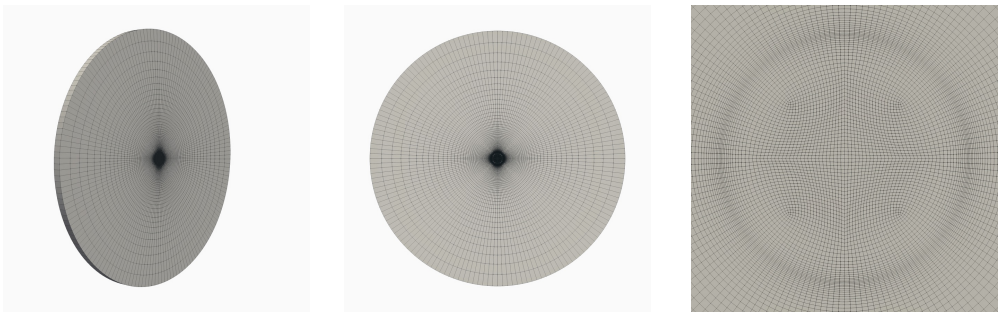


FIGURE 5.1: Case directory



(A) Side view (B) Centred view (C) Bubble view

FIGURE 5.2: Pseudo 2D mesh with 13.7 thousand hexahedral cells

5.2.1 Mesh generation

The simulations to evaluate the new developed *bubbleInterTrackModel* solver are carried out with a pseudo two-dimensional mesh. The pseudo 2D mesh is already available and can be taken from the foam-extend-4.0 tutorial case `bubble2D_r0.75mm`. Furthermore, two 3D meshes are created to test the original *bubbleInterTrackFoam* solver beyond the tutorial case provided with foam-extend-4.0. A 3D geometry is generated and meshed twice with different settings in StarCCM+ 13.02. The StarCCM+ meshes are converted into FOAM meshes using the `ccm26ToFoam` utility.

Pseudo 2D Mesh

Figure 5.2 shows the pseudo 2D mesh. The mesh consists of 13,800 hexahedral cells. The plane, with a thickness of one cell, has a radius of 150 mm with a bubble in the centre of 0.75 mm radius. The bubble is meshed finer with an extra refinement at the phase boundary. A log of the quality report is found in Appendix B.

3D Meshes

The two 3D meshes (Figure 5.3 and Figure 5.4) are created in StarCCM+ 13.02. A spherical geometry with a radius of 150 mm with an additional sphere of 0.75 mm radius in the centre of is created. The polyhedral unstructured mesh operation is applied to the geometry. A surface growth rate of 1.2 is applied to the imprinted surface of the two spheres to ensure a finer mesh at the phase boundary. Two meshes are then created from the geometry. The first mesh is a coarse one with a base size of 0.7 mm resulting in a polyhedral mesh with 406 thousand cells. The second finer mesh has got a base size of 0.5 mm and 1.2 million polyhedral cells. The StarCCM+ mesh files are converted with the `ccm26ToFoam` utility into a foam-extend-4.0 compatible file systems. Logs of the mesh quality reports are found in Appendix B.

5.2.2 Boundary and initial conditions

These boundary conditions are updated prior the pressure/velocity coupling according to the jump conditions. The applied boundary conditions are shown in Table 5.1. The

(A) Side view (B) Centred view (C) Bubble view

FIGURE 5.3: 3D mesh with 407 thousand polyhedral cells

(A) Side view (B) Centred view (C) Bubble view

FIGURE 5.4: 3D mesh with 1.2 million polyhedral cells

variables `p` and `U` refer to pressure and velocity, whereas `motionU` denotes to the velocity of the interface $\mathbf{v}_S$. The stress tensor boundary conditions are chosen with a fixed value of zero inside the bubble, as the extra stress tensor is zero in the whole bubble. For the `freeSurface` patch outside the bubble, a boundary condition of zero gradient is chosen arbitrarily. Figure 5.5 illustrates the locations of the boundaries. As for all partial stress tensors the same boundary conditions are applied, all elements are abbreviated in this work as `tau*` and `tauSurface*`, respectively. In case of a pseudo 2D simulation, an additional boundary is required that refers to the front and back panes enclosing the single layer of cells (see Figure 5.2). In OpenFOAM, the boundary condition for this is `empty` [OpenFOAM® 2020b].

TABLE 5.1: Boundary conditions

	space	freeSurfaceShadow	freeSurface
`motionU`	fixedValue, 0	fixedValue, 0	fixedValue, 0
`p`	zeroGradient	fixedValue, 0	fixedGradient, 0, 0
`tau*`	zeroGradient	fixedValue, 0	zeroGradient
`tauSurface*`	zeroGradient	fixedValue, 0	zeroGradient
`U`	inletOutlet, (0, -1e-6, 0)	fixedValue, 0	fixedGradient, 0, 0

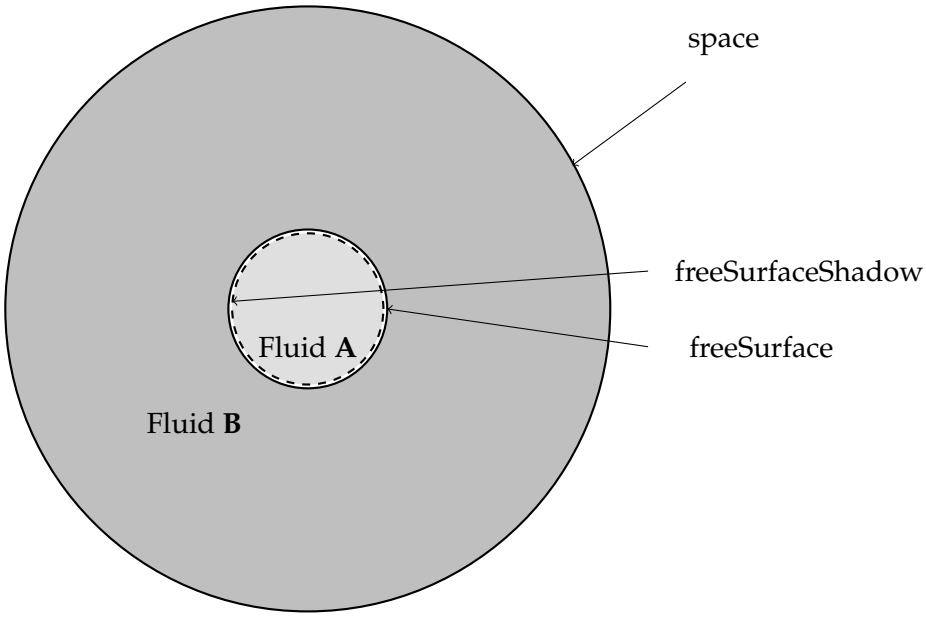


FIGURE 5.5: Boundary conditions

TABLE 5.2: `fvSolution` sub-dictionary: solver

	solver	preconditioner	tolerance	relTol
`p`	`PCG`	`DIC`	1e-06	0
`U`	`BiCGStab`	`DILU`	1e-05	0
`tau*`	`BiCGStab`	`ILU0`	1e-6	0

5.2.3 Iterative methods

The selected methods for preconditioning of matrices and linear-solvers to evaluate the corresponding discretised equations are shown Table 5.2 and Table 5.3. The pressure equations solved with the Preconditioned Conjugate Gradient method (PCG) and preconditioned with the Simplified Diagonal-based Incomplete Cholesky preconditioner (DIC). The velocity equations are preconditioned with the Diagonal Incomplete-LU Method (DILU) and solved with the Biconjugate Gradient Stabilised Method (BICGStab). Whereas, the stress tensor equations are precondition with the Incomplete-LU with no fill-ins Method (ILU0) and solved with the BICGStab. Also a residual control for the solvers is specified. The keyword `tolerance` specifies that the solver stops, when the residual falls below the tolerance level. Whereas the keyword `relTol` specifies the ratio of current to initial residuals [CFD Direct 2020].

TABLE 5.3: `faSolution` sub-dictionary: solver

	solver	preconditioner	tolerance	relTol
`tauSurface*`	`BiCGStab`	`ILU0`	1e-6	0

TABLE 5.4: Numerical schemes in `fvSchemes`

term	numerical scheme
`ddtSchemes`	`backward`
`gradSchemes`	`Gauss linear`
`divSchemes:`	
`div(phi,U)`	`Gauss GammaVDC 0.5`
`div(phi,tau*)`	`Gauss upwind`
`div(tau*)`	`Gauss linear`
`laplacianSchemes`	`Gauss linear corrected`
`interpolationSchemes`	`linear`
`snGradSchemes`	`corrected`

TABLE 5.5: Numerical schemes in `faSchemes`

term	numerical scheme
`ddtSchemes`	`backward`
`gradSchemes`	`Gauss linear`
`divSchemes:`	
`div(Us)`	`Gauss linear`
`div(phis,tauSurface*)`	`Gauss upwind`
`laplacianSchemes`	`none`
`interpolationSchemes`	`none`
`snGradSchemes`	`none`

5.2.4 Discretisation schemes

Tables 5.4 and 5.5 contain parameters of the `fvSchemes` and `faSchemes` dictionaries in the `systems` directory. These tables contain information about the numerical schemes for numerical terms, such as derivatives or gradients, that have to be calculated during a simulation.

5.2.5 Case set-up

To ensure a correct exchange of information between the two boundary patches, all boundary cells must be computed on the same processor. Before starting a calculation on a single processor, two commands have to be executed in the case directory. First, a command called `makeFaMesh` to calculate the FA meshes at the free surface from the two FV meshes. Second, the `setFluidIndicator` command to assign the fluid indicator to every mesh cell.

In order to run a case on multiple processors, a special work around is required. The functionality of parallel processing on multiple processors is handled by default via the public domain openMPI implementation of the standard message passing interface (MPI). Prior to starting a parallel computation the mesh and associate fields must be broken down into parts for each processor. The aim is an allocation of approximately 10 thousand cells per processor. This step of decomposition requires special attention.

For this step an OpenFOAM 5.x utility called `topoSet` is executed before the decomposition step. This utility is not part of foam-extend. Afterwards the mesh decomposition command `decomposePar` has to be executed through OpenFOAM as well. This command creates a new folder for each processor and copies partial mesh data and field data into it. However, the `decomposePar` command does not copy the foam-extend exclusive fields and therefore the `decomposePar -fields` command must be executed through a foam-extend. Afterwards, the `makeFaMesh` utility is applied each processor directory. The dictionaries for the correct decomposition of the boundary cells (`decomposeParDict` and `topoSetDict`) are located in Appendix A.

5.3 Simulations

5.3.1 Pseudo 2-Dimensional

The 2D simulations are processed locally on a CentOS7 Linux operating system. To start a simulation, the compiled solver file is executed in the case directory. The 2D simulation of this section are performed on a single processor. Thus, no decomposition procedure is required.

In order to make an initial comparison of the newly developed solver with the original, validated *bubbleInterTrackFoam*, a calculation with parameters corresponding to water is carried out first. The adaptation of the mathematical model for the viscoelastic case is based on the fact that the total stress tensor is now calculated as the sum of viscous and elastic stress tensor (see Equation 5.1). The calculation of the viscous (Newtonian) contribution was already part of the original solver, the elastic part was added. To test the new solver, the implemented Phan-Thien–Tanner model is simplified to a Newtonian model by zeroing certain model parameters.

As pointed out before, the total viscous stress tensor $\boldsymbol{\tau}$ is calculated as the sum of the viscous stress tensor and the elastic stress tensor:

$$\boldsymbol{\tau} = \boldsymbol{\tau}_s + \boldsymbol{\tau}_p \tag{5.1}$$

where the viscous, Newtonian contribution is calculated via the following equation:

$$\boldsymbol{\tau}_s = 2\mu_s \mathbf{D} \tag{5.2}$$

In Section 3.3: Constitutive equations, the Phan-Thien–Tanner model for the calculation of the elastic stress tensor was introduced. Here, the model with the exponential form of function f is used.

$$f(\mathrm{tr}(\boldsymbol{\tau}_p))\boldsymbol{\tau}_p + \lambda \overset{\triangledown}{\boldsymbol{\tau}}_p + \zeta(\mathbf{D}\cdot\boldsymbol{\tau}_p - \boldsymbol{\tau}_p \cdot \mathbf{D}^{\mathrm{T}}) = 2\mu_p \mathbf{D} \tag{5.3}$$

$$f(\mathrm{tr}(\boldsymbol{\tau}_p)) = \exp\left(\frac{\epsilon\lambda}{\mu_p}\mathrm{tr}(\boldsymbol{\tau}_p)\right) \tag{5.4}$$

TABLE 5.6: bubbleInterTrackModel water: `fvSolution` sub-dictionary: PIMPLE

	value
`pRefPoint`	`(0 0 0)`
`pRefValue`	`0`
`momentumPredictor`	`off`
`nCorrectors`	`3`
`nOuterCorrectors`	`200`
`nNonOrthogonalCorrectors`	`2`
`residualControl:`	
`U`	`tolerance:5e-5; relTol:0`
`p`	`tolerance:1e-4; relTol:0`

By setting the relaxation time, the rate of extension and the constant model constant ζ to zero ($\lambda, \epsilon, \zeta = 0$), the equation can be converted into a Newtonian form:

$$\boldsymbol{\tau}_p = 2\mu_p \mathbf{D} \tag{5.5}$$

With this simplification, a Newtonian simulation can be performed using both the viscous stress tensor calculation as well as calculation through the implemented *viscoelasticModel* library.

Solver comparison with water

The simulation of the *bubbleInterTrackModel* with parameters corresponding to water is realised by using the simplified viscoelastic model and setting the viscosities μ_s and μ_p to half the viscosity of water $\frac{1}{2}\mu_{water} = \mu_s = \mu_p$. The sum of both stress contributions should then be equal to the results of the original solver. The `viscoelasticProperties` dictionary for the simulation is found in Appendix B. A residual control setting for pressure and velocity has been added to the standard settings in the PIMPLE subdictionary (Table 5.6). The simulations are performed with a time step of 10^{-6} s. The maximal simulation time is set to 3 s.

Viscoelastic variation

The aim of this work was the simulation of a free rising single bubble in a viscoelastic fluid. Unfortunately, simulations with real viscoelastic model parameters, e.g. EPPT model parameters for a 5 g/kg Xanthan solution, crash after a very short period of time and do not provide any useful data. To test the behaviour of the viscoelastic model, very small values for the parameters initially set to zero are employed and moved to higher level of viscoelasticity (Table 5.7). These are compared with the *bubbleInterTrackModel* simulation of parameters corresponding to water from the section above. This gives the opportunity to study the behaviour of the model due to changing levels of viscoelasticity.

TABLE 5.7: EPPT model parameter: viscoelastic variation

	water	least viscoelastic	medium viscoelastic	most viscoelastic
ρ	1011	1011	1011	1011
μ_s	41.65e-5	41.65e-5	41.65e-5	41.65e-5
μ_p	41.65e-5	41.65e-5	41.65e-5	41.65e-5
λ	0	10e-6	10e-5	10e-4
ϵ	0	10e-6	10e-5	10e-5
ζ	0	10e-6	10e-5	10e-4

5.3.2 3-Dimensional

Numerical parameter influence

Before the newly developed solver is tested on a 3D geometry, simulations are performed with the original *bubbleInterTrackFoam* and the meshes created in Section 5.2.1. The computations are performed on high performance computers of the Nordeutscher Verbund für Hoch- und Höchstleistungsrechnen (HLRN) at the location Göttingen. The simulations are performed on 40 and 120 cores yielding approximately 10,000 cells per core. In order to get the simulation running, numerical parameter shown in Table 5.3.2 are varied. Among the varied parameter are PIMPLE parameters discussed in Section 2.4.2 as well as parameters adjusting the MRF (`lambdaFf` and `lambdaF0`). Furthermore, an initialisation of the vertical velocity with the expected value obtained by the 2D simulation is made as well as a simulation with a decrease of the time step.

TABLE 5.8: *bubbleInterTrackFoam* 3D simulations: numerical configurations

Simulation ID	nCorrectors	nOuter Correctos	nonOrthogonal Correctors	`lambdaFf`	`lambdaF0`	velocity initialisation	time step
standard	6	2	1	1	1	(0 -1e-6 0)	10e-6
3nonOrtho	6	2	3	1	1	(0 -1e-6 0)	10e-6
3nonOrtho_1000nOuter	3	1000	3	1	1	(0 -1e-6 0)	10e-6
3nonOrtho_1000nOuter_mrf0.5	3	1000	3	0.5	0.5	(0 -1e-6 0)	10e-6
3nonOrtho_200nOuter	3	200	3	1	1	(0 -1e-6 0)	10e-6
50nCorr	50	2	1	1	1	(0 -1e-6 0)	10e-6
initU	3	200	2	1	1	(0 0.014 0)	10e-6
mrf_0.1	3	200	2	0.1	0.1	(0 -1e-6 0)	10e-6
mrf_0.5	3	200	2	0.5	0.5	(0 -1e-6 0)	10e-6
mrf_0.7	3	200	2	0.7	0.7	(0 -1e-6 0)	10e-6
dt_10e-7	3	200	2	1	1	(0 -1e-6 0)	10e-7

Chapter 6

Results and Discussion

This sections contains the results and discussion of the simulations performed with parameters presented in the previous chapter. The first section treats the comparison of the newly developed solver with the original solver using parameters corresponding to water. The second section discusses the behaviour of the implemented viscoelastic model to changes in the viscoelastic model parameters. Finally, the 3D simulations performed with the original solver are discussed and the influence of different numerical parameters evaluated.

6.1 Pseudo 2D: Solver comparison with parameters corresponding to water

6.1.1 Mean vertical bubble velocity

Figure 6.1 shows the mean vertical bubble velocity over time with parameters corresponding to water. The course of the original *bubbleInterTrackFoam* solver is presented in a solid blue line, while the new *bubbleInterTrackModel* solver is presented in a dotdashed orange line.

The first 0.05s, both vertical velocities follow the same path with a short dip at the beginning and then followed by a steep increase in velocity. The velocity for the original solver rises until ≈0.1 s and then reaches a quasi-stationary value of ≈0.014 m/s. The velocity oscillates slightly with a rising amplitude. At ≈0.45 s an unsteady section with velocity drops into negative values is reached. The simulation crashed after ≈1.1 s.

While the velocity curve for the modified solver initially shows a similar behaviour, it stagnates to a lower terminal rise velocity. An oscillating section with a nearly constant slope is reached between 0.15 s and 0.25 s with a maximal velocity of ≈0.012 m/s. After this section, the mean velocity drops to a constant value of ≈0.010 m/s. At ≈0.6 s, a similar unstable section is reached as it was observed by the *bubbleInterTrackFoam*-curve. The simulation crashed after ≈1.3 s.

An explanation for the initial drop in both curves is given by the initial velocity condition in y-direction as presented in Table 5.1. The simulation time until approx. 0.15 s is considered as the running-in period. As no physically correct initial condition can be given for the bubble rise, this section does not represent actual bubble behaviour. Both unstable regions also do not correspond to any real bubble behaviour, even though numerical solutions have been found here.

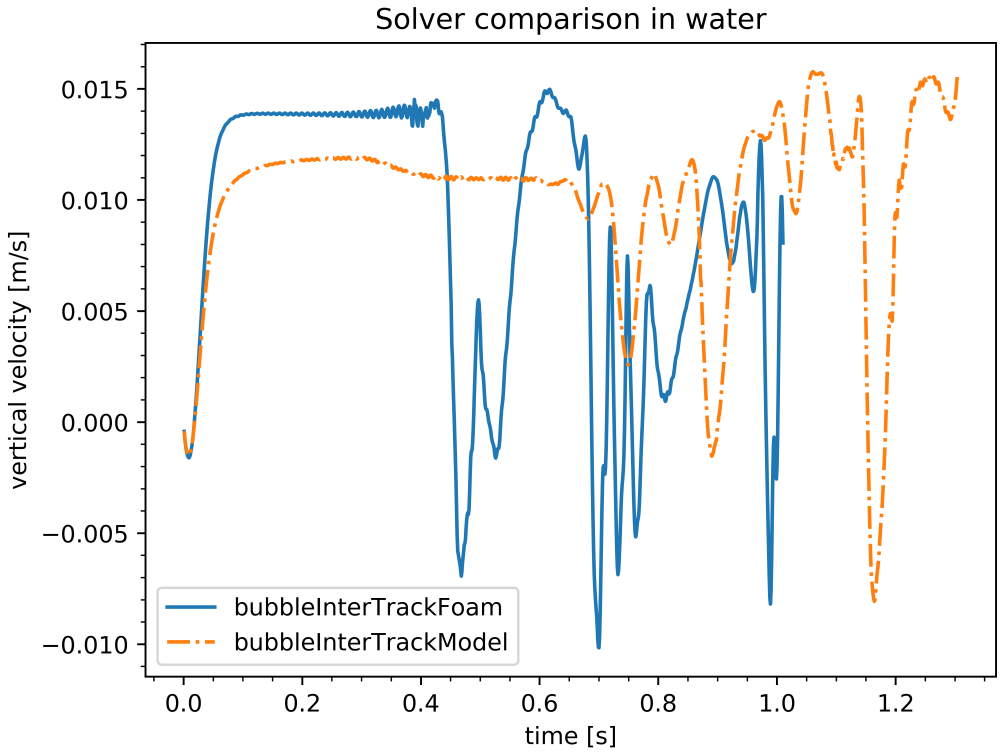


FIGURE 6.1: Solver comparison in water: mean vertical bubble velocity over time

The oscillation of the vertical bubble velocity of the *bubbleInterTrackFoam*-curve increases constantly during the course up to 0.38 s. Between 0.38 s and 0.4 s, a section of still small but irregular oscillation occurs. Shortly after that the curve enters the totally unstable section. The period of steady oscillation is regarded as quasi-stationary. Entering the following unstable section can be caused by a high build-up of the speed and associated errors in the calculation.

The curve of the *bubbleInterTrackModel* solver reaches an average lower vertical velocity than the original solver after reaching the quasi-stationary section between 0.15 s and 0.3 s. This indicates that the calculation of viscosity does not work as intended. The bubble faces a higher flow resistance. The curve shows oscillation but the increase of the amplitude is much smaller compared to the *bubbleInterTrackFoam*-curve. It can be observed that the velocity drops to a lower level around the same time the original solver leaves the quasi-stationary section (0.3 s – 0.45 s). The decrease in velocity level indicates that the fluid dynamics of the bubble are changing at this point, resulting in a greater loss of energy in the flow. An evaluation of the pressure, velocity and stress fields in this area should provide useful information about the processes happening in this section. Unfortunately, the data was not available at the time of the evaluation and to the best of my knowledge I could not find an explanation for the behaviour.

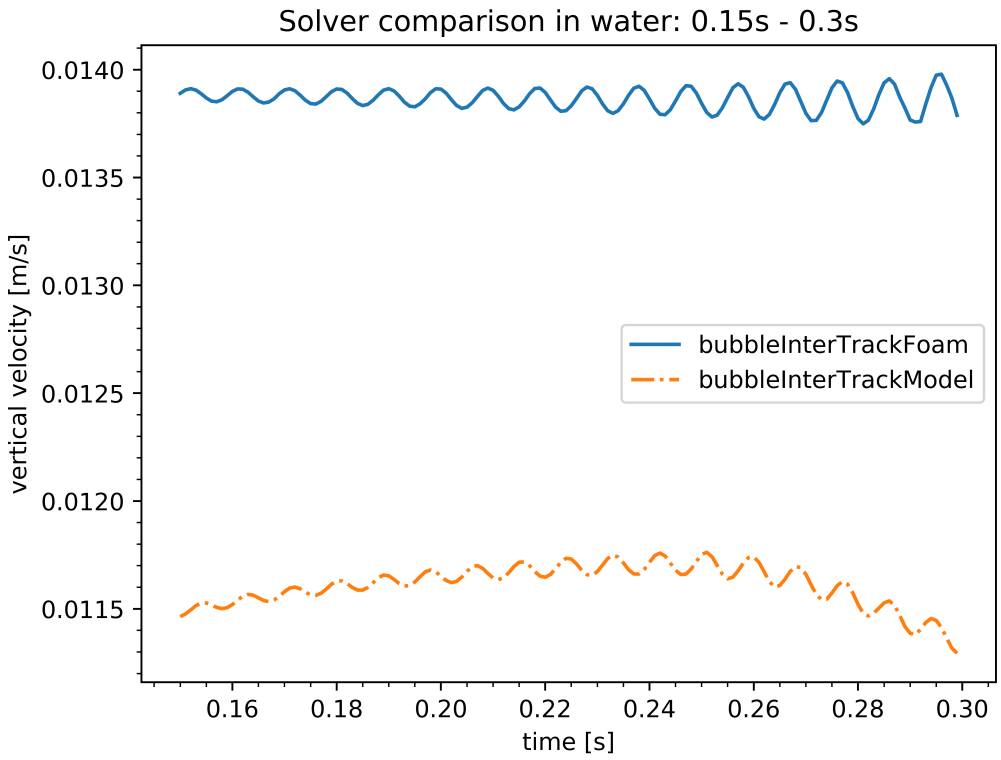


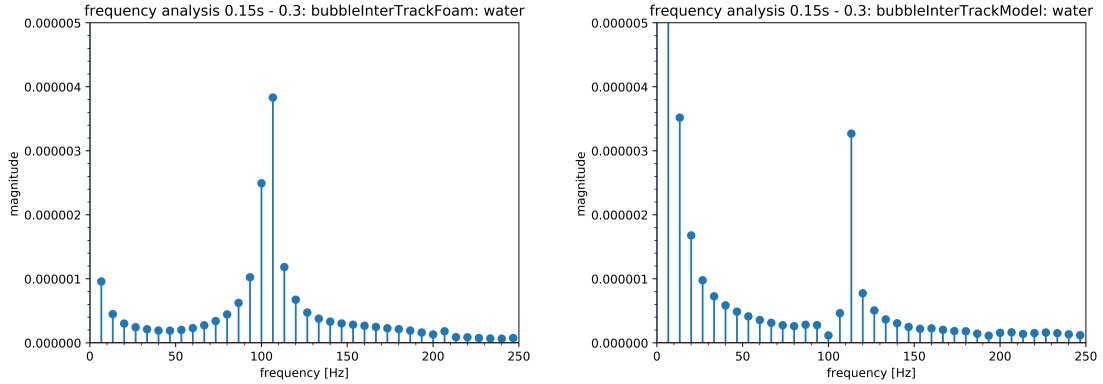
FIGURE 6.2: Solver comparison in water: mean bubble velocity over time 0.15s – 0.3s

6.1.2 Frequency analysis

In both cases the region before 0.15 s and after 0.3 s is considered as not stationary and thus not further evaluated, as the results do not reflect natural behaviour. Figure 6.2 shows the quasi-stationary section of Figure 6.1 for a more detailed view. It can be seen, that, in addition to the lower velocity level, the amplitude of the *bubbleInterTrackModel* curve oscillation is smaller than of the original solver. For a more detailed investigation of the oscillation, a frequency analysis of the two sections is presented in Figure 6.2. The frequency analysis is carried out using a fast Fourier Transformation (fft) in Python. The results are shown in Figure 6.3. The new developed solver (Figure 6.3b) showed a slightly higher main oscillation frequency of about 112 Hz compared to frequencies of 100 Hz and 108 Hz at the two main peaks in the left Figure 6.3a of the original solver. Higher oscillation frequencies indicate a lower flow resistances. However, this is complementary to the observation of a lower mean vertical velocity of the *bubbleInterTrackModel* solver, which was an indication of higher flow resistance. As mentioned above, a different bubble shape and associated fluid dynamic resistance can impact on the bubble behaviour.

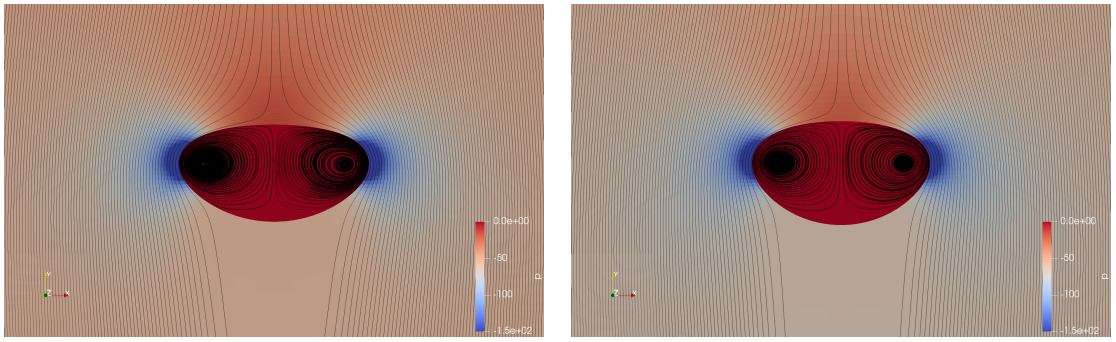
6.1.3 Visual bubble shape comparison

A visual comparison of the bubble shapes through a pressure plot is done after a simulation time of 0.2 s (Figure 6.4). The visual oscillation of both bubbles is without animation



(A) bubbleInterTrackFoam (B) bubbleInterTrackModel

FIGURE 6.3: Frequency analysis: solver comparison in water 0.15s – 0.3s



(A) bubbleInterTrackFoam (B) bubbleInterTrackModel

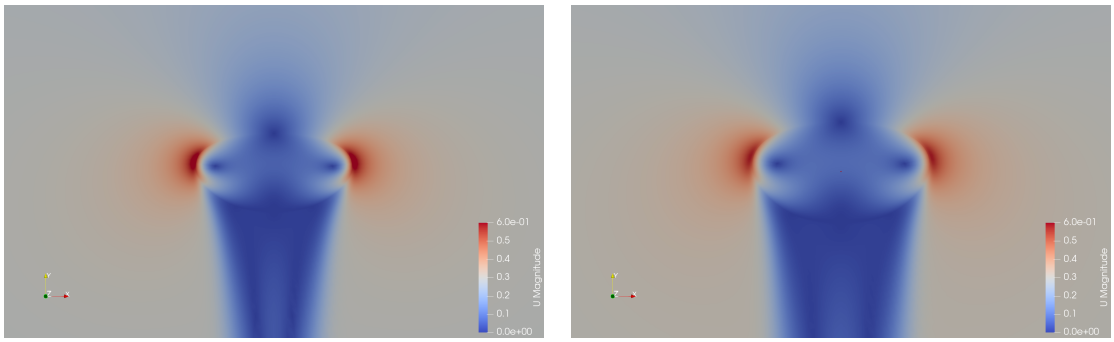
FIGURE 6.4: Bubble pressure plot with velocity steam lines after 0.2s with parameters corresponding to water: solver comparison

barely noticeable and can be neglected for this comparison. While the basic bubble shape as well as areas of higher and lower pressure are similar, the exact bubble shapes differ. The bubble calculated with new solver shows a bit more compact shape, while the bubble calculated with the original solver is flatter horizontally. The different bubble shape indicates again that the simulation with the new solver does not work as intended, because otherwise similar results were expected.

6.1.4 Pressure and velocity field comparison

A comparison of the pressure plots in the quasi-stationary section at 0.2 s shows areas of high and low pressure at the same location in both simulations. The simulation with the original solver shows higher pressure of the surrounding fluid. Especially the area above the bubble shows higher pressure values (Figure 6.4). This is probably due to the higher mean vertical velocity of the original solver (Figure 6.1).

The velocity fields at the same moment show a similar picture: Areas of high and low velocity are at the same locations in both simulations. These areas are more pronounced in the simulation with the original solver due to higher mean vertical velocity (Figure 6.5).



(A) bubbleInterTrackFoam (B) bubbleInterTrackModel

FIGURE 6.5: Bubble velocity field after 0.2s with parameters corresponding to water: solver comparison

An explanation for the different results with the newly developed solver cannot be found.

6.2 Pseudo 2D: Viscoelastic variation

6.2.1 Mean Vertical bubble velocity

Figure 6.6 shows the mean vertical bubble velocity over time of the simulation with the varied viscoelastic model parameters. In a blue solid line the simulation with *water* without viscoelasticity is shown. The orange dotdashed line shows the simulation with very small viscoelastic parameters (*least viscoelastic*). The green dotted line shows more viscoelastic parameters (*medium viscoelastic*) and the red dashed line shows the *most viscoelastic* parameters. The *least viscoelastic* and *medium viscoelastic* curves show courses similar to the *water* simulation. The curves of the least viscoelastic curve and the medium viscoelastic curve are both lying above the curve with water but show a very similar course. The most viscoelastic simulation initially follows this pattern and remains in the running-in section above the medium viscoelastic curve. Until about 0.05 s the curve bends into a constant section with slight oscillation before reaching an unstable range at about 0.13 s. The increase in bubble velocity can be explained by considering the constitutive model.

Inserting model parameters >0 increases the left side of Equation 3.10, which leads to a shear-thinning effect. This pattern is initially followed by all simulations. The *most viscoelastic* simulation then leaves this pattern into a constant region shortly before diverging. It is already known from Section 5.3.1 that the new *bubbleInterTrackModel* solver does not provide physical correct solutions. In order to compare the velocity oscillation, the *most viscoelastic* simulation is not taken into account.

6.2.2 Quasi-stationary section

Figure 6.7 shows the bubble velocity curves for *water*, *least viscoelastic* and *medium viscoelastic* in a range from 0.15 s to 0.3 s. A visual comparison shows very similar frequencies in all three simulations. The *least viscoelastic* curve follows almost the same path as the

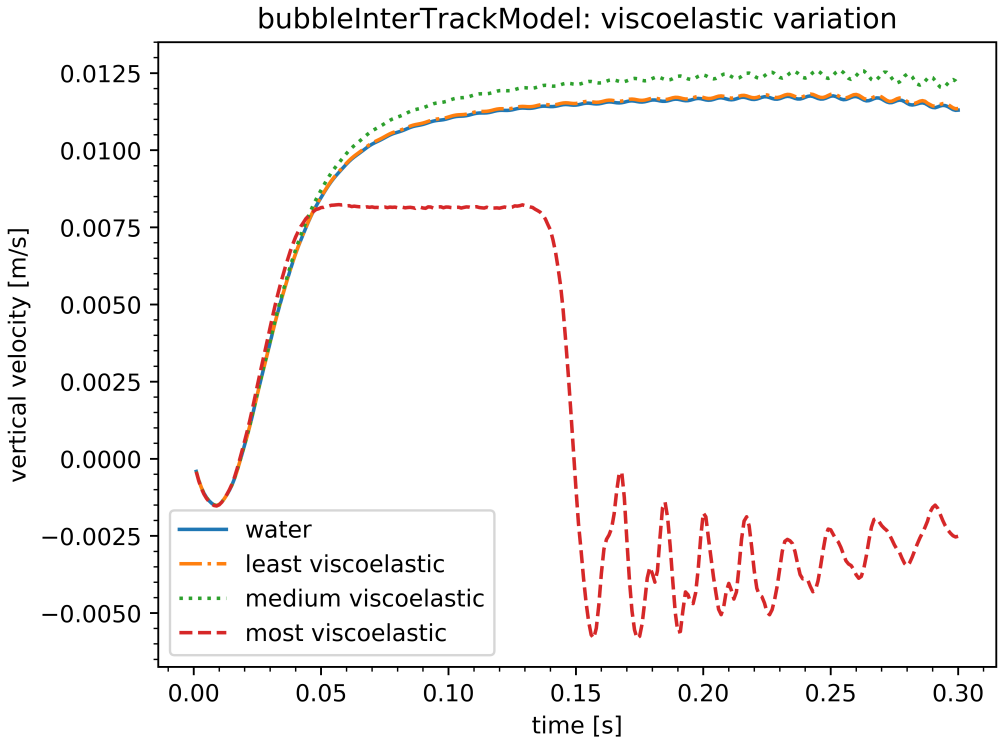


FIGURE 6.6: Viscoelastic variation: mean vertical bubble velocity

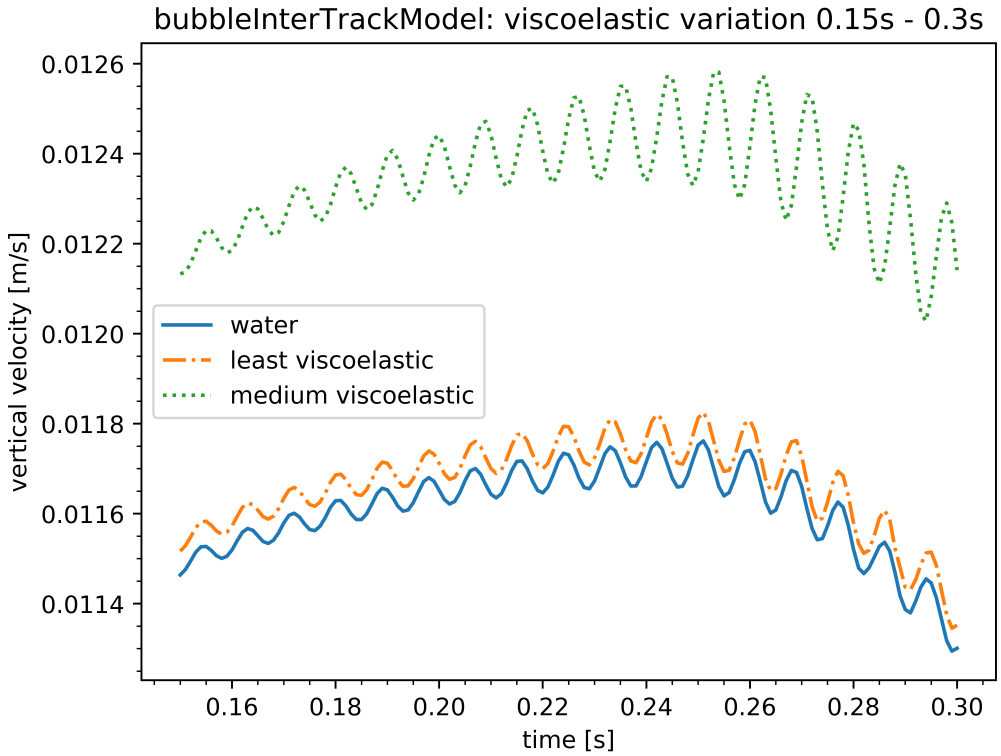


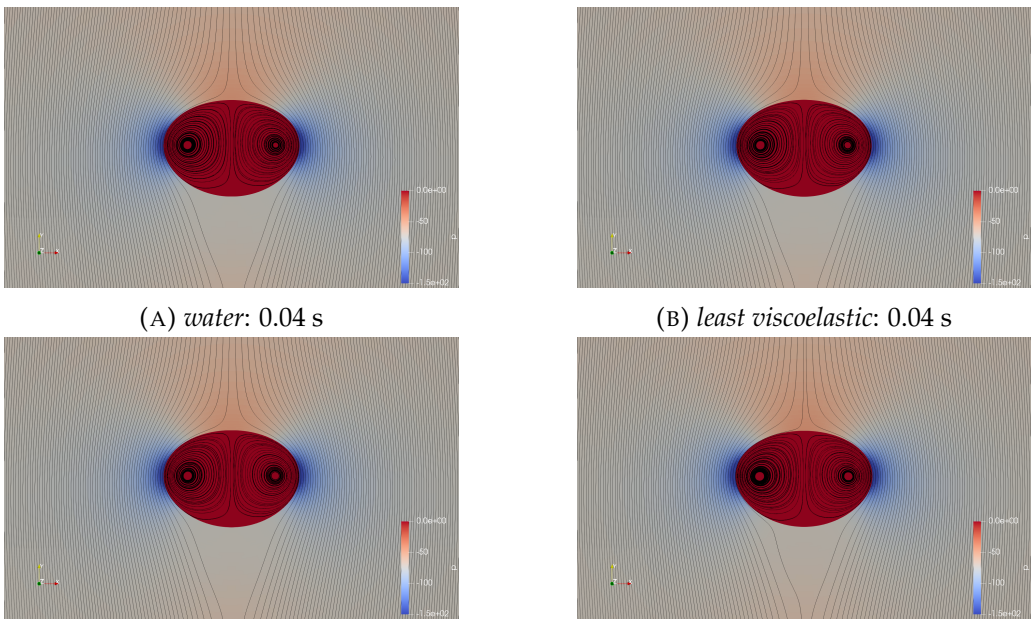
FIGURE 6.7: Viscoelastic variation: mean vertical bubble velocity 0.15s – 0.3s

water curve with a little off-set to higher velocities. The *medium viscoelastic* curve however, shows a high increase in the bubble amplitude. This higher amplitude can be explained through lower fluid viscosity and a higher velocity. It can be concluded that the implemented model responds as expected to changes in the viscoelastic modelling parameters. High levels of viscoelasticity cause numerical errors and unexpected behaviour.

6.2.3 Pressure fields

To identify differences in the simulations, the pressure fields are compared. Three characteristic moments in the course of the simulation are considered. First, after 0.04 s before the *most viscoelastic* curve changes it slope abruptly (Figure 6.11). After 0.06 s, which is the moment after the change of the slope (Figure 6.12). The last figure at 0.114 s shows the velocity fields at the moment before the *most viscoelastic* mean vertical velocity curve drops (Figure 6.13).

Before the *most viscoelastic* curve shows larger differences in the plot of the mean vertical velocity over time (Figure 6.6), the pressure fields look similar in all simulations at 0.04 s (Figure 6.8). The change in mean vertical velocity is not reflected in a change of the pressure field at 0.06 s. The pressure still shows an similar distribution in all simulations (Figure 6.9). The pressure plots after 0.114 s show a different picture (Figure 6.10). The simulations of *water*, *least viscoelastic* and *medium viscoelastic* are still similar and changed according to the time passed. However, the calculation of the pressure field of the *most viscoelastic* simulation after 0.114 s produced wrong results. Half of the surrounding fluid has a high and the other half a low pressure. Also, a checkerboard pattern is visible in the upper left and lower right corners of Figure 6.10d. All this indicates that the solution of the pressure equations has failed.



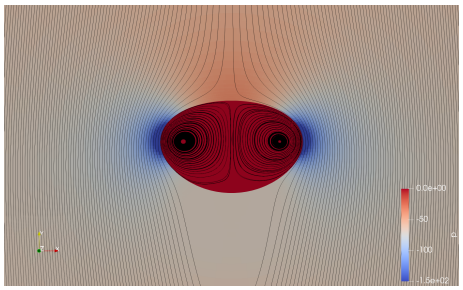
(A) *water*: 0.04 s

(B) *least viscoelastic*: 0.04 s

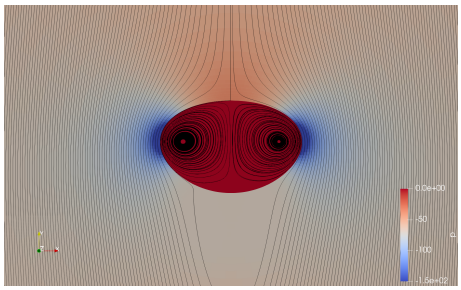
(C) *medium viscoelastic*: 0.04 s

(D) *most viscoelastic*: 0.04 s

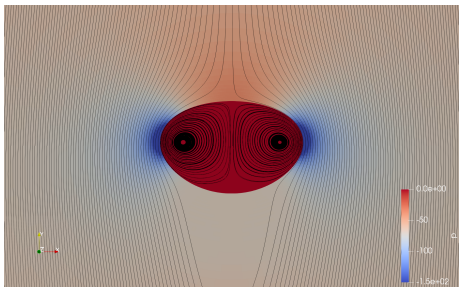
FIGURE 6.8: Viscoelastic variation: bubble pressure fields at 0.04 s



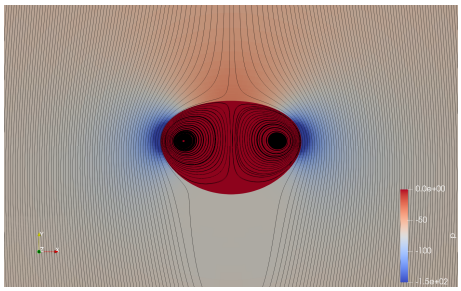
(A) *water*: 0.06 s



(B) *least viscoelastic*: 0.06 s

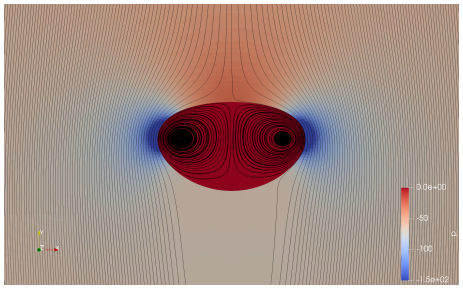


(C) *medium viscoelastic*: 0.06 s

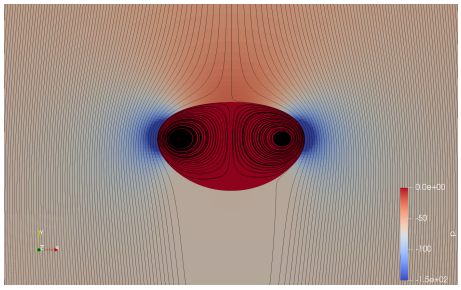


(D) *most viscoelastic*: 0.06 s

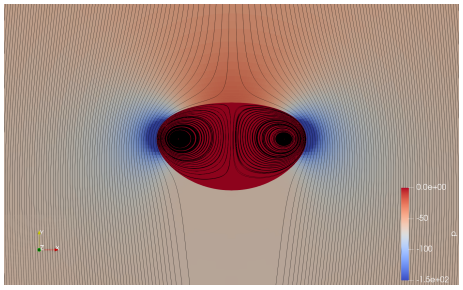
FIGURE 6.9: Viscoelastic variation: bubble pressure fields at 0.06 s



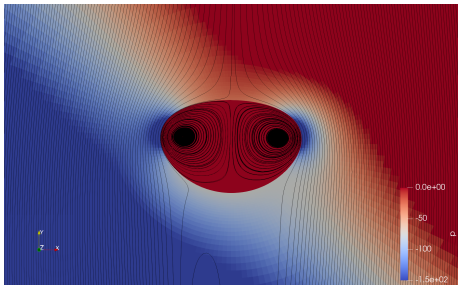
(A) *water*: 0.114 s



(B) *least viscoelastic*: 0.114 s



(C) *medium viscoelastic*: 0.114 s



(D) *most viscoelastic*: 0.114 s

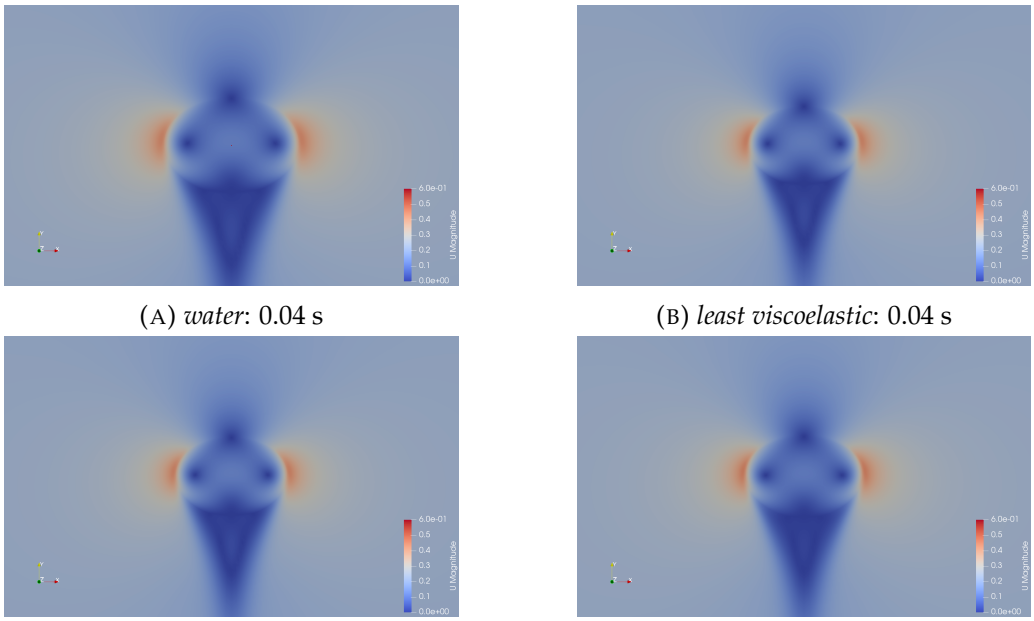
FIGURE 6.10: Viscoelastic variation: bubble pressure fields at 0.114 s

6.2.4 Velocity fields

The velocity fields of all four simulations after 0.04 s look similar. While the plot of the mean vertical velocity over time (Figure 6.6) shows a change of the slope of the *most viscoelastic* simulation between 0.04 s and 0.06 s, this is not reflected in the velocity field (Figure 6.11 and Figure 6.12). This is a similar results to the comparison of the pressure fields. No visible change of the pressure field is noticeable at this characteristic point.

Comparing the velocity fields after 0.06 s and 0.114 s (Figure 6.12 and Figure 6.13),

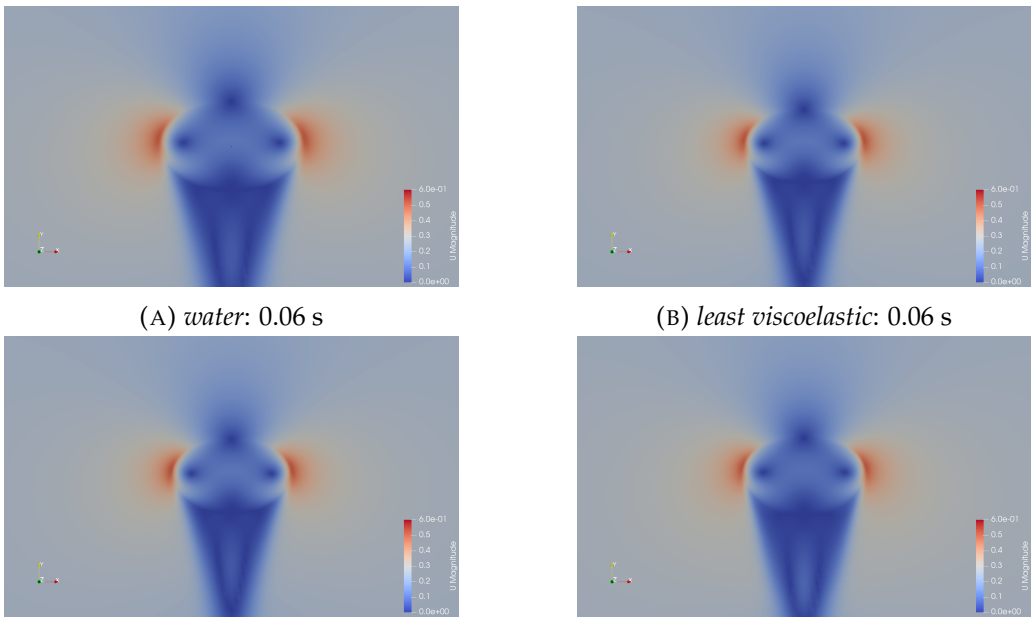
one can see that areas of low velocity below the bubble have become wider in all simulations. For the simulations *water*, *least viscoelastic* and *medium viscoelastic* this is due to the increase of the bubble velocity. This widening of the low velocity area is also observed in the *most viscoelastic* simulation. However, between 0.06 s and 0.114 s the mean vertical bubble velocity is constant and no change in the velocity field is expected. The abnormal behaviour of the pressure field at 0.014 s (Figure 6.10d) does not show changes in velocity fields.



(A) *water*: 0.04 s (B) *least viscoelastic*: 0.04 s

(C) *medium viscoelastic*: 0.04 s (D) *most viscoelastic*: 0.04 s

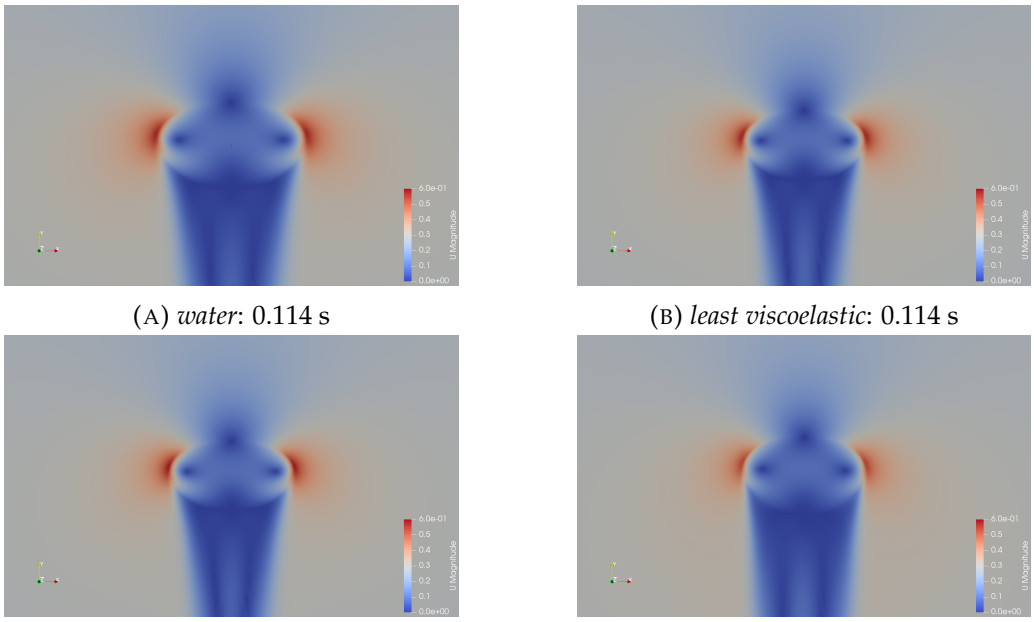
FIGURE 6.11: Viscoelastic variation: bubble velocity fields at 0.04 s



(A) *water*: 0.06 s (B) *least viscoelastic*: 0.06 s

(C) *medium viscoelastic*: 0.06 s (D) *most viscoelastic*: 0.06 s

FIGURE 6.12: Viscoelastic variation: bubble velocity fields at 0.06 s



(A) *water*: 0.114 s

(B) *least viscoelastic*: 0.114 s

(C) *medium viscoelastic*: 0.114 s

(D) *most viscoelastic*: 0.114 s

FIGURE 6.13: Viscoelastic variation: bubble velocity fields at 0.114 s

6.2.5 Stress fields

As already mentioned in Section 4.5.5, a mistake was overlooked during the development process of the methods that calculates the total stress field (Code Sample 4.20). Unfortunately, the calculated output stress field only contains the viscous stress contribution making an evaluation with regards to the viscoelastic stress tensor impossible. This error should be fixed immediately to enable an evaluation of stress fields for further research.

6.2.6 Further observations

Several simulations displayed the same `corrupted size vs. prev_size error`, this did not lead to a crash of the simulation and was displayed at the end of the simulation. The error may occur if modified libraries with modified function lead to multiple instances of functions with the same name in the memory. The destruction of these functions causes the corrupted memory error [CFD Online 2020]. As in the process of the viscoelastic implementation several existing libraries were modified. If the destruction of the objects are the root of this error, it would explain why the error shows up after finishing the simulation.

Furthermore, it could be seen from the log-files during the simulation that for the convergence of the pressure equations about 10 times more iterations are required than for the convergence of the velocity equations. The differential equations for the calculation of the viscoelastic model, always converged quick. Difficulties in the convergence of the pressure equations may be due to the implicit procedure but might also mean that further research is necessary to address the problem.

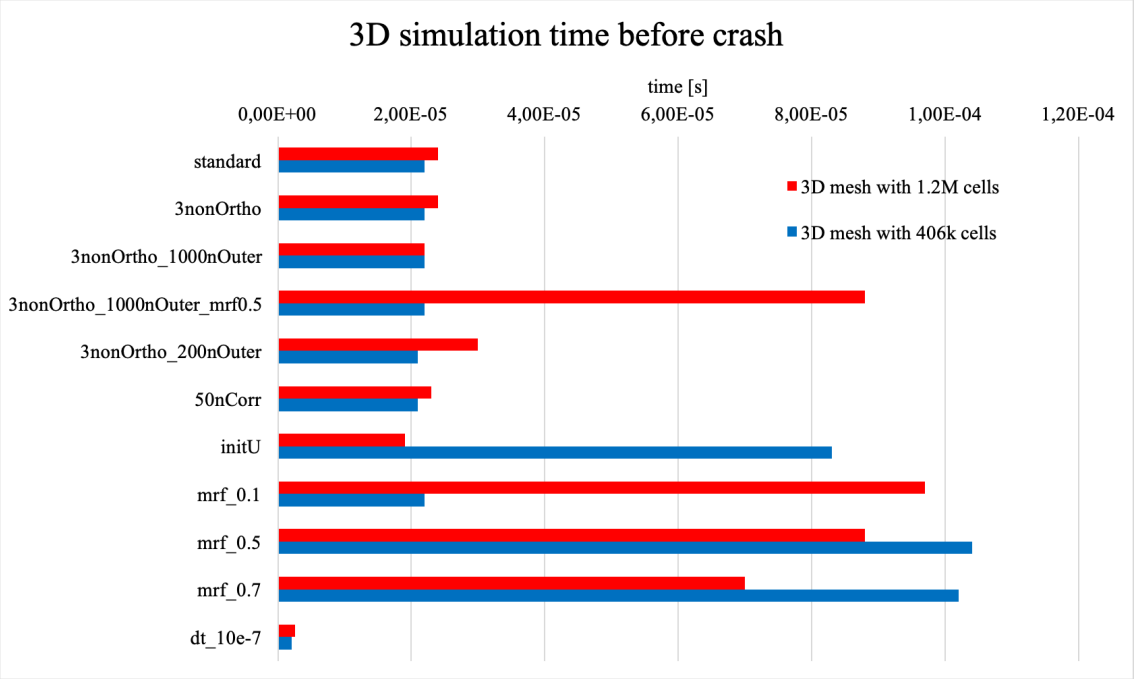


FIGURE 6.14: *bubbleInterTrackFoam* 3D Simulation times with varying parameters

6.3 3D: Influence of numerical parameter on the convergence

Figure 6.14 shows run-times of the simulations of the two 3D meshes with different numerical configurations. At first glance it can be seen that simulations crashed very early in the process and none produced any expected results. The results show that changes of the PIMPLE parameters does not influence the simulation run-times a lot. Both inner and outer iterations do not show any change of the run-time. Also, the addition of extra non-orthogonal Correctors (both meshes are highly non-orthogonal) does not show big effects. Although the finer mesh produces slightly longer run-times in the standard settings, this is not confirmed during the initialisation of the velocity field. In comparison, the coarser mesh shows significantly higher run-times, while the finer mesh even shows a small decrease compared to the standard case. The biggest positive effect on the run-times is the alteration of the MRF properties with an optimum value of 0.5. Contrary to expectations, a reduction of the time step does not show longer run-times but the lowest values overall.

All in all, it is not possible to make any real statement about the effect of the numerical parameters on the simulation time from the results. Although the change in MRF properties appears to have a positive effect on the run-times, the general crashes after such short periods of time cannot be explained. Since 3D simulations were already performed with the original solver, the cause of the problem may be found in the mesh.

Unfortunately, due to time limitations, no further experiments were possible here. Nevertheless, some assumptions can be made about the cause of the problems: The first source of the problems could be found in the creation of the geometry. Possibly, the

proportions of the geometry are not correct and could cause the observed quick crashes. Another source of error could be the generation of the mesh itself. The meshes computed in the literature (e.g. [Pesci et al. 2018; Charin et al. 2017]) are in the same range of the number of cells as the ones used here, but clearly more structured. If it is not possible yet to run a simulation with the original solver, the newly developed solver itself should not be run on the 3D case.

Chapter 7

Conclusion and Outlook

The first part of this work consists of the modification of the mathematical model forming the foundation of the viscoelastic implementation. The key steps during the surface-tracking procedure of the *bubbleInterTrackFoam* solver are the update-methods for the boundary conditions at the interface. These methods were derived from force balances at the interface, which are reworked for viscoelastic fluid behaviour in this work. The calculation of the extra elastic stress tensor was performed with the Phan-Thien–Tanner constitutive model in its exponential form. The *viscoelasticModel* library of foam-extend-4.0 is implemented in the solver structure and modified to solve the constitutive equation on a Finite Area mesh at the interface. This modification resulted in further changes in the *finiteArea* library to enable finite area operations with the newly introduced data structures, which are used to solve the constitutive equation at the interface.

To compare the newly developed *bubbleInterTrackModel* solver with the original solver, a simplification of the constitutive model is used first. With this simplification, certain model parameters are set to zero, which allows the simulation of Newtonian fluid behaviour. The simulations are carried out with a pseudo 2D mesh and parameters corresponding to water. Unfortunately, a comparison of the mean vertical bubble velocity over time reveals that the newly developed solver does not deliver the same results as the original solver. The reached mean vertical bubble velocity remains below the velocity of the original solver and the course of the curve differs. This suggests that the new solver calculates a flow resistance, which is too high. A frequency analysis quasi-stationary section of the bubble rise underlines that the implementation has failed and the new solver cannot reproduce a simulation with parameters corresponding to water. Pressure and velocity plots of the both bubbles during the quasi-stationary section at 0.2 s reveal different bubble shapes but same location of high and low values. The original solver shows more pronounced plots due to the higher mean vertical velocity of the simulation.

In a second set of pseudo 2D simulations, small values for the parameters set to zero are employed and slowly moved to higher levels of viscosity. The model reacts to changes in the viscoelastic model parameters while the parameters are small. However, if the viscoelasticity values are too high, the model does not react as expected. Possible sources of error can be a `corrupted size vs. prev_size error`, as well as the difficult convergence of the pressure equations. Further improvements to the solution procedure could be made by implementing the Convergent and Universally Bounded Interpolation Scheme for the Treatment of Advection (CUBISTA), which is a highly stable discretisation scheme for advection term in the constitutive equation [Habla et al. 2014; de Paulo et

al. 2007]. Another point of improvements can be simulations with a `momentumPredictor on` compared to the `off` configuration in the PIMPLE sub-dictionary, which could lead to higher accuracy in solving the pressure equations. Using a finer pseudo 2D mesh should also lead to solution with a higher accuracy. Probably the most promising improvement would be an investigation about appropriate boundary conditions of the stress tensor at the interface. The boundary conditions used in this work are chosen arbitrary, because to the best of my knowledge only minimal literature has been published on this specific topic. An advanced proposal is the derivation of a new boundary condition at the interface for each viscoelastic model individually.

Finally, 3D Simulations were carried out using the original solver and two different meshes. The simulation with the standard numerical parameter crashed unexpectedly quick. A variation of the numerical parameter did not yield in simulations with acceptable run-times. The biggest positive effect on the run-times showed the alteration of the MRF properties with an optimum value of 0.5. A possible source of error for the overall short run-times until the simulations crashed, is most likely the geometry or the mesh generation. An initial objective of this work was the 3D simulation of a viscoelastic fluid with the new developed solver. As a successful simulation with the original solver was not possible yet, it was refrained to perform simulation with the new solver with 3D meshes.

Utilising the presented modified mathematical model and the given recommendations to improvement of the code, a simulation of a free rising bubble in a viscoelastic fluid with the surface-tracking method may be possible throughout a further research study.

Bibliography

Barriga, Elias H. and Roberto Mayor (2019). "Adjustable viscoelasticity allows for efficient collective cell migration". In: *Seminars in Cell and Developmental Biology* 93, pp. 55–68. URL: https://doi.org/10.1016/j.semcdb.2018.05.027.

Batchelor, G. K. (2000). *An Introduction to Fluid Dynamics*. Cambridge University Press. URL: https://www.cambridge.org/core/product/identifier/9780511800955/type/book.

Caboussat, Alexandre (2005). "Numerical simulation of two-phase free surface flows". In: *Archives of Computational Methods in Engineering* 12.2, pp. 165–224.

Cardoso, F. A.R. et al. (2018). "CFD-based modeling of precipitation by supercritical antisolvent process of microparticles from grape pomace extract with population balance approach". In: *Journal of Supercritical Fluids* 133, pp. 519–527. URL: http://dx.doi.org/10.1016/j.supflu.2017.10.027.

CFD Direct (2020). *OpenFOAM v6 User Guide: 4.5 Solution and algorithm control (date accessed: 13.02.2020)*. URL: https://cfd.direct/openfoam/user-guide/v6-fvsolution/.

CFD Online (2020). *CFD Online (date accessed: 29.03.2020)*. URL: https://www.cfd-online.com/Forums/openfoam-programming-development/211597-corrupted-size-vs-prev_size-error-foam-extend4-0-a.html.

Charin, A. H.L.M. et al. (2017). "A moving mesh interface tracking method for simulation of liquid–liquid systems". In: *Journal of Computational Physics* 334, pp. 419–441.

Chen, K. P., W. Saric, and H. A. Stone (2000). "On the deviatoric normal stress on a slip surface". In: *Physics of Fluids* 12.12, pp. 3280–3281.

Chhabra, R P and J F Richardson (1999). "Non-Newtonian flow in the process industries : fundamentals and engineering applications". In: *Non-Newtonian Flow in the Process Industries, Butterworth-Heinemann*, pp. 73–161.

de Paulo et al. (2007). "A marker-and-cell approach to viscoelastic free surface flows using the PTT model". In: *Journal of Non-Newtonian Fluid Mechanics* 147.3, pp. 149–174.

Demirdzic, I. and M. Peric (1988). "Space Conservation Law in Finite Volume Calculations of Fluid Flow". In: *International Journal for Numerical Methods in Fluids* 8.December 1987, pp. 1037–1050.

Favero, J. L. et al. (2010). "Viscoelastic fluid analysis in internal and in free surface flows using the software OpenFOAM". In: *Computers and Chemical Engineering* 34.12, pp. 1984–1993. URL: http://dx.doi.org/10.1016/j.compchemeng.2010.07.010.

Gopala, Vinay R. and Berend G.M. van Wachem (2008). "Volume of fluid methods for immiscible-fluid and free-surface flows". In: *Chemical Engineering Journal* 141.1-3, pp. 204–221.

Habla, Florian et al. (2014). "Numerical simulation of the viscoelastic flow in a three-dimensional lid-driven cavity using the log-conformation reformulation in OpenFOAM®". In: *Journal of Non-Newtonian Fluid Mechanics* 212, pp. 47–62. URL: http://dx.doi.org/10.1016/j.jnnfm.2014.08.005.

Issa, R. I. (1986). "Solution of the implicitly discretised fluid flow equations by operator-splitting". In: *Journal of Computational Physics* 62.1, pp. 40–65.

Jasak, H. (2005). *http://powerlab.fsb.hr/ped/kturbo/OpenFOAM/slides/UniDarmstadt_11Jan2005.pdf (2020-01-09)*.

Jasak, H and Ž Tuković (2006). "Automatic mesh motion for the unstructured Finite Volume Method". In: *Transactions of Famena* 30.2, pp. 1–20.

Kassiotis, C (2008). "Which strategy to move the mesh in the Computational Fluid Dynamic code OpenFOAM". In: *Elements*, pp. 1–14.

Morrison, F.A. (1998). *Understandig Rheology*. Raymond F. Boyer Library Collection. Oxford University Press. URL: https://books.google.de/books?id=bwTn8ZbR0C4C.

Muzaferija, S and M. Peric (1997). "COMPUTATION OF FREE-SURFACE FLOWS USING THE FINITE-VOLUME METHOD AND MOVING GRIDS". In: *Numerical Heat Transfer, Part B: Fundamentals* 32.4, pp. 369–384. URL: http://www.tandfonline.com/doi/abs/10.1080/10407799708915014.

Nikrityuk, Petr a (2011). *Computational Thermo-Fluid Dynamics*. Weinheim, Germany: Wiley-VCH Verlag GmbH & Co. KGaA. URL: http://doi.wiley.com/10.1002/9783527636075.

Oliveira, Paulo J. and Fernando T. Pinho (1999). "Analytical solution for fully developed channel and pipe flow of Phan-Thien-Tanner fluids". In: *Journal of Fluid Mechanics* 387, pp. 271–280.

OpenFOAM® (2020a). *Official home of The Open Source Computational Fluid Dynamics (CFD) Toolbox (date accessed: 13.01.2020)*. URL: https://www.openfoam.com/.

— (2020b). *OpenFOAM: User Guide: Empty (date accessed: 12.02.2020)*. URL: https://www.openfoam.com/documentation/guides/latest/doc/guide-bcs-constraint-empty.html.

OpenFOAMWiki (2020a). *Contrib interTrackFoam (date accessed: 10.01.2020)*. URL: https://openfoamwiki.net/index.php/Contrib_interTrackFoam.

— (2020b). *OpenFOAM guide/The PIMPLE algorithm in OpenFOAM (date accessed: 07.02.2020)*. URL: https://openfoamwiki.net/index.php/OpenFOAM_guide/The_PIMPLE_algorithm_in_OpenFOAM.

— (2020c). *OpenFOAM guide/The SIMPLE algorithm in OpenFOAM (date accessed: 07.02.2020)*. URL: https://openfoamwiki.net/index.php/OpenFOAM_guide/The_SIMPLE_algorithm_in_OpenFOAM.

Pesci, Chiara et al. (2018). "Computational analysis of single rising bubbles influenced by soluble surfactant". In: *Journal of Fluid Mechanics* 856, pp. 709–763.

Phan-Thien, N. (1978). "A Nonlinear Network Viscoelastic Model". In: *Journal of Rheology* 22.3, pp. 259–283.

Phan-Thien, Nhan and Roger I. Tanner (1977). "A new constitutive equation derived from network theory". In: *Journal of Non-Newtonian Fluid Mechanics* 2.4, pp. 353–365.

Quan, Shaoping and David P. Schmidt (2007). "A moving mesh interface tracking method for 3D incompressible two-phase flows". In: *Journal of Computational Physics* 221.2, pp. 761–780.

Rusche, Henrik (2002). "Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions". PhD thesis. University of London.

Sadrehaghighi, Ideen (2020). *Special Topics in CFD*.

Schwarze, Rüdiger (2013). *CFD-Modellierung*. Vol. 1. 4. Berlin, Heidelberg: Springer Berlin Heidelberg, p. 53. URL: http://link.springer.com/10.1007/978-3-642-24378-3.

Tuković, Ž (2005). "Metoda kontrolnih volumena na domenama promjenjivog oblika". In:

Tuković, Ž and H Jasak (2008). "Simulation of Free-Rising Bubble With Soluble Surfactant Using Moving Mesh Finite Volume / Area Method". In: *6th International Conference on CFD in Oil & Gas, Metallurgical and Process Industries* June, pp. 1–11.

— (2012). "A moving mesh finite volume interface tracking method for surface tension dominated interfacial fluid flow". In: *Computers and Fluids* 55, pp. 70–84.

Wang, Shi-qing (2017). *Nonlinear Polymer Rheology*. Wiley. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119029038.

Acknowledgements

An dieser Stelle möchte ich mich zunächst ganz herzlich bei Frau Frauke Enders für die fachliche Betreuung dieser Arbeit danken, sowie Herrn Prof. Dr.-Ing. Matthias Kraume für die Möglichkeit am Fachbereich Verfahrentechnik der TU Berlin forschen zu dürfen.

Besonderer Dank gilt Frau Prof. Dr.-Ing. Mirjam Altendorfner für die Prüfung dieser Arbeit, vor allem jedoch für die stetige Unterstützung und die Möglichkeit meine Ideen und Projekte während des gesamten Masters zu verwirklichen zu können.

Ebenso möchte ich mich ganz herzlich bei meinen Eltern und meiner Oma für die moralische und finanzielle Unterstützung bedanken, ohne die das Anfertigen dieser Arbeit nicht möglich gewesen wäre. Zuletzt möchte ich noch all denjenigen danken, die in der Zeit der Erstellung dieser Arbeit für mich da waren, Korrektur gelesen haben und mir emotionalen Rückhalt gegeben haben, insbesondere meiner Freundin Juli.

Appendix A

Files

A.1 system directory

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      decomposeParDict;
}

numberOfSubdomains   4;

//method               simple;
method               scotch;
//method               hierarchical;
//method               metis;
//method               manual;

preserveFaceZones (selectedFZ);
preservePatches (freeSurface freeSurfaceShadow);
singleProcessorFaceSets ((selectedFaces 0));
preserveBaffles true;


simpleCoeffs
{
    n                   (2 2 1);
    delta               0.001;
}

hierarchicalCoeffs
{
    n                   (2 2 1);
    delta               0.001;
    order               xyz;
}

manualCoeffs
{
```

```
    dataFile                "decompositionData";
}
```

CODE SAMPLE A.1: decomposeParDict

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    * //
actions
(
  {
    name    selectedFaces;
    type    faceSet;
    action  new;
    source  patchToFace;
    sourceInfo
    {
      name freeSurface;
    }
  }

  {
    name    selectedFaces;
    type    faceSet;
    action  add;
    source  patchToFace;
    sourceInfo
    {
      name freeSurfaceShadow;
    }
  }
  {
    name    selectedFZ;
    type    faceZoneSet;
    action  new;
    source  setToFaceZone;
    sourceInfo
    {
      faceSet selectedFaces;
    }
  }
);
```

CODE SAMPLE A.2: topoSetDict

Appendix B

Mesh quality reports

B.1 2D

```
Mesh stats
    all points:           28002
    live points:          28002
    all faces:            55280
    live faces:           55280
    internal faces:       27280
    cells:                13760
    boundary patches:     4
    point zones:          0
    face zones:           0
    cell zones:           0

Overall number of cells of each type:
    hexahedra:     13760
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    0
    polyhedra:     0

Checking topology...
    Boundary definition OK.
    Point usage OK.
    Upper triangular ordering OK.
    Face vertices OK.
   *Number of regions: 2
    The mesh has multiple regions which are not connected by any face.
  <<Writing region information to "0/cellToRegion"
Nuumber of cells per region:
   0  9280
   1  4480


Checking patch topology for multiply connected surfaces ...
    Patch               Faces    Points   Area [m^2]  Surface topology
    space               160      320      7.99667e-05 ok (non-closed singly
     connected)
```

```
38     freeSurfaceShadow   160      320      3.99834e-06 ok (non-closed singly
       connected)
39     freeSurface         160      320      3.99834e-06 ok (non-closed singly
       connected)
40     frontAndBackPlanes  27520    28002    0.00141335  ok (non-closed singly
       connected)
41
42 Checking geometry...
43     This is a 2-D mesh
44     Overall domain bounding box (-0.015 -0.015 -0.000424264) (0.015 0.015
      0.000424264)
45     Mesh (non-empty, non-wedge) directions (1 1 0)
46     Mesh (non-empty) directions (1 1 0)
47     Mesh (non-empty, non-wedge) dimensions 2
48     All edges aligned with or perpendicular to non-empty directions.
49     Boundary openness (-4.8265e-18 3.39336e-18 4.01988e-18) Threshold = 1e
      -06 OK.
50     Max cell openness = 2.19754e-16 OK.
51     Max aspect ratio = 116.813 OK.
52     Minumum face area = 1.39635e-10. Maximum face area = 1.15706e-06.  Face
       area magnitudes OK.
53     Min volume = 1.18484e-13. Max volume = 6.5042e-10.  Total volume =
      5.99634e-07.  Cell volumes OK.
54     Mesh non-orthogonality Max: 26.0936 average: 5.59252 Threshold = 70
55     Non-orthogonality check OK.
56     Face pyramids OK.
57     Max skewness = 0.35902 OK.
58
59 Mesh OK.
60
61 End
```

CODE SAMPLE B.1: 2D mesh check report

B.2 3D

```
 1
 2 Mesh stats
 3     all points:           2392795
 4     live points:          2392795
 5     all faces:            2798752
 6     live faces:           2798752
 7     internal faces:       2785218
 8     cells:                405954
 9     boundary patches:     3
10     point zones:          0
11     face zones:           1
12     cell zones:           0
13
14 Overall number of cells of each type:
```

```
    hexahedra:     250
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    0
    polyhedra:     405704

Checking topology...
    Boundary definition OK.
    Point usage OK.
    Upper triangular ordering OK.
    Face vertices OK.
   *Number of regions: 2
    The mesh has multiple regions which are not connected by any face.
  <<Writing region information to "0/cellToRegion"
Nuumber of cells per region:
   0  11982
   1  393972


Checking patch topology for multiply connected surfaces ...
    Patch               Faces    Points   Area [m^2]  Surface topology
    freeSurfaceShadow   2974     5944     7.0485e-06  ok (closed singly
    connected)
    freeSurface         2974     5944     7.0485e-06  ok (closed singly
    connected)
    space               7586     15168    0.00282471  ok (closed singly
    connected)

Checking geometry...
    This is a 3-D mesh
    Overall domain bounding box (-0.0149916 -0.0149839 -0.0149926)
    (0.0149915 0.0149839 0.014993)
    Mesh (non-empty, non-wedge) directions (1 1 1)
    Mesh (non-empty) directions (1 1 1)
    Mesh (non-empty, non-wedge) dimensions 3
    Boundary openness (5.75353e-17 7.08993e-18 6.11298e-16) Threshold = 1e
    -06 OK.
    Max cell openness = 2.44358e-16 OK.
    Max aspect ratio = 6.22447 OK.
    Minumum face area = 1.17441e-10. Maximum face area = 8.31129e-07.  Face
     area magnitudes OK.
    Min volume = 5.78447e-15. Max volume = 1.3458e-09.  Total volume =
    1.41135e-05.  Cell volumes OK.
    Mesh non-orthogonality Max: 52.4113 average: 10.795 Threshold = 70
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 1.23522 OK.

Mesh OK.
```

```
59
60 End
```

CODE SAMPLE B.2: 3D mesh check report: 406k cells

```
 1 Mesh stats
 2     all points:           6940372
 3     live points:          6940372
 4     all faces:            8115888
 5     live faces:           8115888
 6     internal faces:       8090767
 7     cells:                1175513
 8     boundary patches:     3
 9     point zones:          0
10     face zones:           1
11     cell zones:           0
12
13 Overall number of cells of each type:
14     hexahedra:     1250
15     prisms:        0
16     wedges:        0
17     pyramids:      0
18     tet wedges:    0
19     tetrahedra:    0
20     polyhedra:     1174263
21
22 Checking topology...
23     Boundary definition OK.
24     Point usage OK.
25     Upper triangular ordering OK.
26     Face vertices OK.
27    *Number of regions: 2
28     The mesh has multiple regions which are not connected by any face.
29   <<Writing region information to "0/cellToRegion"
30 Nuumber of cells per region:
31    0  37658
32    1  1137855
33
34
35 Checking patch topology for multiply connected surfaces ...
36     Patch               Faces    Points   Area [m^2]  Surface topology
37     freeSurfaceShadow   6182     12360    7.05458e-06 ok (closed singly
       connected)
38     freeSurface         6182     12360    7.05458e-06 ok (closed singly
       connected)
39     space               12757    25510    0.00282587  ok (closed singly
       connected)
40
41 Checking geometry...
42     This is a 3-D mesh
```

```
43     Overall domain bounding box (-0.0149956 -0.0149955 -0.0149932)
       (0.0149956 0.0149955 0.014993)
44     Mesh (non-empty, non-wedge) directions (1 1 1)
45     Mesh (non-empty) directions (1 1 1)
46     Mesh (non-empty, non-wedge) dimensions 3
47     Boundary openness (9.89642e-17 -9.13692e-18 -7.25641e-16) Threshold = 1
       e-06 OK.
48     Max cell openness = 2.57767e-16 OK.
49     Max aspect ratio = 6.22964 OK.
50     Minumum face area = 3.57786e-11. Maximum face area = 3.87447e-07.  Face
        area magnitudes OK.
51     Min volume = 1.32632e-15. Max volume = 5.7399e-10.  Total volume =
       1.41237e-05.  Cell volumes OK.
52     Mesh non-orthogonality Max: 51.6248 average: 10.7437 Threshold = 70
53     Non-orthogonality check OK.
54     Face pyramids OK.
55     Max skewness = 1.64247 OK.
56
57 Mesh OK.
58
59 End
```

CODE SAMPLE B.3: 3D mesh check report: 1.2M cells

Appendix C

2D simulation parameter

C.1 Solver comparison with water

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    * //

rheology
{

        type multiMode;

        models
        (
            first
            {
            type PTT-Exponential;
                rho                rho [1 -3 0 0 0 0 0] 1011;
                etaS               etaS [1 -1 -1 0 0 0 0] 41.650e-5;
                etaP               etaP [1 -1 -1 0 0 0 0] 41.650e-5;
                lambda             lambda [0 0 1 0 0 0 0] 0;
                epsilon            epsilon [0 0 0 0 0 0 0] 0;
                zeta               zeta [0 0 0 0 0 0 0] 0;
            }

        );

}
```

CODE SAMPLE C.1: viscoelasticProperties