DROPS-Package
User's guide

March 19, 2008

Contents

Chapter 1

Introduction

The development of the software package DROPS started in the interdisciplinary Collaborative Research Center SFB 540 "Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems" [84]. The goal of the DROPS related research activities is two-fold: on the one hand we want to develop and improve numerical methods for the simulation of two-phase flow problems and on the other hand the aim is to provide accurate and reliable numerical simulations of realistic two-phase systems which are of interest for project partners from engineering departments.

The development of DROPS is mainly carried out at the Chair of Numerical Mathematics, RWTH Aachen University. Due to the complexity of two-phase flow problems we need the ability to perform parallel computations. In a tight cooperation the parallelization of DROPS is realized at the Chair of Scientific Computing, RWTH Aachen University.

The DROPS code is written in C++. Especially the implementation of the iterative solvers heavily uses the object-oriented and template programming features of C++[1].

Some further information including a gallery of simulation examples can be found on the DROPS website [16]. More information concerning the numerical methods used and on results of simulations with can be found in a sequence of papers [1]-[13].

In this guide we describe the main features of the DROPS package, which has been developed at the Chair of Numerical Mathematics at RWTH Aachen University. The guide consists of three parts. In this introductory chapter we describe the *models of one- and two-phase flow problems* that can be solved numerically using DROPS (sections 1.1 and 1.2) and we give an overview of the numerical methods that are available for such simulations. In part I we explain these *numerical methods*. Our treatment, however, is restricted to a rather global description of the methods and for further information references to the literature are given. In the second part we discuss *implementation issues* such as data structures, important classes and routines corresponding to the numerical methods. In the third part we present *several examples* which show how numerical simulations of different one- and two-phase flow models can be realized in DROPS.

Three-dimensional incompressible two-phase flow problems, with or without mass transport, form the main application area of DROPS. The following models fit in the DROPS framework and will be explained in more detail in section 1.1:

1) Navier-Stokes equations for one-phase flow (NS1).

[1] Our code is used by some compiler manufacturers as a benchmark test for their C++ compilers (e. g., SUN, Microsoft).

2) Navier-Stokes equations for two-phase flow (NS2).

3) NS2 combined with transport of a dissolved species (NS2+T).

4) NS2 combined with transport of a surfactant *at* the interface (NS2+S).

5) NS2 combined with transport of both a dissolved species and a surfactant at the interface (NS2+T+S).

In the models 2)-5) surface tension forces are taken into account.

We list some main features of the numerical methods that are used in DROPS for the simulation of these models:

- A *multilevel* hierarchy of *tetrahedral* triangulations is used. Local refinement and coarsening routines are available.
- For spatial discretization we apply Finite Element (FE) techniques based on conforming spaces. Special FE spaces suitable for functions that are discontinuous across the interface can be used.
- A level set method for interface capturing is used.
- A special Laplace-Beltrami method for the discretization of surface tension forces is implemented.
- A "sharp" interface approach is applied in the sense that there is no smoothing or regularization of discontinuities or delta functions at the interface.
- We use implicit time integration methods in which flow variables, surface tension forces and the level set function are strongly coupled.
- For the solution of large sparse linear systems preconditioned Krylov subspace methods can be applied. Several efficient preconditioners are available. Inexact Uzawa type solvers for saddle point problems are implemented. Multigrid solvers can be used.
- An MPI based parallel version of DROPS is currently developed.

For input an interface to the GAMBIT package is implemented and for visualization purposes data can be transfered to Ensight, Tecplot, Geomview.

DROPS is written in C++. In chapter 8 some properties of the implementation are treated. Currently a parallel version on DROPS is under development. More information on parallelization issues is given in chapter 9.

In the remainder of this introduction we describe the models 1)-5) more precisely and give an overview of the numerical methods. For ease of presentation the partial differential equations used in the models are given in the *strong* formulation (section 1.1). The numerical methods, in particular the FE methods for spatial discretization, are based on the *weak* formulation of these partial differential equations, as explained in more detail in part I. In section 1.2 we address the issue of initial and boundary conditions used in our models. The brief summary of numerical methods given in section 1.3 is elaborated in part I.

1.1 One- and two-phase flow models in strong formulation

In this section we give the partial differential equations (in strong formulation) which describe the different one- and two-phase incompressible flow models that can be solved numerically using DROPS.

We always assume that the computational domain $\Omega \subset \mathbb{R}^3$ is a bounded polyhedral domain.

1) **Navier-Stokes equations for one-phase flow** (NS1)

Let $\mathbf{u} = \mathbf{u}(x,t)$ and $p(x,t)$ be the velocity and pressure. For these unknowns we consider the standard incompressible Navier-Stokes equations

$$\begin{aligned} \rho\big(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\big) &= -\nabla p + \rho\mathbf{g} + \operatorname{div}(\mu\mathbf{D}(\mathbf{u})) \text{ in } \Omega \\ \operatorname{div}\mathbf{u} &= 0 \text{ in } \Omega, \end{aligned} \tag{1.1}$$

with a strictly positive density ρ and viscosisty μ, and the strain tensor $\mathbf{D}(\mathbf{u}) := \nabla\mathbf{u} + (\nabla\mathbf{u})^T$. The vector $\mathbf{g}$ is a known external force (gravity). Initial and boundary conditions corresponding to these Navier-Stokes equations are discussed in section 1.2.

Remark 1 The standard case is that ρ and μ are strictly positive *constants*. Using $\operatorname{div}\mathbf{u} = 0$ we then get

$$\operatorname{div}(\mu\mathbf{D}(\mathbf{u})) = \mu\Delta\mathbf{u} = \mu \begin{pmatrix} \Delta u_1 \\ \Delta u_1 \\ \Delta u_3 \end{pmatrix}.$$

2) **Navier-Stokes equations for two-phase flow** (NS2)

We consider the situation in which the domain $\Omega \subset \mathbb{R}^3$ contains two different immiscible incompressible newtonian phases (fluid-fluid or fluid-gas). A model problem is a liquid drop contained in a surrounding fluid. The time-dependent domains which contain the phases are denoted by $\Omega_1 = \Omega_1(t)$ and $\Omega_2 = \Omega_2(t)$ with $\overline{\Omega}_1 \cup \overline{\Omega}_2 = \overline{\Omega}$. The interface between the two phases ($\partial\Omega_1 \cap \partial\Omega_2$) is denoted by $\Gamma = \Gamma(t)$. To model the forces at the interface we make the standard assumption that the surface tension balances the jump of the normal stress on the interface, i. e., we have a free boundary condition

$$[\boldsymbol{\sigma}\mathbf{n}]_\Gamma = \tau\mathcal{K}\mathbf{n},$$

with $\mathbf{n} = \mathbf{n}_\Gamma$ the unit normal at the interface (pointing from Ω_1 in Ω_2), τ the surface tension coefficient (material parameter), $\mathcal{K}$ the curvature of Γ and $\boldsymbol{\sigma}$ the stress tensor, i. e., $\boldsymbol{\sigma} = -p\mathbf{I} + \mu\mathbf{D}(\mathbf{u})$. Due to the assumption that the phases are viscous and immiscible the velocity should be continuous across the interface. In combination with the conservation laws of mass and momentum this yields the following standard model:

$$\begin{aligned} &\begin{cases} \rho_i\big(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\big) = -\nabla p + \rho_i\mathbf{g} + \operatorname{div}(\mu_i\mathbf{D}(\mathbf{u})) & \text{in } \Omega_i \\ \operatorname{div}\mathbf{u} = 0 & \text{in } \Omega_i \end{cases} \quad \text{for } i = 1,2, \\ &[\boldsymbol{\sigma}\mathbf{n}]_\Gamma = \tau\mathcal{K}\mathbf{n}, \quad [\mathbf{u}\cdot\mathbf{n}]_\Gamma = 0\,. \end{aligned} \tag{1.2}$$

Initial conditions and boundary conditions are discussed in section 1.2.

This model for a two-phase incompressible flow problem is often used in the literature. The effect of the surface tension can be expressed in terms of a localized force at the interface, cf. the so-called continuum surface force (CSF) model [32, 38]. This localized force is given by

$$f_\Gamma = \tau \mathcal{K} \delta_\Gamma \mathbf{n}_\Gamma \ .$$

Here δ_Γ is a Dirac δ-function with support on Γ. Its action on a smooth test function ψ is given by

$$\int_\Omega \delta_\Gamma(x)\psi(x)\,dx = \int_\Gamma \psi(s)\,ds \ .$$

This localization approach can be combined with the level set method for capturing the unknown interface. We outline the main idea, for a detailed treatment we refer to the literature [38]. The level set function, denoted by $\phi = \phi(x,t)$ is a scalar function. At the initial time $t = 0$ we assume a function $\phi_0(x)$ such that $\phi_0(x) < 0$ for $x \in \Omega_1(0)$, $\phi_0(x) > 0$ for $x \in \Omega_2(0)$, $\phi_0(x) = 0$ for $x \in \Gamma(0)$. It is desirable to have the level set function as an approximate signed distance function. For the evolution of the interface we consider the trace $x(t)$ of a single particle $x(0) \in \Omega$ over time. A particle on the interface remains on the interface for all time, i. e., for all $x(0) \in \Gamma(0)$ and all $t \geq 0$ we have $x(t) \in \Gamma(t)$. This is equivalent to the condition $\phi(x(t),t) = 0$ $(t \geq 0)$ which we extend to the whole domain as $\phi(x(t),t) = \phi(x(0),0)$ for all $x(0) \in \Omega$ and all $t \geq 0$. By differentiating this condition with respect to t we obtain $\phi_t + \nabla\phi(x,t) \cdot x_t = 0$. The displacement of a particle coincides with the velocity field, i. e., $x_t = \mathbf{u}$ holds. Hence one obtains the first order differential equation $\phi_t + \mathbf{u} \cdot \nabla\phi = 0$ for $t \geq 0$ and $x \in \Omega$.

The jumps in the coefficients ρ and μ can be described using the level set function (which has its zero level set precisely at the interface Γ) in combination with the Heaviside function $H : \mathbb{R} \to \mathbb{R}$:

$$H(\zeta) = 0 \quad \text{for} \ \ \zeta < 0 \ , \qquad H(\zeta) = 1 \quad \text{for} \ \ \zeta > 0 \ .$$

For ease one can take $H(0) = \frac{1}{2}$. We define

$$\begin{aligned} \rho(\phi) &:= \rho_1 + (\rho_2 - \rho_1)H(\phi) \\ \mu(\phi) &:= \mu_1 + (\mu_2 - \mu_1)H(\phi). \end{aligned} \tag{1.3}$$

Combination of the CSF approach with the level set method leads to the following model for the two-phase problem in $\Omega \times [0,T]$:

$$\boxed{\begin{aligned} \rho(\phi)\Big(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\Big) &= -\nabla p + \rho(\phi)\mathbf{g} + \operatorname{div}(\mu(\phi)\mathbf{D}(\mathbf{u})) + \tau\mathcal{K}\delta_\Gamma\mathbf{n}_\Gamma \\ \operatorname{div}\mathbf{u} &= 0 \\ \phi_t + \mathbf{u}\cdot\nabla\phi &= 0, \end{aligned}} \tag{1.4}$$

together with suitable initial and boundary conditions for $\mathbf{u}$ and ϕ, cf. section 1.2.

3) NS2 combined with transport of a dissolved species (NS2+T)

We consider a two-phase flow problem as described above in NS2. We assume that one or both phases contain a dissolved species that is transported due to convection and diffusion and does not adhere to the interface. The concentration of this species is denoted by $c(x,t)$. This flow problem can be modeled by the equations (1.4) for the flow variables and a convection-diffusion

equation for the concentration c. At the interface we need interface conditions for c. The first interface condition comes from mass conservation, which implies flux continuity. The second condition results from a constitutive equation known as Henry's law (derived from continuity of chemical potentials at the interface) which states that the quotient of the concentration values at both sides of the interface equals a given constant (Henry's constant, which depends on the given phases). This model is as follows, cf. [31, 98]:

Two-phase flow model (1.4) combined with:

$$\begin{aligned} \frac{\partial c}{\partial t} + \mathbf{u} \cdot \nabla c &= D(\phi)\Delta c, \qquad (1.5)\\ [D(\phi)\nabla c \cdot \mathbf{n}] &= 0 \quad \text{at the interface,}\\ c_1 &= C_H c_2 \quad \text{at the interface.} \end{aligned}$$

The diffusion coefficient is piecewise constant: $D(\phi) = D_1 + (D_2 - D_1)H(\phi)$. In the interface condition we use the notation c_i for $c_{|\Omega_i}$ restricted to the interface. The constant $C_H > 0$ is given (Henry's constant). Using $\hat{C}(\phi) := 1 + (C_H - 1)H(\phi)$ the Henry interface condition can also be written as $[\hat{C}c] = 0$. The model has to be combined with suitable initial and boundary conditions, cf. section 1.2.

4) NS2 **combined with transport of a surfactant at the interface** (NS2+S)

We consider a two-phase flow problem as described above in NS2. We assume that there is a species (called tenside or surfactant) which adheres to the interface and that the concentration of this surfactant in the two phases is so small that it can be neglected in the model. The concentration of this surfactant is denoted by $S(x,t)$, $x \in \Gamma$. We introduce the orthogonal projection $P = I - \mathbf{n}\mathbf{n}^T$ ($\mathbf{n}$: normal on Γ). Correspondingly, for $x \in \Gamma$ we have an orthogonal decomposition $\mathbf{u}(x,t) = P\mathbf{u}(x,t) + (I-P)\mathbf{u}(x,t) =: \mathbf{u}_\Gamma(x,t) + \mathbf{u}_\perp(x,t)$. The tangential gradient is defined by $\nabla_\Gamma := P\nabla$, and $\mathrm{div}_\Gamma := \nabla_\Gamma^T$, $\Delta_\Gamma := \mathrm{div}_\Gamma \nabla_\Gamma$. The two-phase fluid dynamics model with transport of surfactants at the interface is as follows, cf. [31, 58]:

Two-phase flow model (1.4) combined with:

$$\partial_{t,n} S + \mathrm{div}_\Gamma(S\mathbf{u}_\Gamma) + S\mathcal{K}\mathbf{u} \cdot \mathbf{n} = D_\Gamma \Delta_\Gamma S. \qquad (1.6)$$

The derivative $\partial_{t,n} S$ stands for the time derivative of S along a normal path. The diffusion coefficient D_Γ can be assumed to be constant on Γ. For the convection-diffusion equation in (1.6) no boundary conditions are needed if the interface Γ is a manifold without boundary.

5) NS2 **combined with transport of both a dissolved species and a surfactant at the interface** (NS2+T+S)

Forthcoming. This model combines the models in (1.5) and (1.6). If c in (1.5) models the concentration of the surfactant in the two phases, then in (1.6) one usually includes an additional source term that accounts for the change of the surfactant concentration at the interface (S) due to ad- and desorption, cf. [31, 98].

1.2 Initial and boundary conditions

In this section we describe the initial and boundary conditions that can be used in the models 1)-5) to make the problem well-posed.

For the NS1 model one needs suitable initial and boundary conditions only for the velocity $\mathbf{u}$. The initial condition is $\mathbf{u}(x,t) = \mathbf{u}_0(x)$ with a given function $\mathbf{u}_0$, which usually comes from the underlying physical problem. For the boundary conditions we distinguish between *essential* and *natural* boundary conditions. Let $\partial\Omega$ be subdivided into two parts $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ with $\partial\Omega_D \cap \partial\Omega_N = \emptyset$. We use essential boundary conditions on $\partial\Omega_D$ that are of Dirichlet type. In applications these describe inflow conditions or conditions at walls (e.g., no-slip). Such Dirichlet conditions are of the form $\mathbf{u}(x,t) = \mathbf{u}_D(x,t)$ for $x \in \partial\Omega_D$, with a given function $\mathbf{u}_D$. If, for example, $\partial\Omega_D$ corresponds to a fixed wall, then a no-slip boundary condition is given by $\mathbf{u}(x,t) = 0$ for $x \in \partial\Omega_D$. On $\partial\Omega_N$ we prescribe natural boundary conditions, which are often used to describe outflow conditions. These natural boundary conditions are of the form

$$\boldsymbol{\sigma}\mathbf{n}_\Omega = -p_{ext}\mathbf{n}_\Omega, \quad \text{on} \quad \partial\Omega_N, \tag{1.7}$$

with $\mathbf{n}_\Omega$ the outward pointing normal on $\partial\Omega$ and p_{ext} a given function (external pressure). For the case $p_{ext} = 0$ we thus obtain a homogeneous natural boundary condition.

For the two-phase flow model NS2 in addition we have to consider initial and boundary conditions for the level set funtion ϕ. The initial condition is $\phi(x,0) = \phi_0(x)$, in which ϕ_0 is given and should be such that $\{\, x \in \mathbb{R}^3 \mid \phi_0(x) = 0 \,\} = \Gamma(0)$. Moreover, ϕ_0 should be an (approximate) signed distance function to $\Gamma(0)$. To make the linear hyperbolic equation $\phi_t + \mathbf{u}\cdot\nabla\phi = 0$ well-posed one needs boundary conditions on the inflow boundary $\partial\Omega_{in} := \{\, x \in \partial\Omega \mid \mathbf{u}\cdot\mathbf{n}_\Omega < 0 \,\}$. The function ϕ in the NS2 model (1.4) is introduced for the numerical purpose of capturing the interface and has no physical interpretation. There are no natural (e.g., physics based) boundary conditions for ϕ at the inflow boundary. We are only interested in values of ϕ close to the interface (= zero level on ϕ) and ϕ is evolved according to $\phi_t + \mathbf{u}\cdot\nabla\phi = 0$ only for a short time interval. After this short time a re-initialization of ϕ is applied, cf. section 4.2.6. Due to this one can use the level set equation in (1.4) *without any boundary conditions* on $\partial\Omega$.

In the model NS2+M in (1.5) one needs in addition initial and boundary conditions for the concentration c. The initial condition is $c(x,0) = c_0(x)$ with a given initial concentration c_0. For the boundary conditions the standard ones, namely a Dirichlet (i.e., c given on part of $\partial\Omega$) and a Neumann ($\frac{\partial c}{\partial \mathbf{n}_\Omega}$ given on part of the boundary) condition can be used.

In model NS2+S in (1.6) one has to prescribe an inititial concentration $S(x,0) = S_0(x),\ x \in \Gamma$, for the surfactant. If Γ is a manifold without boundary (droplet) no boundary conditions for S are needed.

1.3 Overview of numerical methods

For the numerical simulation of the models 1)-5) many numerical methods are implemented in DROPS. These methods are explained in part I. In this section we present a compact overview of all important methods used, in the form of a matrix of methods, cf. table 1.1. As can be seen from this table, we arrange the different methods according to two criteria, namely *the models for which they are used* and *the computational method class they belong to.* To be more specific we briefly address the methods shown in table 1.1 in a row wise order.

Model NS1. For the one-phase Navier-Stokes problem we use a multilevel hierarchy of nested tetrahedral meshes. Local refinement and coarsening methods are available, too. These *grid*

related methods are also used for all other models. For spatial discretization we use the standard Hood-Taylor P_2-P_1 *finite element pair*. We need *quadrature* rules to evaluate the integrals that occur in the weak formulation of the problem. After spatial discretization we obtain a large system of nonlinear ordinary differential equations coupled with algebraic constraints (due to $\operatorname{div} \mathbf{u} = 0$), i.e., a DAE system (Differential Algebraic Equation). For this system we use a *numerical time integration rule*. In DROPS a fractional-step θ-scheme and a method based on operator splitting are available. Per time step such a time integration rule results in a large nonlinear system of algebraic equations, in which velocity $\mathbf{u}$ and pressure p are coupled. For *linearization* we apply a standard Picard iteration (with steplength optimization). After linearization we have a large sparse linear system of algebraic equations that is of saddle point type. Several efficient *iterative solvers*, like for example preconditioned CG (PCG) and multigrid methods (MG), are available.

Model NS2. One very important issue for this class of problems is the *discretization of the level set equation*. For this we use P_2 finite elements combined with streamline diffusion stabilization (SDFEM). Another topic is the *approximation of the zero level* of this discretization ϕ_h of ϕ ($\Gamma \rightsquigarrow \Gamma_h$). Related to the level set funtion we also need a *re-initialization method*. A further issue is the *discretization of the localized surface tension force* in (1.4). For this we use a Laplace-Beltrami technique. In this type of problems, due to surface tension, the pressure is discontinuous across the interface. For an appropriate treatment of this discontinuity we use a *special finite element space* (P_1X). Due to this discontinuity and discontinuities in density and viscosity we need *special quadrature rules*. After application of a time integration rule we obtain a large nonlinear system of algebraic equations in which $\mathbf{u}, p$ and ϕ are coupled. We apply an iterative *decoupling strategy* to split the coupled problem for $\mathbf{u}, p, \phi$ into two subproblems for $\mathbf{u}, p$ and ϕ, respectively. If in the model we have very large jumps in density and viscosity across the interface (as, for example, in a liquid-gas system) then in the iterative solvers we need *special preconditioners* that are robust with respect to variation in the size of these jumps.

Model NS2+M. For the spatial discretization of the convection-diffusion equation for the concentration c we use standard P_1 finite elements. Due to the interface condition $c_1 = C_H c_2$ (with $C_H \neq 1$) in (1.5), the concentration is discontinuous across the interface. A special *jump removing transformation* is used to eliminate this discontinuity. An alternative approach is to use a P_1X finite element space (instead of P_1) for spatial discretization. For the *time integration* a standard θ-scheme (which includes Euler backward and the Crank-Nicolson method) is available. A simple method is used for the *decoupling* of $(\mathbf{u}, p, \phi)$ and c in each time step.

Model NS2+S. In this model we have a convection-diffusion equation on the (moving) interface Γ, cf. (1.6). For the spatial discretization we use *special finite element spaces* that are obtained from suitable restriction of the P_1 finite element space corresponding to the tetrahedral triangulation. Furthermore, interface adapted quadrature rules are needed. The time integration has to be adapted to the special time derivative $\partial_{t,n}$ that occurs in (1.6).

Model NS2+M+S. Forthcoming.

	Grids	spatial discretization	time integration	treatment of couplings /linearization	iterative solvers
NS1	multilevel tetrahedral hierarchy; local refinement; local coarsening;	P_2-P_1 FE; quadrature;	θ-scheme; fractional-step scheme operator splitting;	$(\mathbf{u}, p)$ fully coupled; Picard iteration for linearization;	PCG, MG; inexact Uzawa; Schur compl. precond.; GMRES, GCR, MINRES; SSOR, Jacobi;
NS2	$\downarrow$	$\downarrow$ P_2-P_1X; P_2+ SDFEM for ϕ; mass conservation; re-initialization of ϕ; $\Gamma \rightsquigarrow \Gamma_h$; discretization of f_Γ; special quadrature;	$\downarrow$	$\downarrow$ fixed point for decoupling $(\mathbf{u}, p)$; defect correction;	$\downarrow$ special preconditioners (due to jumps);
NS2+M	$\downarrow$	$\downarrow$ P_1 + transf. for c; P_1X for c;	$\downarrow$ θ-scheme for c;	$\downarrow$ decoupling of $(\mathbf{u}, p, \phi) \leftrightarrow c$;	$\downarrow$
NS2+S	$\downarrow$	$\downarrow$ FE space on Γ_h; quadrature on Γ_h;	$\downarrow$ discr. of time derivative $\partial_{t,n}$;	$\downarrow$ decoupling of $(\mathbf{u}, p, \phi) \leftrightarrow s$;	$\downarrow$
NS2+M+S	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$ decoupling of $(\mathbf{u}, p, \phi) \leftrightarrow c, s$;	$\downarrow$

Table 1.1: Overview of numerical methods in DROPS

Part I

Numerical methods

In part I of this guide we will explain the numerical methods implemented in DROPS for the numerical simulation of the one- and two-phase flow problems described in section 1.1. The methods for spatial and time discretization and for the iterative solution of the resulting nonlinear and linear large sparse systems, as summarized in table 1.1, are treated. In chapter 2 we discuss the multilevel family of tetrahedral nested meshes that is used in all our simulations. In the chapters 3-7 we explain the methods used for the models 1)-5) given in section 1.1. The presentation follows the row wise ordering of the methods in table 1.1.

Chapter 2

Hierarchy of tetrahedral grids

We outline the basic ideas of the multilevel grid hierarchy on which our finite element discretization method is based. We only consider multilevel *tetrahedral* meshes based on *red/green refinement strategies* (cf. [20, 22, 27]). The idea of a *multilevel* refinement (and coarsening) strategy was introduced in [22] and further developed in [24, 27, 29, 63, 64, 97]. This grid refinement technique is used in UG [95]; for an overview we refer to [23, 25]. Similar techniques are used in several other packages.

We first introduce a few basic notions. We assume that Ω is a polyhedral domain.

Definition 1 (Triangulation) A finite collection $\mathcal{T} = \{T\}$ of tetrahedra $T \subset \overline{\Omega}$ is called a *triangulation* of Ω if the following holds:

1. $\mathrm{vol}(T) > 0$ for all $T \in \mathcal{T}$,
2. $\bigcup_{T\in\mathcal{T}} T = \overline{\Omega}$,
3. $\mathrm{int}(S) \cap \mathrm{int}(T) = \emptyset$ for all $S, T \in \mathcal{T}$ with $S \neq T$.

Definition 2 (Consistency) A triangulation $\mathcal{T}$ is called *consistent* if the intersection of any two tetrahedra in $\mathcal{T}$ is either empty, a common face, a common edge or a common vertex.

Definition 3 (Stability) A sequence of triangulations $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots$ is called *stable* if all angles of all tetrahedra in this sequence are uniformly bounded away from zero.

Definition 4 (Refinement) For a given tetrahedron T a triangulation $\mathcal{K}(T)$ of T is called a *refinement* of T if $|\mathcal{K}(T)| \geq 2$ and any vertex of any tetrahedron $T' \in \mathcal{K}(T)$ is either a vertex or an edge midpoint of T. In this case T' is called a child of T and T is called the parent of T'. A triangulation $\mathcal{T}_{k+1}$ is called *refinement* of a triangulation $\mathcal{T}_k \neq \mathcal{T}_{k+1}$ if for every $T \in \mathcal{T}_k$ either $T \in \mathcal{T}_{k+1}$ or $\mathcal{K}(T) \subset \mathcal{T}_{k+1}$ for some refinement $\mathcal{K}(T)$ of T.

Definition 5 (Multilevel triangulation) A sequence of consistent triangulations $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ is called a *multilevel triangulation* of Ω if the following holds:

1. For $0 \leq k < J$: $\mathcal{T}_{k+1}$ is a refinement of $\mathcal{T}_k$.
2. For $0 \leq k < J$: $T \in \mathcal{T}_k \cap \mathcal{T}_{k+1} \Rightarrow T \in \mathcal{T}_J$.

Definition 6 (Hierarchical decomposition) Let $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ be a multilevel triangulation of Ω. For every tetrahedron $T \in \mathcal{M}$ a unique level number $\ell(T)$ is defined by $\ell(T) := \min\{\, k \mid T \in \mathcal{T}_k \,\}$. The set $\mathcal{G}_k \subset \mathcal{T}_k$

$$\mathcal{G}_k := \{\, T \in \mathcal{T}_k \mid \ell(T) = k \,\}$$

is called the *hierarchical surplus* on level k ($k = 0, \ldots, J$). Note that $\mathcal{G}_0 = \mathcal{T}_0$, $\mathcal{G}_k = \mathcal{T}_k \setminus \mathcal{T}_{k-1}$ for $k = 1, \ldots, J$. The sequence $\mathcal{H} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$ is called the *hierarchical decomposition* of $\mathcal{M}$. Note that the multilevel triangulation $\mathcal{M}$ can be reconstructed from its hierarchical decomposition.

Remark 2 Let $\mathcal{M}$ be a multilevel triangulation and V_k ($0 \le k \le J$) be the corresponding finite element spaces of continuous functions $p \in C(\overline{\Omega})$ such that $p_{|T} \in P_q$ for all $T \in \mathcal{T}_k$ ($q \ge 1$). The refinement property 1 in definition 5 implies nestedness of these finite element spaces: $V_k \subset V_{k+1}$.

Now assume that based on some error indicator certain tetrahedra in the finest triangulation $\mathcal{T}_J$ are marked for refinement. In many refinement algorithms one then modifies the finest triangulation $\mathcal{T}_J$ resulting in a new one, $\mathcal{T}_{J+1}$. Using such a strategy (which we call a *one*-level method) the new sequence $(\mathcal{T}_0, \ldots, \mathcal{T}_{J+1})$ is in general not a multilevel triangulation because the nestedness property 1 in definition 5 does not hold. We also note that when using such a method it is difficult to implement a reasonable coarsening strategy. In *multilevel* refinement algorithms the whole sequence $\mathcal{M}$ is used and as output one obtains a sequence $\mathcal{M}' = (\mathcal{T}'_0, \ldots, \mathcal{T}'_{J'})$, with $\mathcal{T}'_0 = \mathcal{T}_0$ and $J' \in \{J-1, J, J+1\}$. In general one has $\mathcal{T}'_k \neq \mathcal{T}_k$ for $k > 0$. We list a few important properties of this method:

- Both the input and output are *multilevel triangulations*.
- The method yields *stable* and *consistent* triangulations.
- Local *refinement* and *coarsening* are treated in a similar way.
- The implementation uses only the hierarchical decomposition of $\mathcal{M}$. This allows *relatively simple data structures* without storage overhead.
- The costs are proportional to the number of tetrahedra in $\mathcal{T}_J$.

For a detailed discussion of these and other properties we refer to the literature ([22, 27, 63, 1]). In our implementation we use the multilevel refinement algorithm described in [1].

Chapter 3

Navier-Stokes equations for one-phase flow (NS1)

3.1 Weak formulation

We consider the Navier-Stokes model as in (1.1). For simplicity we take $\rho \equiv 1$, μ =constant and homogeneous Dirichlet boundary conditions for $\mathbf{u}$ on $\partial\Omega$. For the weak formulation of this problem we introduce the spaces

$$\mathbf{V} := H_0^1(\Omega)^3, \quad Q := L_0^2(\Omega) = \{\, q \in L^2(\Omega) \mid \int_\Omega q = 0 \,\}. \tag{3.1}$$

A weak formulation of this problem is as follows:

Determine $\mathbf{u}(t) \in \mathbf{V}$ and $p(t) \in Q$ such that

$$\begin{aligned} m(\mathbf{u}_t(t), \mathbf{v}) + a(\mathbf{u}(t), \mathbf{v}) + c(\mathbf{u}(t); \mathbf{u}(t), \mathbf{v}) - b(\mathbf{v}, p(t)) &= (\mathbf{g}, \mathbf{v}) \qquad \forall \mathbf{v} \in \mathbf{V} \\ b(\mathbf{u}(t), q) &= 0 \qquad \forall q \in Q, \end{aligned}$$

for (almost) all $t \in [0, T]$. Here and in the remainder, $(\cdot, \cdot)$ denotes the (component wise) L^2 scalar product. In this variational formulation we used the bilinear forms

$$\begin{aligned} m(\mathbf{u}, \mathbf{v}) &= \int_\Omega \mathbf{u} \cdot \mathbf{v} \, dx \\ a(\mathbf{u}, \mathbf{v}) &= \mu \int_\Omega \left(\nabla \mathbf{u} \cdot \nabla \mathbf{v} \right) \, dx \\ b(\mathbf{u}, q) &= \int_\Omega q \, \mathrm{div}\, \mathbf{u} \, dx, \end{aligned}$$

and the trilinear form

$$c(\mathbf{u}; \mathbf{v}, \mathbf{w}) = \int_\Omega (\mathbf{u} \cdot \nabla \mathbf{v}) \cdot \mathbf{w} \, dx.$$

3.2 Spatial discretization

3.2.1 Galerkin finite element discretization

Let $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ be a multilevel triangulation of Ω. With each triangulation $\mathcal{T}_k$ $(0 \le k \le J)$ we associate a mesh size parameter $h = h_k$. Let $\mathbf{V}_h \subset \mathbf{V}$, $Q_h \subset L^2(\Omega)$ and $V_h \subset V$ be standard

polynomial finite element spaces corresponding to the triangulation $\mathcal{T}_k$. We assume the pair $(\mathbf{V}_h, Q_h)$ to be LBB stable. The standard pair used in DROPS is Hood-Taylor P_2-P_1, i.e., piecewise quadratic finite elements for the velocity and piecewise linears for the pressure.

Using such a pair $(\mathbf{V}_h, Q_h)$, the Galerkin discretization reads: Find $\mathbf{u}_h(t) \in \mathbf{V}_h$, $p_h(t) \in Q_h$, with $(p_h(t), 1) = 0$, such that for all $\mathbf{v}_h \in \mathbf{V}_h$ and all $q_h \in Q_h$:

$$\begin{aligned} m((\mathbf{u}_h)_t(t), \mathbf{v}_h) + a(\mathbf{u}_h(t), \mathbf{v}_h) + c(\mathbf{u}_h(t); \mathbf{u}_h(t), \mathbf{v}_h) - b(\mathbf{v}_h, p_h(t)) &= (\mathbf{g}, \mathbf{v}_h) \\ b(\mathbf{u}_h(t), q_h) &= 0. \end{aligned} \tag{3.2}$$

for all $t \in [0, T]$. Let $\{\boldsymbol{\xi}_i\}_{1 \le i \le N}$ and $\{\psi_i\}_{1 \le i \le K}$ be the standard nodal bases of the finite element spaces $\mathbf{V}_h$, Q_h and consider the representations

$$\mathbf{u}_h(t) = \sum_{j=1}^{N} u_j(t) \boldsymbol{\xi}_j, \quad \vec{\mathbf{u}}(t) := (u_1(t), \ldots, u_N(t)) \tag{3.3}$$

$$p_h(t) = \sum_{j=1}^{K} p_j(t) \psi_j, \quad \vec{\mathbf{p}}(t) := (p_1(t), \ldots, p_K(t)). \tag{3.4}$$

Using this the Galerkin discretization (3.2) can be rewritten as

$$\boxed{\begin{aligned} \mathbf{M} \frac{d\vec{\mathbf{u}}}{dt}(t) + \mathbf{A}\vec{\mathbf{u}}(t) + \mathbf{N}(\vec{\mathbf{u}}(t))\vec{\mathbf{u}}(t) + \mathbf{B}^T \vec{\mathbf{p}}(t) &= \vec{\mathbf{g}} \\ \mathbf{B}\vec{\mathbf{u}}(t) &= 0, \end{aligned}} \tag{3.5}$$

where

$$\begin{aligned} \mathbf{M} \in \mathbb{R}^{N \times N}, \quad & \mathbf{M}_{ij} = \int_\Omega \boldsymbol{\xi}_i \cdot \boldsymbol{\xi}_j \, dx \\ \mathbf{A} \in \mathbb{R}^{N \times N}, \quad & \mathbf{A}_{ij} = \mu \int_\Omega \nabla \boldsymbol{\xi}_i \cdot \nabla \boldsymbol{\xi}_j \, dx \\ \mathbf{B} \in \mathbb{R}^{K \times N}, \quad & \mathbf{B}_{ij} = -\int_\Omega \psi_i \operatorname{div} \boldsymbol{\xi}_j \, dx \\ \mathbf{N}(\vec{\mathbf{u}}) = \mathbf{N}(\mathbf{u}_h) \in \mathbb{R}^{N \times N}, \quad & \mathbf{N}(\vec{\mathbf{u}})_{ij} = \int_\Omega (\mathbf{u}_h \cdot \nabla \boldsymbol{\xi}_j) \cdot \boldsymbol{\xi}_i \, dx \\ \vec{\mathbf{g}} \in \mathbb{R}^N, \quad & \vec{\mathbf{g}}_i = \int_\Omega \mathbf{g} \cdot \boldsymbol{\xi}_i \, dx. \end{aligned}$$

Thus we obtain a system of DAEs for the unknown vector functions $\vec{\mathbf{u}}(t)$, $\vec{\mathbf{p}}(t)$. Numerical methods for the solution of this system are discussed in section 3.3.

3.2.2 Quadrature

For the computation of the entries in the matrices in (3.5) integrals have to be determined. For this we implemented several quadrature rules. In this section we discuss some of these quadrature rules.

The nodes and weights in these rules are denoted by x^i and w^i, $i = 1, \ldots, m$, respectively. The method Q_T is of the form

$$\int_T f(x) \, dx \approx Q_T(f) := \operatorname{vol}(T) \sum_{i=1}^{m} w^i f(x^i).$$

We use quadrature rules which have degree of exactness two and five on a tetrahedron. These rules are of the form

$$\int_T f(x)\,dx = \mathrm{vol}(T)\sum_{i=1}^{m} w^i f(x^i), \quad f \in \mathbb{P}_n,$$

The nodes are given in barycentric coordinates. A quadrature rule of degree two with $m = 5$ is availabe. In this method as nodes the four vertices, with weights $1/120$, are used and the center of the tetrahedron (i.e. coordinates $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$) with weight $2/15$. Also a method of degree five with $m = 15$ has been implemented. The nodes and weights of this method are listed in table 3.1.

nodes	weights	
$(0.25, 0.25, 0.25, 0.25)$	$16/135$	
(A_1, A_1, A_1, B_1) (A_1, A_1, B_1, A_1) (A_1, B_1, A_1, A_1) (B_1, A_1, A_1, A_1)	$3(2665 + 14\sqrt{15})/112900$	$A_1 = (7 - \sqrt(15))/34$ $B_1 = (13 + 3\sqrt(15))/34$
(A_2, A_2, A_2, B_2) (A_2, A_2, B_2, A_2) (A_2, B_2, A_2, A_2) (B_2, A_2, A_2, A_2)	$3(2665 - 14\sqrt{15})/112900$	$A_2 = (7 + \sqrt(15))/34$ $B_2 = (13 - 3\sqrt(15))/34$
(A_3, A_3, B_3, B_3) (A_3, B_3, A_3, B_3) (A_3, B_3, B_3, A_3) (B_3, A_3, A_3, B_3) (B_3, A_3, B_3, A_3) (B_3, B_3, A_3, A_3)	$10/189$	$A_3 = (10 - 2\sqrt(15))/40$ $B_2 = (10 + 2\sqrt(15))/40$

Table 3.1: quadrature rule of degree 5

3.3 Time integration

Consider an initial value problem of the form

$$\frac{du}{dt} + F(u) = f(t), \qquad u(0) = u_0. \tag{3.6}$$

The Navier-Stokes system of DAEs in (3.5) takes this form if one elimates the pressure variable by restricting to the subspace of (discrete) divergence free velocities. We consider three time integration methods, namely the *theta-scheme*, the *fractional-step θ-scheme* and a *fractional-step method based on operator splitting*. These three methods are treated in the following subsections. As default in DROPS, for one-phase Navier-Stokes equations, the fractional-step θ-scheme is applied.

3.3.1 The θ-scheme

The θ-scheme applied to the initial value problem (3.6) is is given by

$$\frac{u^{\mathrm{new}} - u^{\mathrm{old}}}{\Delta t} + \theta F(u^{\mathrm{new}}) + (1-\theta)F(u^{\mathrm{old}}) = \theta f(t_{\mathrm{new}}) + (1-\theta)f(t_{\mathrm{old}}), \quad \theta \in [0,1].$$

For $\theta = 0$, we have the explicit Euler scheme, which is only first order accurate and not A-stable. The Crank-Nicholson scheme ($\theta = \frac{1}{2}$) is second order accurate and A-stable, but opposite to the fractional-step θ-scheme (cf. section 3.3.2) *not* strongly A-stable. When $\theta = 1$, we obtain the implicit Euler scheme, which is also first order accurate but strongly A-stable.

Application of this method to the Navier-Stokes DAE system results in

$$\begin{aligned} \mathbf{M}\tfrac{\vec{\mathbf{u}}^{n+1}-\vec{\mathbf{u}}^{n}}{\Delta t} + \theta[\mathbf{A}\vec{\mathbf{u}}^{n+1} + \mathbf{N}(\vec{\mathbf{u}}^{n+1})\vec{\mathbf{u}}^{n+1}] + \mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\ = \theta\vec{\mathbf{g}}^{n+1} - (1-\theta)[\mathbf{A}\vec{\mathbf{u}}^{n} + \mathbf{N}(\vec{\mathbf{u}}^{n})\vec{\mathbf{u}}^{n} - \vec{\mathbf{g}}^{n}] \\ \mathbf{B}\vec{\mathbf{u}}^{n+1} = 0. \end{aligned} \tag{3.7}$$

In each time step a nonlinear system of equations for the unknowns $\vec{\mathbf{u}}^{n+1}, \vec{\mathbf{p}}^{n+1}$ has to be solved. Iterative methods for solving this system are discussed in section 3.4.

3.3.2 fractional-step θ-scheme

For a given decomposition $F = F_1 + F_2$ and a given parameter $\alpha \in (0, \frac{1}{2})$, the fractional-step θ-scheme is based on a subdivision of each time interval $[n\Delta t, (n+1)\Delta t]$ in three subintervals with endpoints $(n+\alpha)\Delta t$, $(n+1-\alpha)\Delta t$, $(n+1)\Delta t$. For given u^n the approximations $u^{n+\alpha}$, $u^{n+1-\alpha}$, u^{n+1} at these endpoints are defined by

$$\frac{u^{n+\alpha} - u^n}{\alpha\Delta t} + F_1(u^{n+\alpha}) + F_2(u^n) = f^{n+\alpha} \tag{3.8}$$

$$\frac{u^{n+1-\alpha} - u^{n+\alpha}}{(1-2\alpha)\Delta t} + F_1(u^{n+\alpha}) + F_2(u^{n+1-\alpha}) = f^{n+1-\alpha} \tag{3.9}$$

$$\frac{u^{n+1} - u^{n+1-\alpha}}{\alpha\Delta t} + F_1(u^{n+1}) + F_2(u^{n+1-\alpha}) = f^{n+1}. \tag{3.10}$$

We use a popular variant of this scheme, cf. [74, 94], with $F = F_1 + F_2 := \theta F + (1-\theta)F$. This scheme is second order accurate and strongly A-stable for

$$\alpha := 1 - \frac{1}{2}\sqrt{2}, \quad \theta := \frac{1-2\alpha}{1-\alpha} = 2 - \sqrt{2}.$$

The Navier-Stokes DAE system (3.5) can be rewritten in a equivalent system of ODEs by eliminating the pressure (which can be interpreted as a Lagrange multiplier). Applying the fractional-step θ-scheme to this Navier-Stokes problem in ODE form and transforming it back to its original DAE form results in

$$\begin{cases} \mathbf{M}\tfrac{\vec{\mathbf{u}}^{n+\alpha}-\vec{\mathbf{u}}^{n}}{\alpha\Delta t} + \theta[\mathbf{A}\vec{\mathbf{u}}^{n+\alpha} + \mathbf{N}(\vec{\mathbf{u}}^{n+\alpha})\vec{\mathbf{u}}^{n+\alpha}] + \mathbf{B}^T\vec{\mathbf{p}}^{n+\alpha} \\ \qquad = \vec{\mathbf{g}}^{n+\alpha} - (1-\theta)[\mathbf{A}\vec{\mathbf{u}}^{n} + \mathbf{N}(\vec{\mathbf{u}}^{n})\vec{\mathbf{u}}^{n}] \\ \mathbf{B}\vec{\mathbf{u}}^{n+\alpha} = 0 \end{cases} \tag{3.11}$$

$$\begin{cases} \mathbf{M}\tfrac{\vec{\mathbf{u}}^{n+1-\alpha}-\vec{\mathbf{u}}^{n+\alpha}}{(1-2\alpha)\Delta t} + (1-\theta)[\mathbf{A}\vec{\mathbf{u}}^{n+1-\alpha} + \mathbf{N}(\vec{\mathbf{u}}^{n+1-\alpha})\vec{\mathbf{u}}^{n+1-\alpha}] + \mathbf{B}^T\vec{\mathbf{p}}^{n+1-\alpha} \\ \qquad = \vec{\mathbf{g}}^{n+1-\alpha} - \theta[\mathbf{A}\vec{\mathbf{u}}^{n+\alpha} + \mathbf{N}(\vec{\mathbf{u}}^{n+\alpha})\vec{\mathbf{u}}^{n+\alpha}] \\ \mathbf{B}\vec{\mathbf{u}}^{n+1-\alpha} = 0 \end{cases} \tag{3.12}$$

$$\begin{cases} \mathbf{M}\tfrac{\vec{\mathbf{u}}^{n+1}-\vec{\mathbf{u}}^{n+1-\alpha}}{\alpha\Delta t} + \theta[\mathbf{A}\vec{\mathbf{u}}^{n+1} + \mathbf{N}(\vec{\mathbf{u}}^{n+1})\vec{\mathbf{u}}^{n+1}] + \mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\ \qquad = \vec{\mathbf{g}}^{n+1} - (1-\theta)[\mathbf{A}\vec{\mathbf{u}}^{n+1-\alpha} + \mathbf{N}(\vec{\mathbf{u}}^{n+1-\alpha})\vec{\mathbf{u}}^{n+1-\alpha}] \\ \mathbf{B}\vec{\mathbf{u}}^{n+1} = 0 \end{cases} \tag{3.13}$$

Note that in this method we perform in each time interval $[n\Delta t,(n+1)\Delta t]$ three successive θ-scheme type substeps. The *non*linear problems for the pairs $(\vec{\mathbf{u}}^{n+\alpha},\vec{\mathbf{p}}^{n+\alpha})$, $(\vec{\mathbf{u}}^{n+1-\alpha},\vec{\mathbf{p}}^{n+1-\alpha})$, $(\vec{\mathbf{u}}^{n+1},\vec{\mathbf{p}}^{n+1})$ in these three substeps have a similar form.

3.3.3 Operator splitting

Another approach is introduced in [35] and further analyzed in [62]. This method is based on a splitting in the subspace of divergence free functions of the operator $F(\vec{\mathbf{u}})=\mathbf{M}^{-1}[\mathbf{A}\vec{\mathbf{u}}+\mathbf{N}(\vec{\mathbf{u}})\vec{\mathbf{u}}]$ into $F_1(\vec{\mathbf{u}})=\mathbf{M}^{-1}(\theta\mathbf{A}\vec{\mathbf{u}})$ and $F_2(\vec{\mathbf{u}})=\mathbf{M}^{-1}[(1-\theta)\mathbf{A}\vec{\mathbf{u}}+\mathbf{N}(\vec{\mathbf{u}})\vec{\mathbf{u}}]$. Application of this operator splitting method to the problem (3.5), results in the following method, cf. [35]:

$$\begin{cases} \mathbf{M}\frac{\vec{\mathbf{u}}^{n+\alpha}-\vec{\mathbf{u}}^{n}}{\alpha\Delta t}+\theta\mathbf{A}\vec{\mathbf{u}}^{n+\alpha}+\mathbf{B}^T\vec{\mathbf{p}}^{n+\alpha} \\ \qquad =\vec{\mathbf{g}}^{n+\alpha}-(1-\theta)\mathbf{A}\vec{\mathbf{u}^n}-\mathbf{N}(\vec{\mathbf{u}}^n)\vec{\mathbf{u}}^n \\ \mathbf{B}\vec{\mathbf{u}}^{n+\alpha}=0 \end{cases} \tag{3.14}$$

$$\begin{cases} \mathbf{M}\frac{\vec{\mathbf{u}}^{n+1-\alpha}-\vec{\mathbf{u}}^{n+\alpha}}{(1-2\alpha)\Delta t}+(1-\theta)\mathbf{A}\vec{\mathbf{u}}^{n+1-\alpha}+\mathbf{N}(\vec{\mathbf{u}}^{n+1-\alpha})\vec{\mathbf{u}}^{n+1-\alpha} \\ \qquad =\vec{\mathbf{g}}^{n+1-\alpha}-\theta\mathbf{A}\vec{\mathbf{u}}^{n+\alpha}-\mathbf{B}^T\vec{\mathbf{p}}^{n+\alpha} \end{cases} \tag{3.15}$$

$$\begin{cases} \mathbf{M}\frac{\vec{\mathbf{u}}^{n+1}-\vec{\mathbf{u}}^{n+1-\alpha}}{\alpha\Delta t}+\theta\mathbf{A}\vec{\mathbf{u}}^{n+1}+\mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\ \qquad =\vec{\mathbf{g}}^{n+1}-(1-\theta)\mathbf{A}\vec{\mathbf{u}}^{n+1-\alpha}-\mathbf{N}(\vec{\mathbf{u}}^{n+1-\alpha})\vec{\mathbf{u}}^{n+1-\alpha} \\ \mathbf{B}\vec{\mathbf{u}}^{n+1}=0 \end{cases} \tag{3.16}$$

An important property of this method is that the *nonlinearity and incompressibility condition in the Navier-Stokes equations are decoupled*. The problems in (3.14), (3.16) are *linear* Stokes type of equations and the problem in (3.15) consists of a nonlinear elliptic system for the velocity unknowns.

3.4 Linearization methods

Using an implicit time-stepping scheme we obtain a nonlinear system of algebraic equations in each time step. As an example, we consider the first step (3.11) in the fractional-step θ-scheme (note that the other two steps have a similar form). The nonlinear system is as follows:

$$\begin{cases} \big(\frac{1}{\alpha\Delta t}\mathbf{M}+\theta\mathbf{A}\big)\vec{\mathbf{u}}^{n+\alpha}+\theta\mathbf{N}(\vec{\mathbf{u}}^{n+\alpha})\vec{\mathbf{u}}^{n+\alpha}+\mathbf{B}^T\vec{\mathbf{p}}^{n+\alpha} \\ \qquad =\vec{\mathbf{g}}+\big(\frac{1}{\alpha\Delta t}\mathbf{M}-(1-\theta)(\mathbf{A}+\mathbf{N}(\vec{\mathbf{u}}^n)\big)\vec{\mathbf{u}}^n \\ \mathbf{B}\vec{\mathbf{u}}^{n+\alpha}=0. \end{cases} \tag{3.17}$$

This nonlinear system has the form

$$\begin{aligned} \tilde{\mathbf{A}}\mathbf{x}+\mathbf{N}(\mathbf{x})\mathbf{x}+\mathbf{B}^T\mathbf{y}&=\mathbf{b} \\ \mathbf{B}\mathbf{x}&=\mathbf{c} \end{aligned} \tag{3.18}$$

and is solved by a fixed point defect correction method with step size control taken from [94].

Algorithm 1 (fixed point defect correction method with step size control)
Set $\omega^0=1$.
Repeat until desired accuracy:

1. *Calculate the defects*

$$\begin{pmatrix}\mathbf{d}_1\\ \mathbf{d}_2\end{pmatrix} = \begin{pmatrix}\tilde{\mathbf{A}}\mathbf{x}^k + \mathbf{N}(\mathbf{x}^k)\mathbf{x}^k + \mathbf{B}^T\mathbf{y}^k - \mathbf{b}\\ \mathbf{B}\mathbf{x}^k - \mathbf{c}\end{pmatrix}$$

2. *Solve the a discrete Oseen system for the corrections* $\mathbf{v}$ *and* $\mathbf{q}$

$$\begin{aligned}[\tilde{\mathbf{A}} + \mathbf{N}(\mathbf{x}^k)]\mathbf{v} + \mathbf{B}^T\mathbf{q} &= \mathbf{d}_1\\ \mathbf{B}\mathbf{v} &= \mathbf{d}_2\end{aligned}$$

with accuracy $\mathrm{tol}^{\mathrm{k}}$.

3. *Step size control: Calculate the step length parameter*

$$\omega^{k+1} := \frac{\left\langle \mathbf{K}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix},\quad \mathbf{K}\begin{pmatrix}\mathbf{x}^k\\ \mathbf{y}^k\end{pmatrix} - \begin{pmatrix}\mathbf{b}\\ \mathbf{c}\end{pmatrix}\right\rangle}{\left\langle \mathbf{K}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix},\quad \mathbf{K}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix}\right\rangle} \tag{3.19}$$

with

$$\mathbf{K} := \begin{pmatrix}\tilde{\mathbf{A}} + \mathbf{N}(\mathbf{x}^k - \omega^k\mathbf{v}) & \mathbf{B}^T\\ \mathbf{B} & 0\end{pmatrix}$$

4. *Update* $\mathbf{x}^k$, $\mathbf{y}^k$

$$\begin{pmatrix}\mathbf{x}^{k+1}\\ \mathbf{y}^{k+1}\end{pmatrix} = \begin{pmatrix}\mathbf{x}^k\\ \mathbf{y}^k\end{pmatrix} - \omega^{k+1}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix}$$

The discrete Oseen system in step 2 can be solved using iterative solvers that are treated in section 3.5.3.

3.5 Iterative solvers for large sparse linear systems

In this section we discuss iterative solvers available in DROPS that can be used for solving the large sparse *linear* systems that arise after discretization and linearization of (Navier-)Stokes equations.

3.5.1 Iterative solvers for discretized scalar elliptic problems

Let $\mathbf{Ax} = \mathbf{b}$ be a large sparse linear system that results from the discretization of a scalar elliptic problem, for example, a Poisson equation or a convection-diffusion equation. Basic iterative methods for such a problem are:

- Jacobi method, Gauss-Seidel method.
- Successive overrelaxation method (SOR) and its symmetric variant (SSOR).
- Conjugate gradient method (CG).
- Krylov subspace methods for nonsymmetric problems: GMRES, BiCGSTAB, GCR.

For a description of these methods we refer to the literature, e.g. [54, 79]. These methods, which are easy to implement, are available in DROPS. However, often their rate of convergence is (very) low. A much more efficient iterative solver can be obtained by using basic iterative methods like a (damped) Jacobi or Gauss-Seidel method in a *multilevel* approach. Since in our discretization technique we have a hierarchy of nested grids and corresponding finite element spaces available such a multigrid approach can be used. A *multigrid iterative solver* with a damped Jacobi or Gauss-Seidel smoother and canonical intergrid transfer operators (induced by the nestedness of the finite element spaces) has been implemented in DROPS. This method is a very efficient solver for scalar diffusion and convection-diffusion problems.

3.5.2 Iterative solvers for discretized Stokes equations

The discrete Stokes problem has a matrix-vector representation of the form

$$\mathbf{K}\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} := \begin{pmatrix}\tilde{\mathbf{A}} & \mathbf{B}^T\\ \mathbf{B} & 0\end{pmatrix}\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} = \begin{pmatrix}\mathbf{f}_1\\ \mathbf{f}_2\end{pmatrix}\ , \quad \tilde{\mathbf{A}} := \mathbf{A} + \beta\mathbf{M}\ , \tag{3.20}$$

where the parameter β is proportional to $1/\Delta t$. The matrix $\tilde{\mathbf{A}}$ is symmetric positive definite. The matrix $\mathbf{K}$ is symmetric and strongly indefinite and has a *saddle point structure*. For this type of linear systems we consider the following methods:

- inexact Uzawa methods,
- preconditioned MINRES,
- multigrid method.

We briefly address these three classes of methods.

Inexact Uzawa method

The Schur complement of the matrix $\mathbf{K}$ is given by $\mathbf{S} = \mathbf{B}\tilde{\mathbf{A}}^{-1}\mathbf{B}^T$. The matrix $\mathbf{K}$ has a block factorization

$$\mathbf{K} = \begin{pmatrix}\tilde{\mathbf{A}} & 0\\ \mathbf{B} & -\mathbf{I}\end{pmatrix}\begin{pmatrix}\mathbf{I} & \tilde{\mathbf{A}}^{-1}\mathbf{B}^T\\ 0 & \mathbf{S}\end{pmatrix}.$$

Solving the problem $\mathbf{K}\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} = \begin{pmatrix}\mathbf{f}_1\\ \mathbf{f}_2\end{pmatrix}$ by block forward-backward substitution yields the equivalent problem:

1. Solve $\tilde{\mathbf{A}}\mathbf{z} = \mathbf{f}_1$. (3.21)
2. Solve $\mathbf{S}\mathbf{y} = \mathbf{B}\mathbf{z} - \mathbf{f}_2, \quad \mathbf{y} \perp \mathbf{M}(1,\dots,1)^T$. (3.22)
3. Solve $\tilde{\mathbf{A}}\mathbf{x} = \mathbf{z} - \mathbf{B}^T\mathbf{y}$. (3.23)

In the Uzawa method one applies an iterative solver (e.g., CG) to the Schur complement system in step 2. Note that the matrix $\mathbf{S}$ in this system is symmetric positive (semi)definite. The $\tilde{\mathbf{A}}$-systems that occur in each iteration of this method and in the steps 1 and 3 are solved sufficiently accurate using some fast Poisson solver.
We consider a simple variant of this method in which the exact solves of the $\tilde{\mathbf{A}}$-systems are replaced by inexact ones. Let $\mathbf{Q}_A$ be a preconditioner of $\tilde{\mathbf{A}}$. We use this preconditioner in the steps 1 and 3 and also for the approximation of the Schur complement is step 2. For this we introduce the notation

$$\hat{\mathbf{S}} := \mathbf{B}\mathbf{Q}_A^{-1}\mathbf{B}^T. \tag{3.24}$$

We use a (nonlinear) approximate inverse of $\hat{\mathbf{S}}$ denoted by Ψ. For each $\mathbf{w}$, $\Psi(\mathbf{w})$ is an approximation to the solution $\mathbf{z}^*$ of $\hat{\mathbf{S}}\mathbf{z} = \mathbf{w}$. We assume that

$$\|\Psi(\mathbf{w}) - \mathbf{z}^*\|_{\hat{S}} \leq \delta \|\mathbf{z}^*\|_{\hat{S}} \quad \text{for all} \quad \mathbf{w} \tag{3.25}$$

holds with a given tolerance parameter $\delta < 1$. In our implementation Ψ is the (preconditioned) CG method. Let $(\mathbf{x}^k, \mathbf{y}^k)$ be a given approximation to the solution $(\mathbf{x}, \mathbf{y})$. Note that using the block factorization of $\mathbf{K}$ we get

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^k \\ \mathbf{y}^k \end{pmatrix} + \begin{pmatrix} \mathbf{I} & -\tilde{\mathbf{A}}^{-1}\mathbf{B}^T\mathbf{S}^{-1} \\ 0 & \mathbf{S}^{-1} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{A}}^{-1} & 0 \\ \mathbf{B}\tilde{\mathbf{A}}^{-1} & -\mathbf{I} \end{pmatrix} \Big(\begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix} - \mathbf{K} \begin{pmatrix} \mathbf{x}^k \\ \mathbf{y}^k \end{pmatrix} \Big). \tag{3.26}$$

With $\tilde{\mathbf{A}}^{-1} \approx \mathbf{Q}_A^{-1}$, $\mathbf{S}^{-1}\mathbf{w} \approx \hat{\mathbf{S}}^{-1}\mathbf{w} \approx \Psi(\mathbf{w})$ and $\mathbf{r}_1^k := \mathbf{f}_1 - \tilde{\mathbf{A}}\mathbf{x}^k - \mathbf{B}^T\mathbf{y}^k$ we obtain the (nonlinear) iterative method

$$\begin{aligned} \mathbf{x}^{k+1} &= \mathbf{x}^k + \mathbf{Q}_A^{-1}\mathbf{r}_1^k - \mathbf{Q}_A^{-1}\mathbf{B}^T\Psi\big(\mathbf{B}(\mathbf{Q}_A^{-1}\mathbf{r}_1^k + \mathbf{x}^k) - \mathbf{f}_2\big) \\ \mathbf{y}^{k+1} &= \mathbf{y}^k + \Psi\big(\mathbf{B}(\mathbf{Q}_A^{-1}\mathbf{r}_1^k + \mathbf{x}^k) - \mathbf{f}_2\big) \end{aligned} \tag{3.27}$$

Thus we obtain an inexact Uzawa method with the following algorithmic structure:

$$\left\{ \begin{array}{l} \begin{pmatrix} \mathbf{x}^0 \\ \mathbf{y}^0 \end{pmatrix} \text{ a given starting vector; } \mathbf{r}_1^0 := \mathbf{f}_1 - \tilde{\mathbf{A}}\mathbf{x}^0 - \mathbf{B}^T\mathbf{y}^0 \\ \text{for } k \geq 0 \ : \\ \quad \mathbf{w} := \mathbf{x}^k + \mathbf{Q}_A^{-1}\mathbf{r}_1^k \\ \quad \mathbf{z} := \Psi(\mathbf{B}\mathbf{w} - \mathbf{f}_2) \\ \quad \mathbf{x}^{k+1} := \mathbf{w} - \mathbf{Q}_A^{-1}\mathbf{B}^T\mathbf{z} \\ \quad \mathbf{y}^{k+1} := \mathbf{y}^k + \mathbf{z} \\ \quad \mathbf{r}_1^{k+1} := \mathbf{r}_1^k - \tilde{\mathbf{A}}(\mathbf{x}^{k+1} - \mathbf{x}^k) - \mathbf{B}^T\mathbf{z}. \end{array} \right. \tag{3.28}$$

Here $\Psi(\cdot)$ is as follows:

$$\Psi(\mathbf{B}\mathbf{w} - \mathbf{f}_2) = \begin{cases} \text{Result of } \ell \text{ PCG iterations with starting vector } 0 \\ \text{and preconditioner } \mathbf{Q}_S \text{ applied to } \hat{\mathbf{S}}\mathbf{v} = \mathbf{B}\mathbf{w} - \mathbf{f}_2. \end{cases} \tag{3.29}$$

This algorithm consists of an inner-outer iteration. An analysis of this method is given in [10] where it is shown that in the inner iteration (3.29) one should use very small ℓ-values ($\ell = 1, 2$) and that this inexact Uzawa method is an efficient solver for the saddle point problem (3.20) *provided*

$$\textit{we have good preconditioners } \mathbf{Q}_A \textit{ of } \tilde{\mathbf{A}} \textit{ and } \mathbf{Q}_S \textit{ of } \mathbf{S}. \tag{3.30}$$

For the preconditioner $\mathbf{Q}_A$ we use a *standard symmetric multigrid V-cycle* (Gauss-Seidel smoother). An appropriate preconditioner $\mathbf{Q}_S$ is due to Cahouet and Chabard [36]. We briefly explain this method. For further information and a convergence analysis we refer to the literature [10]. For $g \in L^2(\Omega)$ consider the Neumann problem: find $w \in H^1(\Omega) \cap Q$ such that

$$(\nabla w, \nabla \phi) = (g, \phi) \quad \text{for all} \quad \phi \in H^1(\Omega) \cap Q. \tag{3.31}$$

Let $\mathbf{T}_h$ be the stiffness matrix of the Galerkin discretization of this problem in $Q_h \subset H^1(\Omega)$:

$$\langle \mathbf{T}_h\mathbf{x}, \mathbf{y} \rangle = (\nabla J_Q\mathbf{x}, \nabla J_Q\mathbf{y}) \quad \text{for all} \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^K$$

with $J_Q : \mathbb{R}^K \to Q_h$ the finite element isomorphism between finite element functions in Q_h and their nodal values, cf. (3.4). Note that $\ker(\mathbf{T}_h) = \operatorname{span}(\mathbf{e})$, with $\mathbf{e} := (1, 1, \ldots, 1)^T$. We define $\tilde{\mathbf{Q}}_S^{-1} : \mathbf{e}^\perp \to \mathbb{R}^K$ by:

$$\tilde{\mathbf{Q}}_S^{-1} = \begin{cases} \mathbf{M}^{-1} + \beta \mathbf{T}_h^{-1}, & \text{if} \quad \beta \leq h^{-2}, \\ \beta h^2 \mathbf{M}^{-1} + \beta \mathbf{T}_h^{-1}, & \text{if} \quad h^{-2} \leq \beta. \end{cases} \tag{3.32}$$

Note, that for $\beta = 0$ (which corresponds to the discrete *stationary* Stokes equation) we get $\tilde{\mathbf{Q}}_S = \mathbf{M}$. In [34] it is shown that $\tilde{\mathbf{Q}}_S$ is uniformly in h and β spectrally equivalent to the Schur complement $\mathbf{S}$.

Preconditioned MINRES method

We consider a preconditioned minimal residual (PMINRES) method for solving the discretized Stokes problem (3.20). This class of methods has been analyzed in [73, 78, 85]. We consider a symmetric positive definite preconditioner

$$\tilde{\mathbf{K}} = \begin{pmatrix} \mathbf{Q}_A & 0 \\ 0 & \mathbf{Q}_S \end{pmatrix} \tag{3.33}$$

for $\mathbf{K}$. Define the norm $\|\mathbf{v}\|_{\tilde{K}} := \langle \tilde{\mathbf{K}}\mathbf{v}, \mathbf{v} \rangle^{\frac{1}{2}}$ for $\mathbf{v} \in \mathbb{R}^{K+N}$. Given a starting vector $\mathbf{v}^0$ with corresponding error $\mathbf{e}^0 := \mathbf{v}^* - \mathbf{v}^0$, then in the preconditioned MINRES method one computes the vector $\mathbf{v}^k \in \mathbf{v}^0 + \operatorname{span}\{\tilde{\mathbf{K}}^{-1}\mathbf{K}\mathbf{e}^0, \ldots, (\tilde{\mathbf{K}}^{-1}\mathbf{K})^k \mathbf{e}^0\}$ which minimizes the preconditioned residual $\|\tilde{\mathbf{K}}^{-1}\mathbf{K}(\mathbf{v}^* - \mathbf{v})\|_{\tilde{\mathbf{K}}}$ over this subspace.
An efficient implementation of this method can be derived using the Lanczos algorithm and Givens rotations. For such an implementation we refer to the literature, e.g. [69, 51]. In an efficient implementation of this method one needs per iteration one evaluation of $\mathbf{Q}_A^{-1}$, one evaluation of $\mathbf{Q}_S^{-1}$ and one matrix-vector product with $\mathbf{K}$. From a convergence analysis of this method it follows that we have fast convergence if $\mathbf{Q}_A$ and $\mathbf{Q}_S$ are good preconditioners of $\tilde{\mathbf{A}}$ and $\mathbf{S}$, respectively. For these preconditioners one can take the ones discussed above. A detailed discussion of this method and a comparison with the inexact Uzawa method are given in [10].

Multigrid method

Multigrid methods can be applied directly to the (generalized) Stokes problem (3.20). These so-called coupled multigrid methods are based on a combination of a *smoother* that is applied to the saddle point system (3.20) and *coarse-grid corrections* that are obtained from discretizations of the form (3.20) on coarser grids. For a treatment of these multigrid methods we refer to the literature, eg. [33, 53]. In DROPS two types of smoothers are available, namely a Braess-Sarazin method and a Vanka smoother. An explanation of these smoothers and of the corresponding multigrid solver is given in [5]. In that paper results of numerical methods are given in which this multigrid method is compared to the preconditioned MINRES and the inexact Uzawa method. These results show that for the class of generalized discrete Stokes equations as in (3.20) all three methods are robust with respect to variation in the parameters h and β and often these three methods have a comparable efficiency.

3.5.3 Iterative solvers for discretized Oseen equations

The discrete Oseen problem has a matrix-vector representation of the form

$$\mathbf{K}\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} := \begin{pmatrix} \hat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{B} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix} \ , \quad \hat{\mathbf{A}} := \mathbf{A} + \mathbf{N}(\mathbf{x}^{\text{old}}) + \beta \mathbf{M} \ , \tag{3.34}$$

where the parameter β is proportional to $1/\Delta t$, cf. (3.17). This linear system has a saddle point structure but opposite to the discrete Stokes equation in (3.20) the system matrix $\mathbf{K}$ in this case is *nonsymmetric*. For this type of linear systems, methods similar to the ones used for the Stokes problem in section 3.5.2 can be used. We again distinguish three classes of methods: inexact Uzawa method, preconditioned Krylov-subspace solvers and multigrid methods. We briefly address these classes of methods.

Inexact Uzawa method

The inexact Uzawa method has an algorithmic structure as in (3.28). For the Oseen problem the Schur complement is *non*symmetric and therefore we do not apply a CG method in (3.29) but for Ψ we use a (preconditioned) GMRES or GCR method. Also the preconditioners have to be modified. For $\mathbf{Q}_A$ one can apply a Jacobi- or Gauss-Seidel iteration, a multigrid method or a (preconditioned) Krylov-subspace iteration. More delicate is the choice of $\mathbf{Q}_S$. One possibility is given in (3.36) below.

Preconditioned Krylov-subspace methods

One can apply a Krylov-subspace method directly to the saddle point system (3.34) and combine this with a block preconditioner $\tilde{\mathbf{K}}$ of the form (3.33). It is convenient to allow for a preconditioner that varies per iteration. Such a *variable* preconditioner arises if for the preconditioning of $\hat{\mathbf{A}}$ or of the Schur complement one uses an *inner* Krylov-subspace iteration. Standard methods like GMRES or BiCGSTAB do not allow such variable preconditioners. There are, however, Krylov-subspace methods that can handle such variable preconditioners. In DROPS two of such methods are available namely GCR (Generalized Conjugate Residuals, cf. [79]) and FGMRES (Flexible GMRES, cf. [79]).

In our applications we often use GCR with a block-preconditioner introduced in [42] of the form

$$\begin{pmatrix} \mathbf{Q}_A & \mathbf{B}^T \\ 0 & -\mathbf{Q}_S \end{pmatrix}. \tag{3.35}$$

The application of $\mathbf{Q}_A^{-1}$ to a vector $\mathbf{b}$ is defined by a Jacobi-preconditioned BICGSTAB solver with the stopping criterion

$$\frac{\|\mathbf{r}^k\|}{\|\mathbf{r}^0\|} \leq tol$$

applied to the linear system $\hat{\mathbf{A}}\mathbf{x} = \mathbf{b}$. For the Schur complement preconditioner $\mathbf{Q}_S^{-1}$ we use the diagonally scaled preconditioner from [42], i. e.

$$\mathbf{Q}_S^{-1} = (\mathbf{B}\,\mathbf{M}_1^{-1}\,\mathbf{B}^T)^{-1}\,\mathbf{B}\,\mathbf{M}_1^{-1}\,\hat{\mathbf{A}}\,\mathbf{M}_1^{-1}\,\mathbf{B}^T\,(\mathbf{B}\,\mathbf{M}_1^{-1}\,\mathbf{B}^T)^{-1} \tag{3.36}$$

where $\mathbf{M}_1$ is the diagonal of the matrix $\mathbf{M}$. The inverse of $(\mathbf{B}^T\,\mathbf{M}_1^{-1}\,\mathbf{B})$ is approximated by using a CG method with the same stopping criterion as of $\mathbf{Q}_A^{-1}$.

Multigrid method

The multigrid method used for the generalized Stokes problem, cf. section 3.5.2 and [5], can be used without changes for the Oseen problem, provided the underlying flow problem is not convection-dominated (i.e., a relatively low Reynolds number). For problems with strong convection, this multigrid method has to be modified. Such multigrid solvers for convection-dominated flows are not available in DROPS, yet.

Chapter 4

Navier-Stokes equations for two-phase flow (NS2)

4.1 Weak formulation

In this section we present a weak formulation of the two-phase flow model (1.4). For simplicity we use homogeneous Dirichlet boundary conditions for $\mathbf{u}$. We use the same spaces $\mathbf{V}$. Q as for the one-phase Navier-Stokes problems, cf. (3.1). We also use $V := H^1(\Omega)$. We define the bilinear forms

$$\begin{aligned} m : L^2(\Omega)^3 \times L^2(\Omega)^3 \to \mathbb{R} \ : \quad & m(\mathbf{u},\mathbf{v}) = \int_\Omega \rho(\phi)\, \mathbf{u}\cdot\mathbf{v}\, dx \\ a : \mathbf{V}\times\mathbf{V} \to \mathbb{R} \ : \quad & a(\mathbf{u},\mathbf{v}) = \frac{1}{2}\int_\Omega \mu(\phi)\ \mathrm{tr}\left(\mathbf{D}(\mathbf{u})\mathbf{D}(\mathbf{v})\right)\, dx \\ b : \mathbf{V}\times Q \to \mathbb{R} \ : \quad & b(\mathbf{u},q) = \int_\Omega q\, \mathrm{div}\,\mathbf{u}\, dx \end{aligned}$$

and the trilinear form

$$c : \mathbf{V}\times\mathbf{V}\times\mathbf{V} \to \mathbb{R} \ : \quad c(\mathbf{u};\mathbf{v},\mathbf{w}) = \int_\Omega \rho(\phi)\,(\mathbf{u}\cdot\nabla\mathbf{v})\cdot\mathbf{w}\, dx.$$

Note, that *the bilinear forms a and m as well as the trilinear form c depend on ϕ.* A weak formulation of the problem (1.4) is as follows:

Determine $\mathbf{u}(t)\in\mathbf{V}, p(t)\in Q$ and $\phi(t)\in V$ such that

$$\begin{aligned} m(\mathbf{u}_t(t),\mathbf{v}) + a(\mathbf{u}(t),\mathbf{v}) + c(\mathbf{u}(t);\mathbf{u}(t),\mathbf{v}) - b(\mathbf{v},p(t)) &= m(\mathbf{g},\mathbf{v}) + f_\Gamma(\mathbf{v}) \\ b(\mathbf{u}(t),q) &= 0 \\ (\phi_t(t),v)_0 + (\mathbf{u}(t)\cdot\nabla\phi(t),v)_0 &= 0 \end{aligned} \tag{4.1}$$

with

$$f_\Gamma(\mathbf{v}) = \int_\Omega \left(\tau\mathcal{K}\delta_\Gamma\mathbf{n}_\Gamma - \nabla_\Gamma\tau\delta_\Gamma\right)\cdot\mathbf{v}\, dx = \int_\Gamma \left(\tau\mathcal{K}\mathbf{n}_\Gamma - \nabla_\Gamma\tau\right)\cdot\mathbf{v}\, ds. \tag{4.2}$$

Here δ_Γ denotes a Dirac distribution and ∇_Γ denotes the tangential gradient as defined in (4.12) below. Note that the functional f_Γ is Γ-dependent and therefore also ϕ-dependent.

4.2 Spatial discretization

4.2.1 Galerkin finite element discretization

Let $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ be a multilevel triangulation of Ω. With each triangulation $\mathcal{T}_k$ $(0 \le k \le J)$ we associate a mesh size parameter $h = h_k$. Let $\mathbf{V}_h \subset \mathbf{V}$, $Q_h \subset L_0^2(\Omega)$ and $V_h \subset V$ be standard polynomial finite element spaces corresponding to the triangulation $\mathcal{T}_k$. For $\mathbf{V}_k$ and V_h we use *piecewise quadratics*. The choice of the pressure finite element space is explained in section 4.2.4.

The Galerkin discretization leads to the following variational problem: Find $\mathbf{u}_h(t) \in \mathbf{V}_h$, $p_h(t) \in Q_h$ and $\phi_h(t) \in V_h$ such that for $t \in [0, T]$:

$$\begin{aligned} m((\mathbf{u}_h)_t(t), \mathbf{v}_h) + a(\mathbf{u}_h(t), \mathbf{v}_h) + c(\mathbf{u}_h(t); \mathbf{u}_h(t), \mathbf{v}_h) - b(\mathbf{v}_h, p_h(t)) & \\ = \quad m(\mathbf{g}, \mathbf{v}_h) + f_{\Gamma_h}(\mathbf{v}_h) \quad \forall \mathbf{v}_h \in \mathbf{V}_h & \quad (4.3) \\ b(\mathbf{u}_h(t), q_h) \quad = \quad 0 \qquad\qquad \forall q_h \in Q_h & \quad (4.4) \\ ((\phi_h)_t(t), v_h)_0 + (\mathbf{u}_h(t) \cdot \nabla \phi_h(t), v_h)_0 \quad = \quad 0 \qquad\qquad \forall v_h \in V_h. & \quad (4.5) \end{aligned}$$

The term $f_{\Gamma_h}(\mathbf{v}_h)$ is an approximation of $f_\Gamma(\mathbf{v}_h)$ with Γ_h being an approximation of the interface Γ. These approximation issues are discussed in sections 4.2.2 and 4.2.3. To make this semi-discrete problem well-posed we need initial conditions $\mathbf{u}_h(0)$ and $\phi_h(0)$.

The discretization (4.5) of the linear hyperbolic level set equation is not stable. It can be stabilized using a standard streamline diffusion technique. This streamline diffusion stabilization applied to the level set equation can be seen as a Petrov-Galerkin method, with trial space V_h and test functions $\hat{v}_h$. For each tetrahedron $T \in \mathcal{T}_k$ a stabilization parameter δ_T is chosen. The test functions are then defined as

$$\hat{v}_h(x) = v_h(x) + \delta_T \mathbf{u}_h(x,t) \cdot \nabla v_h(x), \quad x \in T.$$

For an analysis of the streamline diffusion method and reasonable choices for the stabilization parameter δ_T we refer to [76]. This leads to the following stabilized variant of (4.5):

$$\sum_{T \in \mathcal{T}_k} ((\phi_h)_t(t) + \mathbf{u}_h(t) \cdot \nabla \phi_h(t), v_h + \delta_T \mathbf{u}_h(t) \cdot \nabla v_h)_T = 0 \qquad \forall v_h \in V_h. \quad (4.6)$$

Here $(\cdot,\cdot)_T$ denotes the L^2-scalar product over the domain T. Let $\{\boldsymbol{\xi}_j\}_{1 \le j \le N}$, $\{\psi_j\}_{1 \le j \le K}$ and $\{\chi_j\}_{1 \le j \le L}$ be (nodal) bases of $\mathbf{V}_h$, Q_h and V_h, respectively. These bases induce corresponding representations of the finite element functions in vector form. Functions $\mathbf{u}_h(t) \in \mathbf{V}_h$, $p_h(t) \in Q_h$ and $\phi_h(t) \in V_h$ can be represented as:

$$\begin{aligned} \mathbf{u}_h(t) &= \sum_{j=1}^{N} u_j(t) \boldsymbol{\xi}_j, \quad \vec{\mathbf{u}}(t) := (u_1(t), \ldots, u_N(t)) \\ p_h(t) &= \sum_{j=1}^{K} p_j(t) \psi_j, \quad \vec{\mathbf{p}}(t) := (p_1(t), \ldots, p_K(t)) \\ \phi_h(t) &= \sum_{j=1}^{L} \phi_j(t) \chi_j, \quad \vec{\boldsymbol{\phi}}(t) := (\phi_1(t), \ldots, \phi_L(t)). \end{aligned}$$

For $\phi_h \in V_h$ and $\mathbf{u}_h \in \mathbf{V}_h$ we introduce the following (mass and stiffness) matrices:

$$\begin{aligned}
\mathbf{M}(\phi_h) \in \mathbb{R}^{N\times N}, &\quad \mathbf{M}(\phi_h)_{ij} = \int_\Omega \rho(\phi_h)\boldsymbol{\xi}_i \cdot \boldsymbol{\xi}_j \, dx \\
\mathbf{A}(\phi_h) \in \mathbb{R}^{N\times N}, &\quad \mathbf{A}(\phi_h)_{ij} = \frac{1}{2}\int_\Omega \mu(\phi_h) \operatorname{tr}\left(\mathbf{D}(\boldsymbol{\xi}_i)\mathbf{D}(\boldsymbol{\xi}_j)\right) \, dx \\
\mathbf{B} \in \mathbb{R}^{K\times N}, &\quad \mathbf{B}_{ij} = -\int_\Omega \psi_i \operatorname{div} \boldsymbol{\xi}_j \, dx \\
\mathbf{N}(\phi_h, \mathbf{u}_h) \in \mathbb{R}^{N\times N}, &\quad \mathbf{N}(\phi_h, \mathbf{u}_h)_{ij} = \int_\Omega \rho(\phi_h)(\mathbf{u}_h \cdot \nabla\boldsymbol{\xi}_j)\cdot\boldsymbol{\xi}_i \, dx \\
\mathbf{E}(\mathbf{u}_h) \in \mathbb{R}^{L\times L}, &\quad \mathbf{E}(\mathbf{u}_h)_{ij} = \sum_{T\in\mathcal{T}_k}\int_T \chi_j(\chi_i + \delta_T \mathbf{u}_h\cdot\nabla\chi_i) \, dx \\
\mathbf{H}(\mathbf{u}_h) \in \mathbb{R}^{L\times L}, &\quad \mathbf{H}(\mathbf{u}_h)_{ij} = \sum_{T\in\mathcal{T}_k}\int_T (\mathbf{u}_h\cdot\nabla\chi_j)(\chi_i + \delta_T \mathbf{u}_h\cdot\nabla\chi_i) \, dx.
\end{aligned}$$

We also need the following vectors:

$$\begin{aligned}
\vec{\mathbf{g}}(\phi_h) \in \mathbb{R}^N, &\quad \vec{\mathbf{g}}(\phi_h)_i = \int_\Omega \rho(\phi_h)\mathbf{g}\cdot\boldsymbol{\xi}_i \, dx \\
\vec{\mathbf{f}}_{\Gamma_h}(\phi_h) \in \mathbb{R}^N, &\quad \vec{\mathbf{f}}_{\Gamma_h}(\phi_h)_i = f_{\Gamma_h}(\boldsymbol{\xi}_i).
\end{aligned}$$

Using this notation we obtain the following equivalent formulation of the coupled system of ordinary differential equations (4.3), (4.4), (4.6): Find $\vec{\mathbf{u}}(t) \in \mathbb{R}^N$, $\vec{\mathbf{p}}(t) \in \mathbb{R}^K$ and $\vec{\boldsymbol{\phi}}(t) \in \mathbb{R}^L$ such that for all $t \in [0,T]$

$$\begin{aligned}
\mathbf{M}(\vec{\boldsymbol{\phi}}(t))\frac{d\vec{\mathbf{u}}}{dt}(t) + \mathbf{A}(\vec{\boldsymbol{\phi}}(t))\vec{\mathbf{u}}(t) + \mathbf{N}(\vec{\boldsymbol{\phi}}(t), \vec{\mathbf{u}}(t))\vec{\mathbf{u}}(t) + \mathbf{B}^T\vec{\mathbf{p}}(t) & \\
= \vec{\mathbf{g}}(\vec{\boldsymbol{\phi}}(t)) + \vec{\mathbf{f}}_{\Gamma_h}(\vec{\boldsymbol{\phi}}(t)) & \quad (4.7)\\
\mathbf{B}\vec{\mathbf{u}}(t) = 0 & \quad (4.8)\\
\mathbf{E}(\vec{\mathbf{u}}(t))\frac{d\vec{\boldsymbol{\phi}}}{dt}(t) + \mathbf{H}(\vec{\mathbf{u}}(t))\vec{\boldsymbol{\phi}}(t) = 0. & \quad (4.9)
\end{aligned}$$

In addition we have initial conditions for $\vec{\mathbf{u}}$ and $\vec{\boldsymbol{\phi}}$.

4.2.2 Discrete approximation of the interface: $\Gamma \rightsquigarrow \Gamma_h$

In this section we explain how a polyhedral approximation Γ_h of Γ is constructed.

The level set equation for ϕ (signed distance function) is discretized with continuous piecewise quadratic finite elements on the tetrahedral triangulation $\mathcal{T}_h$. The resulting piecewise *quadratic* finite element approximation of ϕ on $\mathcal{T}_h$ is denoted by $\phi_h = \phi_h(x,t)$. We assume a given fixed time t. We introduce one further regular refinement of $\mathcal{T}_h$, resulting in $\mathcal{T}_h'$. Let $I(\phi_h)$ be the continuous piecewise *linear* function on $\mathcal{T}_h'$ which interpolates ϕ_h at all vertices of all tetrahedra in $\mathcal{T}_h'$. The approximation of the interface Γ is defined by

$$\Gamma_h := \{\, x \in \Omega \mid I(\phi_h)(x) = 0 \,\}. \quad (4.10)$$

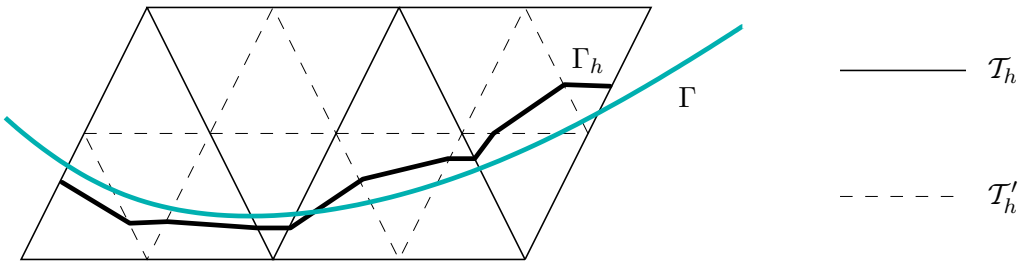

Figure 4.1: Construction of approximate interface for 2D case.

and consists of piecewise planar segments. The mesh size parameter h is the maximal diameter of these segments. This maximal diameter is approximately the maximal diameter of the tetrahedra in $\mathcal{T}_h'$ that contain the discrete interface, i.e., $h = h_\Gamma$ is approximately the maximal diameter of the tetrahedra in $\mathcal{T}_h'$ *that are close to the interface*. In Figure 4.1 we illustrate this construction for the two-dimensional case.

Each of the planar segments of Γ_h is either a triangle or a quadrilateral. The quadrilaterals can (formally) be divided into two triangles. Thus Γ_h consists of a set of triangular faces. Concerning the (maximal) distance between Γ and Γ_h we note the following. If we assume $|I(\phi_h)(x) - \phi(x)| \leq c\, h_\Gamma^2$ for all x in a neighbourhood of Γ, which is reasonable for a smooth ϕ and piecewise quadratic ϕ_h, then we have

$$\operatorname{dist}(\Gamma, \Gamma_h) = \max_{x \in \Gamma_h} |\phi(x)| = \max_{x \in \Gamma_h} |\phi(x) - I(\phi_h)(x)| \leq c\, h_\Gamma^2.$$

4.2.3 Discretization of the curvature localized force term

We now discuss the treatment of the surface tension term. In the weak formulation this localized force is represented by the linear functional f_Γ as in (4.2). If we assume that the surface tension coefficient τ is *constant* this functional takes the form

$$f_\Gamma(\mathbf{v}) := \tau \int_\Omega \mathcal{K} \delta_\Gamma \mathbf{n}_\Gamma \cdot \mathbf{v}\, dx = \tau \int_\Gamma \mathcal{K} \mathbf{n}_\Gamma \cdot \mathbf{v}\, ds \tag{4.11}$$

with $\mathbf{v}$ an element from the space of testfunctions ($H_0^1(\Omega)^3$ in our case). The case with a *variable* surface tension coefficient is discussed in Remark 3. Note that f_Γ is Γ-dependent and therefore ϕ-dependent. A difficulty in the discretization of this surface tension force is that due to the curvature one has to deal with *second* derivatives. One possible approach (often used in finite difference and finite volume discretizations) is to use the representation

$$\mathcal{K} = \operatorname{div} \mathbf{n}_\Gamma = \operatorname{div} \Big(\frac{\nabla \phi}{\|\nabla \phi\|} \Big)$$

and discretize the term on the right handside. In the finite element setting it is possible to avoid the discretization of second derivatives. The approximation of the localized surface tension force is based on a Laplace-Beltrami characterization of the curvature. For this we have to introduce some elementary notions from differential geometry. For ease of notation, we choose a fixed t and suppress the time-dependence throughout the rest of this section. Let U be an open subset in $\mathbb{R}^3$ and Γ a connected C^2 compact hypersurface contained in U. For a sufficiently smooth function $g : U \to \mathbb{R}$ the tangential derivative (along Γ) is defined by projecting the derivative on the tangent space of Γ, i.e.,

$$\nabla_\Gamma g = \nabla g - \nabla g \cdot \mathbf{n}_\Gamma\, \mathbf{n}_\Gamma. \tag{4.12}$$

The *Laplace-Beltrami operator* on Γ is defined by

$$\Delta_\Gamma g := \nabla_\Gamma \cdot \nabla_\Gamma g.$$

It can be shown that $\nabla_\Gamma g$ and $\Delta_\Gamma g$ depend only on values of g on Γ. For vector valued functions $f,\, g : \Gamma \to \mathbb{R}^3$ we define

$$\Delta_\Gamma f := (\Delta_\Gamma f_1, \Delta_\Gamma f_2, \Delta_\Gamma f_3)^T, \qquad \nabla_\Gamma f \cdot \nabla_\Gamma g := \sum_{i=1}^{3} \nabla_\Gamma f_i \cdot \nabla_\Gamma g_i.$$

We recall the following basic result from differential geometry.

Theorem 1 *Let* $\mathrm{id}_\Gamma : \Gamma \to \mathbb{R}^3$ *be the identity on* Γ *and* $\mathcal{K} = \kappa_1 + \kappa_2$ *the sum of the principal curvatures. For all sufficiently smooth vector functions* $\mathbf{v}$ *on* Γ *the following holds:*

$$\int_\Gamma \mathcal{K}\, \mathbf{n}_\Gamma \cdot \mathbf{v}\, ds = -\int_\Gamma (\Delta_\Gamma\, \mathrm{id}_\Gamma) \cdot \mathbf{v}\, ds = \int_\Gamma \nabla_\Gamma\, \mathrm{id}_\Gamma \cdot \nabla_\Gamma \mathbf{v}\, ds. \tag{4.13}$$

Let Γ_h be a polyhedral approximation of Γ as described in section 4.2.2. In view of the result in this theorem an obvious choice for $f_{\Gamma_h}(\mathbf{v}_h)$ in (4.3) (that is used in, e.g. [19, 45, 46, 2, 57]) is the following:

$$f_{\Gamma_h}(\mathbf{v}_h) := \tau \int_{\Gamma_h} \nabla_{\Gamma_h}\, \mathrm{id}_{\Gamma_h} \cdot \nabla_{\Gamma_h} \mathbf{v}_h\, ds, \quad \mathbf{v}_h \in \mathbf{V}_h. \tag{4.14}$$

Here id_{Γ_h} denotes the identity $\Gamma_h \to \mathbb{R}^3$, i.e., the coordinate vector on Γ_h. In [2] it is shown that for piecewise quadratic functions $\mathbf{v}$ the result $f_{\Gamma_h}(\mathbf{v}_h)$ can easily be determined exactly (i.e., without any further approximation errors).

Analysis and numerical experiments in [3] yield that for this choice we have

$$\sup_{\mathbf{v}_h \in \mathbf{V}_h} \frac{|f_\Gamma(\mathbf{v}_h) - f_{\Gamma_h}(\mathbf{v}_h)|}{\|\mathbf{v}_h\|_1} \leq c\sqrt{h_\Gamma}, \tag{4.15}$$

and that this bound is sharp w.r.t. the order of convergence for $h_\Gamma \downarrow 0$. Using the second Strang lemma, cf. [4], this approximation error induces an error of the same order of magnitude (in the H^1-norm) in the velocity approximation and thus relatively *large spurious velocities* can occur. In [3] a modified surface tension force discretization with better approximation quality is presented. We briefly explain this method. For this we have to introduce some further notation. Let $\mathbf{n}_h$ be the unit normal on Γ_h (outward pointing from $\Omega_{1,h}$). Since Γ_h is planar piecewise triangular, this normal is piecewise constant (and not defined at the edges of the surface triangulation). We define the orthogonal projection $\mathbf{P}_h$:

$$\mathbf{P}_h(x) := \mathbf{I} - \mathbf{n}_h(x)\mathbf{n}_h(x)^T \quad \text{for } \; x \in \Gamma_h,\; x \text{ not on an edge.}$$

Recall that the discrete level set function ϕ_h is piecewise quadratic. Define

$$\tilde{\mathbf{n}}_h(x) := \frac{\nabla \phi_h(x)}{\|\nabla \phi_h(x)\|}, \qquad \tilde{\mathbf{P}}_h(x) := \mathbf{I} - \tilde{\mathbf{n}}_h(x)\tilde{\mathbf{n}}_h(x)^T, \quad x \in \Gamma_h,\; x \text{ not on an edge.}$$

For the discrete surface tension force as in (4.14) we have, due to $\nabla_{\Gamma_h} g = \mathbf{P}_h \nabla g$ (for smooth functions g), the representation

$$f_{\Gamma_h}(\mathbf{v}_h) = \tau \sum_{i=1}^{3} \int_{\Gamma_h} \mathbf{P}_h(x)\mathbf{e}_i \cdot \nabla_{\Gamma_h} (\mathbf{v}_h)_i\, ds, \tag{4.16}$$

with $\mathbf{e}_i$ the i-th basis vector in $\mathbb{R}^3$ and $(\mathbf{v}_h)_i$ the i-th component of $\mathbf{v}_h$. The *modified* discrete surface tension force is given by

$$\tilde{f}_{\Gamma_h}(\mathbf{v}_h) = \tau \sum_{i=1}^{3} \int_{\Gamma_h} \tilde{\mathbf{P}}_h(x)\mathbf{e}_i \cdot \nabla_{\Gamma_h}(\mathbf{v}_h)_i \, ds. \tag{4.17}$$

The implementation of this functional requires only a minor modification if the implementation of the one in (4.16) is available. In [3] it is shown that under reasonable assumptions on Γ_h and ϕ_h, we have the error bound

$$\sup_{\mathbf{v}_h \in \mathbf{V}_h} \frac{|f_\Gamma(\mathbf{v}_h) - \tilde{f}_{\Gamma_h}(\mathbf{v}_h)|}{\|\mathbf{v}_h\|_1} \leq c h_\Gamma. \tag{4.18}$$

This bound has a $\mathcal{O}(h_\Gamma)$ behaviour (instead of $\mathcal{O}(\sqrt{h_\Gamma})$). Numerical experiments in [3] show a rate of convergence that is even somewhat higher than first order in h_Γ. In DROPS both discretizations (4.14) and (4.17) of the localized force term f_Γ are available. The default choice is the modified functional in (4.17).

Remark 3 The case with a *variable* surface tension coefficient $\tau = \tau(x)$ can also be treated in DROPS. Using partial integration, the surface tension functional in (4.2) takes the form

$$\begin{aligned} f_\Gamma(\mathbf{v}) &= \int_\Gamma (\tau \mathcal{K}\mathbf{n} - \nabla_\Gamma \tau) \cdot \mathbf{v} \, ds \\ &= \int_\Gamma \mathcal{K}\mathbf{n} \cdot (\tau \mathbf{v}) - \nabla_\Gamma \tau \cdot \mathbf{v} \, ds \\ &= \int_\Gamma \tau \, \nabla_\Gamma \mathrm{id}_\Gamma \cdot \nabla_\Gamma \mathbf{v} \; ds + \sum_{i=1}^{3} \int_\Gamma \big(\nabla_\Gamma (\mathrm{id}_\Gamma)_i \cdot \nabla_\Gamma \tau\big)\mathbf{v}_i \; ds \; - \int_\Gamma \nabla_\Gamma \tau \cdot \mathbf{v} \; ds. \end{aligned} \tag{4.19}$$

For discretization of this term we can easily generalize the approach discussed above for the case of a constant surface tension coefficient. The interface Γ is replaced by its approxmation Γ_h and the tangential derivative are approximated (using the projection $\tilde{\mathbf{P}}_h$) in the same way as in (4.17). This results in the discrete surface tension functional:

$$\begin{aligned} \tilde{f}_{\Gamma_h}(\mathbf{v}_h) = &\sum_{i=1}^{3} \int_{\Gamma_h} \tau \tilde{\mathbf{P}}_h(x)\mathbf{e}_i \cdot \nabla_{\Gamma_h}(\mathbf{v}_h)_i \, ds \\ &+ \sum_{i=1}^{3} \int_{\Gamma_h} \big(\tilde{\mathbf{P}}_h(x)\mathbf{e}_i \cdot \nabla_{\Gamma_h} \tau\big)(\mathbf{v}_h)_i \, ds - \int_{\Gamma_h} \big(\tilde{\mathbf{P}}_h(x) \nabla \tau\big) \cdot \mathbf{v}_h \; ds. \end{aligned}$$

We assume that $\tau = \tau(x)$, which is defined for $x \in \Gamma$, has an extension to $x \in \Gamma_h$.

4.2.4 Modified finite element space for the pressure: XFEM

In this section we discuss the finite element spaces Q_h that are used for the approximation of the pressure variable in the Galerkin discretization (4.3). Note that in the two-phase flow problems that we consider, due to surface tension the pressure is smooth in the two subdomains Ω_i but *discontinuous across the interface* Γ. Moreover, the interface Γ (or its approximation Γ_h) is *not* aligned to the faces in the tetrahedral triangulation. Due to this the finite element space Q_h has to be chosen carefully.

To simplify the presentation, we assume that Γ is known. In practice, the approach explained below is applied with Γ replaced by Γ_h. Let $\mathcal{V} = \{x_k\}_{k\in\mathcal{I}}, \mathcal{I} = \{1,\ldots,n\}$, be the set of all vertices in the triangulation $\mathcal{T}_h$. We also assume that $x_k \notin \Gamma$ for all k. For $m \geq 1$ let $H^m(\Omega_1 \cup \Omega_2)$ denote the Sobolev space of functions, for which $u_{|\Omega_i} \in H^m(\Omega_i),\ i = 1,2$, holds. We use the notation $\|u\|^2_{m,\Omega_1\cup\Omega_2} = \|u\|^2_{m,\Omega_1} + \|u\|^2_{m,\Omega_2}$ for $u \in H^m(\Omega_1 \cup \Omega_2)$. We introduce the standard *linear finite element space*

$$W = W_h = \big\{v \in C(\Omega) \mid v|_T \in \mathcal{P}_1 \ \text{ for all } \ T \in \mathcal{T}_h\big\}.$$

For the approximation of functions in $H^m(\Omega_1 \cup \Omega_2)$, $m \geq 1$, that are discontinuous across Γ (in trace sense) the finite element space W is not suitable. In general, for $u \in H^m(\Omega_1 \cup \Omega_2)$, one can not expect a better bound than

$$\inf_{v\in W} \|u - v\|_0 \leq c\sqrt{h}\,\|u\|_{m,\Omega_1\cup\Omega_2}\,,$$

cf. [4]. Therefore one should *not* use the space $Q_h = W$ in the Galerkin discretization (4.3) of the two phase flow problem. To improve this poor approximation quality we extend the space W by adding functions that can represent discontinuities across Γ. For the definition of this space we first introduce some further notation. To simplify this notation we do not express the dependence on h in our notation (for example, W instead of W_h).

The nodal basis in W is denoted by $\{\psi_k\}_{k\in\mathcal{I}}$. Let Ω_Γ be the collection of all tetrahedra that are intersected by Γ, i.e., $\Omega_\Gamma = \cup\{\,T \in \mathcal{T}_h \mid T \cap \Gamma \neq \emptyset\}$. Let $R_i : L^2(\Omega) \to L^2(\Omega)$, $i = 1,2$, be *restriction* operators:

$$R_i v = \begin{cases} v_{|\Omega_i} & \text{on } \ \Omega_i \\ 0 & \text{on } \ \Omega \setminus \Omega_i \end{cases}$$

(in L^2-sense). We introduce subsets of $\mathcal{I}$ for which the corresponding basis functions have a nonzero intersection with Γ:

$$\begin{aligned} \mathcal{I}_1^\Gamma &:= \{\, k \in \mathcal{I} \mid x_k \in \Omega_2 \ \text{ and } \ \operatorname{supp}(\psi_k) \cap \Gamma \neq \emptyset\,\} \\ \mathcal{I}_2^\Gamma &:= \{\, k \in \mathcal{I} \mid x_k \in \Omega_1 \ \text{ and } \ \operatorname{supp}(\psi_k) \cap \Gamma \neq \emptyset\,\}. \end{aligned}$$

Corresponding spaces are defined by

$$W_i^\Gamma := \operatorname{span}\{\, R_i\psi_k \mid k \in \mathcal{I}_i^\Gamma\,\}, \quad i = 1,2.$$

We introduce the *extended finite element space*, which is defined as

$$W^\Gamma := R_1 W \oplus R_2 W. \tag{4.20}$$

Another characterization of W^Γ is given by

$$W^\Gamma = W \oplus W_1^\Gamma \oplus W_2^\Gamma.$$

Thus the standard linear finite element space $W = W_h$ is *extended* by the spaces W_i^Γ, $i = 1,2$. Note that $v \in W_i^\Gamma$ is discontinuous across Γ, $\operatorname{supp}(v) \subset \Omega_\Gamma$ and that $v(x_j) = 0$ for all $x_j \in \mathcal{V}$. In [13] the following approximation property of the XFEM space W^Γ is derived:

Theorem 2 *For integers $0 \leq l < m \leq 2$ the following holds*

$$\inf_{v\in W^\Gamma} \|u - v\|_{l,\Omega_1\cup\Omega_2} \leq c\,h^{m-l}\|u\|_{m,\Omega_1\cup\Omega_2}, \quad \textit{for all } \ u \in H^m(\Omega_1 \cup \Omega_2). \tag{4.21}$$

This result shows that this space has optimal approximation properties for piecewise smooth functions. Therefore *the space $Q_h = W^\Gamma$ is a good choice in the Galerkin discretization of the two-phase flow problem.* This space has been implemented in DROPS. We briefly discuss a few specific issues that do not arise if standard finite element spaces (like, for example, W) are used.

Basis used in the XFEM space W^Γ

Let $\{\psi_k\}_{1\le k\le n}$ be the standard nodal basis in the finite element space W. In [13] it is shown that

$$\{\psi_k\}_{1\le k\le n} \cup \{R_1\psi_k\}_{k\in\mathcal{I}_1^\Gamma} \cup \{R_2\psi_k\}_{k\in\mathcal{I}_2^\Gamma}$$

forms a basis of the extended finite element space W^Γ. Thus there is a unique representation

$$v = \sum_{k=1}^{n} \beta_k\psi_k + \sum_{k\in\mathcal{I}_1^\Gamma} \beta_k^{(1)} R_1\psi_k + \sum_{k\in\mathcal{I}_2^\Gamma} \beta_k^{(2)} R_2\psi_k, \quad v \in W^\Gamma. \tag{4.22}$$

This basis is used in the implementation. Concerning stability of this basis we note the following. Let $\mathbf{M}_\Gamma$ be the mass matrix of this basis with respect to the L^2-scalar product and $\mathbf{D}_\Gamma := \operatorname{diag}(\mathbf{M}_\Gamma)$. In [13] it is proved that the matrix $\mathbf{D}_\Gamma^{-1}\mathbf{M}_\Gamma$ has a spectral condition number that is unformly (w.r.t. the mesh size h) bounded. Moreover, the constants that occur in the spectral condition number bounds are also independent of the supports of the basis functions $R_i\psi_k$, $k \in \mathcal{I}_i^\Gamma$. In other words, a simple scaling is sufficient to control the stability (in L^2) of the basis functions with "very small" supports.

Modified XFEM space: delete functions with very small supports

In general there are basis functions $R_i\psi_k \in W_i^\Gamma$ with very small support in the sense that $|\operatorname{supp}(R_i\psi_k)|/|\operatorname{supp}(\psi_k)| \ll 1$. It is clear that if functions with "very small" support are deleted from the space W_i^Γ this will not influence the approximation quality of the XFEM space W^Γ significantly. Therefore we introduce a smaller space in which basis functions from W_i^Γ with very small support are deleted. Avoiding very small supports has advantages, for example if the contributions are dominated by rounding errors. We will explain how we chose the maximal size of these "small supports" in order to maintain optimal approximation properties of the resulting reduced XFEM space.

Let $\alpha > 0$, $\tilde{c} > 0$ be given parameters. Let $\mathcal{I}_i^\gamma \subset \mathcal{I}_i^\Gamma$ be the index set such that for all $k \in \mathcal{I}_i^\Gamma \setminus \mathcal{I}_i^\gamma$:

$$\frac{\|\psi_k\|_{l,T\cap\Omega_i}}{\|\psi_k\|_{l,T}} \le \tilde{c}\, h_T^\alpha \quad \text{for all } T \subset \big(\operatorname{supp}(\psi_k) \cap \Omega_\Gamma\big). \tag{4.23}$$

Remark 4 Note that for a function $R_i\psi_k \in W_i^\Gamma$ ($k \in \mathcal{I}_i^\Gamma$) we have $\|R_i\psi_k\|_{l,T} = \|\psi_k\|_{l,T\cap\Omega_i}$ for all $T \in \mathcal{T}_h$. Furthermore, because $\|\psi_k\|_{l,T} \sim c h_T^{\frac{1}{2}-l}$, for $l = 0, 1$, the condition (4.23) can be replaced by the following one:

$$\|\psi_k\|_{l,T\cap\Omega_i} \le \hat{c}\, h_T^{\alpha+1\frac{1}{2}-l} \quad \text{for all } T \subset \big(\operatorname{supp}(\psi_k) \cap \Omega_\Gamma\big). \tag{4.24}$$

The constant $\hat{c}$ may differ from $\tilde{c}$ in (4.23).

We define the *reduced* spaces $W_i^\gamma \subset W_i^\Gamma$ by

$$W_i^\gamma := \operatorname{span}\{\, R_i\psi_k \mid k \in \mathcal{I}_i^\gamma \,\}, \quad i = 1, 2,$$

and a *modified* XFEM space $\tilde{W}^\Gamma := W \oplus W_1^\gamma \oplus W_2^\gamma$. For this space the following approximation property holds, cf. [13]:

Theorem 3 *We assume $\{\mathcal{T}_h\}_{h>0}$ to be quasi-uniform. For $0 \leq l < m \leq 2$ the following holds:*

$$\inf_{v \in \tilde{W}^\Gamma} \|u - v\|_{l,\Omega_1 \cup \Omega_2} \leq c\big(h^{m-l} + h^{\alpha-l}\big)\|u\|_{m,\Omega_1 \cup \Omega_2} \quad \textit{for all} \quad u \in H^m(\Omega_1 \cup \Omega_2).$$

From this result we conclude that the order of approximation of the modified space $\tilde{W}^\Gamma$ is the same as that of W^Γ if we take $\alpha = m$. In the context of our applications $m = 1$ is a natural choice. Therefore the criterion (4.24) with $l = 0$ and $\alpha = 1$ is used to decide which basis functions are deleted from W_i^Γ. The constant $\hat{c}$ has to be set by the user. The default value is $\hat{c} = 0.1$.

Implementation issues

We comment on two implementation issues. The basis of the (modified) XFEM space contains functions that are discontinuous across Γ. Therefore, in the process of assembling the matrices in the finite element discretization, integrals over $T \in \Omega_\Gamma$ have to be treated carefully. This is further discussed in section 4.2.5.

The (modified) XFEM space depends on the position of the interface (and thus on the level set function ϕ). Therefore, in an instationary two-phase flow problem in each time step the XFEM finite element space can be different. This causes additional technical difficulties in the implementation. For example, one needs suitable interpolation procedures to handle pressure unknowns that are deleted or created due to the change of the XFEM space. Furthermore, in the linear algebra part one has to deal with the varying dimension of the pressure space.

4.2.5 Quadrature

At several places integrals over subdomains $\Omega_{i,h} \cap T$ $(i = 1, 2)$ occur. Here T is a tetrahedron and $\Omega_{i,h}$ the discrete approximation of the subdomain Ω_i. This approximation is given through the discrete aproximation of the interface:

$$\Omega_{1,h} = \mathrm{int}(\Gamma_h), \quad \Omega_{2,h} = \Omega \setminus \Omega_{1,h}.$$

Due to the fact that Γ_h is piecewise planar, these subdomains $\Omega_{i,h} \cap T$ have a relatively simple geometric structure, cf. Figure 4.2.

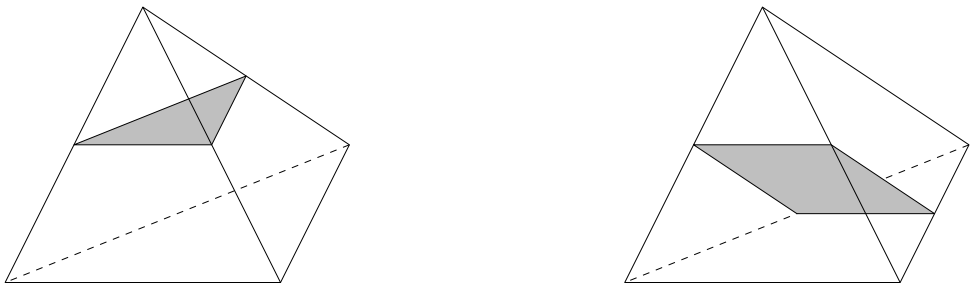

Figure 4.2: Planar intersections of Γ_h and $T' \in \mathcal{T}_h'$

Quadrature rules for numerical integration over these subdomains are available. We outline these numerical integration methods. In section 3.2.2 we explained quadrature rules of degree two and five on tetrahedra. If $T \cap \Gamma_h = \emptyset$ we can use these rules. Otherwise we use the fact that Γ_h is piecewise planar with respect to the triangulation $\mathcal{T}_h'$. We compute the planar intersection between Γ_h and the eight children of T. Such a child is denoted by T', cf. Figure 4.2. Now we iterate over these children and distinguish two cases (cf. Figure 4.2):

- The intersection is a triangle; this means that the interface divides the tetrahedron into a tetrahedron and a prism. The prism can be subdivided into three tetrahedra. Hence we can use the quadrature rules implemented for tetrahedra.
- The intersection is a quadrilateral; the interface divides the tetrahedron into two prisms. Both prisms can be subdivided into three tetrahedra. Hence we can use the quadrature rules implemented for tetrahedra.

We also have to compute integrals over the approximate interface Γ_h. For this we use a quadrature rule on triangles of degree five. This rule has seven nodes and corresponding weights as given in table 4.1 (nodes in barycentric coordinates).

nodes	weights	
$(1/3, 1/3, 1/3)$	$9/40$	
(A_1, A_1, B_1) (A_1, B_1, A_1) (B_1, A_1, A_1)	$(155 - \sqrt{15})/1200$	$A_1 = (6 - \sqrt{15})/21$ $B_1 = (9 + 2\sqrt{15})/21$
(A_2, A_2, B_2) (A_2, B_2, A_2) (B_2, A_2, A_2)	$(155 + \sqrt{15})/1200$	$A_2 = (6 + \sqrt{15})/21$ $B_2 = (9 - 2\sqrt{15})/21$

Table 4.1: Quadrature rule on triangles of degree 5

4.2.6 Re-initialization method for the level set function

During the evolution the level set function ϕ_h can become distorted and in general loses its property of being an approximate signed distance function. At several places in the numerical routines it plays a role that

$$|\nabla \phi_h(x)\,| := \sqrt{\Big(\frac{\partial \phi_h(x)}{\partial x_1}\Big)^2 + \Big(\frac{\partial \phi_h(x)}{\partial x_2}\Big)^2 + \Big(\frac{\partial \phi_h(x)}{\partial x_3}\Big)^2} \approx 1, \tag{4.25}$$

in particular for x "close" to the interface. Thus there is a need for re-initialization of the level set function ϕ_h. After evolving ϕ_h over a few time steps it is re-initialized based on the following criteria: we try to fulfil (4.25); the change in the discrete interface Γ_h (= zero level of ϕ_h) should be small; we aim at mass conservation.

The condition (4.25) leads to the so called Eikonal equation

$$\begin{aligned} |\,\nabla \phi\,| &= 1 \quad \text{in} \quad \Omega \\ \phi &= \phi_\Gamma \quad \text{on} \quad \Gamma \subset \Omega. \end{aligned}$$

For this kind of equation several numerical solution methods are known. In particular, for re-initialization one often introduces a pseudo time variable τ and considers an instationary problem of the form

$$\frac{\partial \phi}{\partial \tau} = S_\alpha(\phi^{\text{old}})\big(1 - |\phi|\big)$$

with initial condition $\phi(0, x) = \phi_h^{\text{old}}$ and S_α a regularized sign function. Numerical methods can be applied to determine an aproximate stationary solution of this equation which then is taken as re-initialization of ϕ_h^{old}.

Another approach, called fast marching method (FMM), is based on a greedy grid traversal technique combined with a function for computing (approximate) distances. Such an FMM has been implemented in DROPS. We outline this method. It consists of two phases: an initialization phase, where the vertices that are directly adjacent to Γ are assigned new values and a second (far field) phase in which all the vertices further away are updated.

Let Γ_h be the discrete interface that has been constructed as described in section 4.2.2 and $\mathcal{T}_h'$ the uniform refinement of $\mathcal{T}_h$ that is used in this construction. Let

$$\mathcal{T}_\Gamma := \{\, T \in \mathcal{T}_h' \mid \mathrm{meas}_2(T \cap \Gamma_h) > 0 \,\}$$

be the set of tetrahedra that have a nonzero intersection with the interface. The discrete interface Γ_h consists of *planar* segments and can be represented as

$$\Gamma_h = \cup_{T \in \mathcal{T}_\Gamma} \Gamma_T \quad \text{with} \quad \Gamma_T := \Gamma_h \cap T.$$

For a vertex v the set of tetrahedra $T \in \mathcal{T}_h'$ that have v as a vertex is denoted by $\mathcal{T}(v)$. For $T \in \mathcal{T}_h'$ the set of vertices of T is denoted by $\mathcal{V}(T)$.

Initialization phase

In the initialization phase an approximate disctance function for the vertices $v \in \mathcal{V}_\Gamma := \{\, v \in \mathcal{V}(T) \mid T \in \mathcal{T}_\Gamma \,\}$ is determined. For a given $T \in \mathcal{T}_\Gamma$ let $\Gamma_T = \Gamma_h \cap T$ be the planar segment that is part of Γ_h (cf. Fig. 4.2) and let $Q_1, \ldots, Q_m$, $m = 3$ or 4, be the vertices of this segment. Let $W \subset \mathbb{R}^3$ be the plane that contains Γ_T and P_W the orthogonal projection on W. For $T \in \mathcal{T}_\Gamma$ and $v \in \mathcal{V}(T)$ we define

$$d_T(v) := \begin{cases} ||v - P_W(v)|| & \text{if } P_W(v) \in T \\ \min_{1 \le j \le m} ||v - Q_j|| & \text{otherwise.} \end{cases} \tag{4.26}$$

Since a vertex is part of multiple tetrahedra, we still have to decide which tetrahedron T we use to determine the (approximate) distance $d(v)$:

$$d(v) := \min \left\{\, d_T(v) \mid v \in \mathcal{V}(T) \text{ and } T \in \mathcal{T}_\Gamma \,\right\}. \tag{4.27}$$

All vertices $v \in \mathcal{V}(T)$, $T \in \mathcal{T}_\Gamma$ now have values $d(v)$.

Far field phase

In this second phase of the FMM we extend the approximate distance function $d(v)$ to the neighbors of those vertices which are already assigned a value. To organize this procedure, we introduce three sets: *FAR* to contain those vertices which are not affected by the algorithm yet, *FIN* containing those that are already assigned a final value and *ACT* which contains the vertices that are currently updated, i.e. those adjacent to *FIN*. Let $\mathcal{V}(\Omega)$ denote the set of all vertices in Ω and $\mathcal{T}(v)$ denote the set of tetrahedra adjacent to v. The three sets are initialized as follows:

$$\begin{aligned} \mathit{FIN} &:= \{v \in \mathcal{V}(\Omega) \mid v \text{ is assigned a value } d(v) \text{ during initialization}\} \\ \mathit{ACT} &:= \{v \in \mathcal{V}(\Omega) \mid v \text{ has a neighbor in } \mathit{FIN}\} \\ \mathit{FAR} &:= \{v \in \mathcal{V}(\Omega) \mid v \text{ has no neighbor in } \mathit{FIN}\}. \end{aligned}$$

We determine values $d(v)$ for all $v \in \mathit{ACT}$ in a similar way as in the initialization phase. Take $v \in \mathit{ACT}$, $T \in \mathcal{T}(v)$ such that $\mathcal{V}(T) \cap \mathit{FIN} \neq \emptyset$. There are three possible cases, namely $|\mathcal{V}(T) \cap \mathit{FIN}| = 1, 2$ or 3. If $|\mathcal{V}(T) \cap \mathit{FIN}| = 1$, say $\mathcal{V}(T) \cap \mathit{FIN} = w$, we define

$$d_T(v) := d(w) + ||v - w||.$$

If T has $m = 2$ or $m = 3$ vertices common with *FIN* we use an orthogonal projection as in the initialization phase. Let w_1, w_2 (and w_3) be these common vertices and W either the line or plane through w_1, w_2 (and w_3). Then

$$d_T(v) := \begin{cases} d(P_W v) + ||P_W v - v|| & \text{if } P_W v \in T \\ \min_{1 \le j \le m}\{d(w_j) + ||v - w_j||\} & \text{otherwise.} \end{cases} \tag{4.28}$$

The value $d(P_W v)$ is calculated by linear interpolation of the known values $d(w_j),\ 1 \le j \le m$. The approximate distance for $v \in ACT$ is then determined by

$$d(v) := \min\big\{\, d_T(v) \mid T \in \mathcal{T}(v),\ \text{ with } \mathcal{V}(T) \cap FIN \neq \emptyset \,\big\}. \tag{4.29}$$

After all $v \in ACT$ have been updated, we choose

$$v_{min} := \text{argmin}_{v \in ACT}\, d(v).$$

This vertex is removed from ACT and added to *FIN*. All the neighbors of v_{min} that are in *FAR* become an element of ACT and the whole process is repeated.

After we have determined the approximate distance grid function $d(v)$, we still have to mark if a vertex is "inside" or "outside" the zero level set. This is done by using the sign of the given level set function ϕ_h^{old}:

$$d(v) := \text{sign}(\phi_h^{\text{old}}) d(v). \tag{4.30}$$

The calculated values $d(v)$ uniquely determine a piecewise quadratic function ϕ_h^{new} on the triangulation $\mathcal{T}_h$. This function is defined to be the re-initialization of ϕ_h^{old}. For this function one can construct an approximate zero level set Γ_h^{new} as explained in section 4.2.2. The re-initialization procedure guarantees that

$$\Gamma_h^{\text{new}} \subset \cup_{T \in \mathcal{T}_\Gamma} T$$

holds, and in this sense the change in the discrete interface is small. Clearly in general we have $\Gamma_h^{\text{new}} \neq \Gamma_h$.

4.2.7 Mass conservation

The temporal and spatial discretization of the level set equation are not mass conserving. Due to the surface tension, we usually *lose* mass from Ω_1. This loss of mass is reduced if the grid is refined. Such finer grids, however, result in higher computational costs. Therefore we introduce another strategy to compensate for the mass loss.

After each time step, we shift the interface in normal direction such that the volume of Ω_1 at current time is the same as at time $t = 0$. To realize this we exploit the fact that the level set function is close to a signed distance function. In order to shift the interface over a distance d in outward normal direction, we only have to subtract d from the level set function.

Let $V(\phi) := \text{vol}\{\, x \in \Omega \mid \phi(x) < 0 \,\}$ denote the volume of Ω_1 corresponding to a level set function ϕ and let ϕ_h be the discrete level set function at a given time. We have to find $d \in \mathbb{R}$ such that

$$V(\phi_h - d) - \text{vol}(\Omega_1(0)) = 0$$

holds. In order to keep the number of evaluations of V low, we use a method with a high rate of convergence, namely the Anderson-Björk method [17], to solve this equation. We then set $\phi_h^{\text{new}} := \phi_h - d$ and discard ϕ_h.

Note that this strategy only works if Ω_1 consists of a single component. If there are multiple components, mass must be preserved for each of them. In this case the algorithm can be modified to shift ϕ_h only locally. Discontinuities that may occur in the level set function can be removed by a reparametrization step.

Finally note that the shifting of the level set function to obtain a better mass conservation introduces a new source of discretization errors.

4.3 Time integration

For the two-phase flow problem, the time discretization is based on a generalization of the θ-scheme given in section 3.3.1 for the one-phase flow Navier-Stokes equations. This generalized method is not found in the literature and therefore we describe its derivation in detail. The need for a generalization has two causes. Firstly, opposite to the one-phase flow problem the mass matrix $\mathbf{M}$ is no longer constant but may vary in time. Secondly, if in the discretization the XFEM space is used then the matrix $\mathbf{B}$ is in general also time dependent (due to the dynamics of the interface). In the sections below we present two derivations of a generalized θ-schema. In the first case we allow $\mathbf{M}$ to be time dependent but assume that $\mathbf{B}$ does *not* depend on t. In the second case we allow both $\mathbf{M}$ and $\mathbf{B}$ to be time dependent. The resulting schemes are very similar.

4.3.1 The generalized θ-scheme for a time independent B

We first consider the Navier-Stokes part in (4.7)-(4.8). We use the notation

$$\mathbf{G}(\vec{\mathbf{u}}, \vec{\boldsymbol{\phi}}, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) = \vec{\mathbf{g}}(\vec{\boldsymbol{\phi}}(t)) + \vec{\mathbf{f}}_{\Gamma_h}(\vec{\boldsymbol{\phi}}(t)) - \mathbf{A}(\vec{\boldsymbol{\phi}}(t))\vec{\mathbf{u}}(t) - \mathbf{N}(\vec{\boldsymbol{\phi}}(t), \vec{\mathbf{u}}(t))\vec{\mathbf{u}}(t).$$

Then the Navier-Stokes equations can be written as

$$\begin{aligned} \mathbf{M}(\vec{\boldsymbol{\phi}}(t))\frac{d\vec{\mathbf{u}}}{dt}(t) + \mathbf{B}^T\vec{\mathbf{p}}(t) &= \mathbf{G}(\vec{\mathbf{u}}, \vec{\boldsymbol{\phi}}, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) \\ \mathbf{B}\vec{\mathbf{u}}(t) &= 0, \end{aligned} \tag{4.31}$$

or, equivalently,

$$\begin{aligned} \frac{d\vec{\mathbf{u}}}{dt}(t) + \mathbf{M}(\vec{\boldsymbol{\phi}}(t))^{-1}\mathbf{B}^T\vec{\mathbf{p}}(t) &= \mathbf{M}(\vec{\boldsymbol{\phi}}(t))^{-1}\mathbf{G}(\vec{\mathbf{u}}, \vec{\boldsymbol{\phi}}, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) \\ \mathbf{B}\vec{\mathbf{u}}(t) &= 0. \end{aligned} \tag{4.32}$$

We assume that $\mathbf{B}$ does *not* depend on t. To obtain a system of ODEs we eliminate the algebraic equation $\mathbf{B}\vec{\mathbf{u}}(t) = 0$ and the (Lagrange multiplier) $\vec{\mathbf{p}}(t)$ as follows. In the notation we suppress the dependence on $\vec{\boldsymbol{\phi}}, \vec{\mathbf{g}}$ and $\vec{\mathbf{f}}_{\Gamma_h}$ and write $\mathbf{M}(t) = \mathbf{M}(\vec{\boldsymbol{\phi}}(t))$, $\mathbf{G}(\vec{\mathbf{u}}, t) = \mathbf{G}(\vec{\mathbf{u}}, \vec{\boldsymbol{\phi}}, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h})$. From $\mathbf{B}\frac{d\vec{\mathbf{u}}}{dt}(t) = 0$ (here we used that $\mathbf{B}$ does not depend on t) and substitution of $\frac{d\vec{\mathbf{u}}}{dt}(t)$ from the first equation we obtain

$$\mathbf{S}(t)\vec{\mathbf{p}}(t) = \mathbf{B}\mathbf{M}(t)^{-1}\mathbf{G}(\vec{\mathbf{u}}, t), \quad \mathbf{S}(t) := \mathbf{B}\mathbf{M}(t)^{-1}\mathbf{B}^T. \tag{4.33}$$

The matrix $\mathbf{B}^T$ has full rank and thus $\mathbf{S}(t)$ is invertible. Using (4.33) we can eliminate $\vec{\mathbf{p}}(t)$ from the first equation in (4.32) resulting in

$$\begin{aligned} \frac{d\vec{\mathbf{u}}}{dt}(t) &= \left[\mathbf{I} - \mathbf{M}(t)^{-1}\mathbf{B}^T\mathbf{S}(t)^{-1}\mathbf{B}\right]\mathbf{M}(t)^{-1}\mathbf{G}(\vec{\mathbf{u}}, t) \\ &=: \mathbf{Q}(t)\mathbf{M}(t)^{-1}\mathbf{G}(\vec{\mathbf{u}}, t). \end{aligned} \tag{4.34}$$

The projection $\mathbf{Q}(t) = I - \mathbf{M}(t)^{-1}\mathbf{B}^T\mathbf{S}(t)^{-1}\mathbf{B}$ satisfies $\mathbf{B}\mathbf{Q}(t) = 0$. Hence, if $\mathbf{B}\vec{\mathbf{u}}(0) = 0$ then the solution $\vec{\mathbf{u}}(t)$ of the ordinary differential equation (4.34) remains in the subspace $\mathrm{Ker}(\mathbf{B})$. To this system of ODEs the θ-scheme is applied, resulting in the discretization

$$\frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} = \theta\mathbf{Q}(t_{n+1})\mathbf{M}(t_{n+1})^{-1}\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_{n+1}) + (1-\theta)\mathbf{Q}(t_n)\mathbf{M}(t_n)^{-1}\mathbf{G}(\vec{\mathbf{u}}^n, t_n). \tag{4.35}$$

We assume that for each n this system has a unique solution $\vec{\mathbf{u}}^{n+1}$ (which is the case for Δt sufficiently small). If $\mathbf{B}\vec{\mathbf{u}}^0 = 0$ then $\mathbf{B}\vec{\mathbf{u}}^n = 0$ for all $n \geq 1$. For implementation it is convenient to eliminate the projection $\mathbf{Q}$ by introducing a suitable Lagrange multiplier. Define $\vec{\mathbf{p}}^k := \mathbf{S}(t_k)^{-1}\mathbf{B}\mathbf{M}(t_k)^{-1}\mathbf{G}(\vec{\mathbf{u}}^k, t_k)$. Then (4.35) takes the form

$$\begin{aligned}\frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} &= \theta\mathbf{M}(t_{n+1})^{-1}\big(\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_{n+1}) - \mathbf{B}^T\vec{\mathbf{p}}^{n+1}\big) \\ &\quad + (1-\theta)\mathbf{M}(t_n)^{-1}\big(\mathbf{G}(\vec{\mathbf{u}}^n, t_n) - \mathbf{B}^T\vec{\mathbf{p}}^n\big).\end{aligned} \tag{4.36}$$

Assume that $\vec{\mathbf{u}}^0$ is such that $\mathbf{B}\vec{\mathbf{u}}^0 = 0$. The sequence $(\vec{\mathbf{u}}^n)_{n\geq 0}$ defined by the θ-scheme (4.35) satisfies (4.36) and also $\mathbf{B}\vec{\mathbf{u}}^n = 0$ for all n. We use $\vec{\mathbf{p}}^k$ as a Lagrange multiplier to enforce $\mathbf{B}\vec{\mathbf{u}}^k = 0$ as follows. Given $\vec{\mathbf{u}}^0$ with $\mathbf{B}\vec{\mathbf{u}}^0 = 0$ define

$$\vec{\mathbf{p}}^0 := \mathbf{S}(t_0)^{-1}\mathbf{B}\mathbf{M}(t_0)^{-1}\mathbf{G}(\vec{\mathbf{u}}^0, t_0),$$

and for $n \geq 0$ let $\vec{\mathbf{u}}^{n+1}$, $\vec{\mathbf{p}}^{n+1}$ be such that

$$\begin{aligned}\frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} &= \theta\mathbf{M}(t_{n+1})^{-1}\big(\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_{n+1}) - \mathbf{B}^T\vec{\mathbf{p}}^{n+1}\big) + (1-\theta)\mathbf{M}(t_n)^{-1}\big(\mathbf{G}(\vec{\mathbf{u}}^n, t_n) - \mathbf{B}^T\vec{\mathbf{p}}^n\big) \\ \mathbf{B}\vec{\mathbf{u}}^{n+1} &= 0\end{aligned} \tag{4.37}$$

holds. Note that in this saddle point type system the projection $\mathbf{Q}$ is *not* used. Due to (4.36) this system has a solution. If we assume that for each n the saddle point problem (4.37) has a unique solution (which is true for Δt sufficiently small) then this yields the solution of the θ-scheme in (4.35).

Remark 5 If the mass matrix $\mathbf{M}$ does *not* depend on t then we can take

$$\vec{\mathbf{p}}^k := \mathbf{S}^{-1}\mathbf{B}\mathbf{M}^{-1}\big(\theta\mathbf{G}(\vec{\mathbf{u}}^k, t_k) + (1-\theta)\mathbf{G}(\vec{\mathbf{u}}^{k-1}, t_{k-1})\big),\ k \geq 1,$$

and instead of (4.36) we obtain

$$\frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} = \mathbf{M}^{-1}\big(\theta\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_{n+1}) + (1-\theta)\mathbf{G}(\vec{\mathbf{u}}^n, t_n\big) - \mathbf{M}^{-1}\mathbf{B}^T\vec{\mathbf{p}}^{n+1},$$

which then results in the standard θ-scheme for a one-phase Navier Stokes equation, as described in section 3.3.1.

For $\theta = 0$ the method in (4.37) corresponds to the explicit Euler method applied to (4.34), which is not very useful do to its poor stability properties. We consider $\theta \neq 0$. In this case, in (4.37) inverses of both $\mathbf{M}(t_{n+1})$ and $\mathbf{M}(t_n)$ occur. The latter can be avoided by introducing an additional variable, leading to a more convenient (but equivalent) formulation of (4.37). This is done as follows. Define

$$\vec{\mathbf{z}}^k := \mathbf{M}(t_k)^{-1}\big(\mathbf{G}(\vec{\mathbf{u}}^k, t_k) - \mathbf{B}^T\vec{\mathbf{p}}^k\big),\quad k \geq 0,$$

i.e.,

$$\begin{aligned}\mathbf{M}(t_0)\vec{\mathbf{z}}^0 &= \mathbf{G}(\vec{\mathbf{u}}^0, t_0) - \mathbf{B}^T\vec{\mathbf{p}}^0\\ \theta\vec{\mathbf{z}}^{k+1} &= \frac{\vec{\mathbf{u}}^{k+1} - \vec{\mathbf{u}}^k}{\Delta t} - (1-\theta)\vec{\mathbf{z}}^k, \quad k \geq 0.\end{aligned}$$

Using thus, (4.37) can be reformulated as

$$\begin{aligned}\mathbf{M}(t_{n+1})\frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} + \theta\mathbf{B}^T\vec{\mathbf{p}}^{n+1} &= \theta\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_{n+1}) + (1-\theta)\mathbf{M}(t_{n+1})\vec{\mathbf{z}}^n\\ \mathbf{B}\vec{\mathbf{u}}^{n+1} &= 0\\ \theta\vec{\mathbf{z}}^{n+1} &= \frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} - (1-\theta)\vec{\mathbf{z}}^n,\end{aligned} \tag{4.38}$$

for $n \geq 0$ and a starting value $\vec{\mathbf{z}}^0$ as defined above.

Application of the θ-scheme to the level set equation (4.9) results in

$$\frac{\vec{\boldsymbol{\phi}}^{n+1} - \vec{\boldsymbol{\phi}}^n}{\Delta t} = -\theta\mathbf{E}(\vec{\mathbf{u}}^{n+1})^{-1}\mathbf{H}(\vec{\mathbf{u}}^{n+1})\vec{\boldsymbol{\phi}}^{n+1} - (1-\theta)\mathbf{E}(\vec{\mathbf{u}}^n)^{-1}\mathbf{H}(\vec{\mathbf{u}}^n)\vec{\boldsymbol{\phi}}^n.$$

This can be reformulated using a new variable

$$\vec{\mathbf{w}}^k = -\mathbf{E}(\vec{\mathbf{u}}^k)^{-1}\mathbf{H}(\vec{\mathbf{u}}^k),$$

which satisfies (for $\theta \neq 0$)

$$\theta\vec{\mathbf{w}}^{n+1} = \frac{\vec{\boldsymbol{\phi}}^{n+1} - \vec{\boldsymbol{\phi}}^n}{\Delta t} - (1-\theta)\vec{\mathbf{w}}^n,$$

resulting in

$$\mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\boldsymbol{\phi}}^{n+1} - \vec{\boldsymbol{\phi}}^n}{\Delta t} = -\theta\mathbf{H}(\vec{\mathbf{u}}^{n+1})\vec{\boldsymbol{\phi}}^{n+1} + (1-\theta)\mathbf{E}(\vec{\mathbf{u}}^{n+1})\vec{\mathbf{w}}^n. \tag{4.39}$$

Combining these results and inserting the notation for $\mathbf{G}$ we obtain, for $\theta \neq 0$, the following coupled nonlinear system for $(\vec{\mathbf{u}}^n, \vec{\mathbf{p}}^n, \vec{\boldsymbol{\phi}}^n) \to (\vec{\mathbf{u}}^{n+1}, \vec{\mathbf{p}}^{n+1}, \vec{\boldsymbol{\phi}}^{n+1})$:

Given $\vec{\mathbf{u}}^0$, $\vec{\boldsymbol{\phi}}^0$, determine $\vec{\mathbf{z}}^0$ and $\vec{\mathbf{w}}^0$ as follows:

$$\begin{aligned}
\mathbf{G}(\vec{\mathbf{u}}^0, \vec{\boldsymbol{\phi}}^0, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) &= \vec{\mathbf{g}}(\vec{\boldsymbol{\phi}}^0) + \vec{\mathbf{f}}_{\Gamma_h}(\vec{\boldsymbol{\phi}}^0) - \mathbf{A}(\vec{\boldsymbol{\phi}}^0)\vec{\mathbf{u}}^0 - \mathbf{N}(\vec{\boldsymbol{\phi}}^0, \vec{\mathbf{u}}^0)\vec{\mathbf{u}}^0 \\
\mathbf{B}\mathbf{M}(\vec{\boldsymbol{\phi}}^0)^{-1}\mathbf{B}^T\vec{\mathbf{p}}^0 &= \mathbf{B}\mathbf{M}(\vec{\boldsymbol{\phi}}^0)^{-1}\mathbf{G}(\vec{\mathbf{u}}^0, \vec{\boldsymbol{\phi}}^0, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) \\
\mathbf{M}(\vec{\boldsymbol{\phi}}^0)\vec{\mathbf{z}}^0 &= \mathbf{G}(\vec{\mathbf{u}}^0, \vec{\boldsymbol{\phi}}^0, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) - \mathbf{B}^T\vec{\mathbf{p}}^0 \\
\mathbf{E}(\vec{\mathbf{u}}^0)\vec{\mathbf{w}}^0 &= \mathbf{H}(\vec{\mathbf{u}}^0)\vec{\boldsymbol{\phi}}^0
\end{aligned} \tag{4.40}$$

For $n \geq 0$:

$$\begin{aligned}
\mathbf{M}(\vec{\boldsymbol{\phi}}^{n+1})\frac{\vec{\mathbf{u}}^{n+1}}{\Delta t} + \theta\big[\mathbf{A}(\vec{\boldsymbol{\phi}}^{n+1})\vec{\mathbf{u}}^{n+1} + \mathbf{N}(\vec{\boldsymbol{\phi}}^{n+1}, \vec{\mathbf{u}}^{n+1})\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{g}}(\vec{\boldsymbol{\phi}}^{n+1}) - \vec{\mathbf{f}}_{\Gamma_h}(\vec{\boldsymbol{\phi}}^{n+1})\big] + \theta\mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\
= \mathbf{M}(\vec{\boldsymbol{\phi}}^{n+1})\frac{\vec{\mathbf{u}}^n}{\Delta t} + (1-\theta)\mathbf{M}(\vec{\boldsymbol{\phi}}^{n+1})\vec{\mathbf{z}}^n, \\
\mathbf{B}\vec{\mathbf{u}}^{n+1} = 0 \\
\mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\boldsymbol{\phi}}^{n+1}}{\Delta t} + \theta\mathbf{H}(\vec{\mathbf{u}}^{n+1})\vec{\boldsymbol{\phi}}^{n+1} = \mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\boldsymbol{\phi}}^n}{\Delta t} + (1-\theta)\mathbf{E}(\vec{\mathbf{u}}^{n+1})\vec{\mathbf{w}}^n \\
\theta\vec{\mathbf{z}}^{n+1} = \frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} - (1-\theta)\vec{\mathbf{z}}^n \\
\theta\vec{\mathbf{w}}^{n+1} = \frac{\vec{\boldsymbol{\phi}}^{n+1} - \vec{\boldsymbol{\phi}}^n}{\Delta t} - (1-\theta)\vec{\mathbf{w}}^n.
\end{aligned} \tag{4.41}$$

Remark 6 The derivation above shows that the scheme in (4.40)-(4.41) is a reformulation of the θ-scheme applied to the system of ODEs in (4.34). The latter is A-stable and first order accurate for $\theta \in (0, 1]$ and second order accurate for $\theta = \frac{1}{2}$.

Remark 7 For the case $\theta = 1$ the scheme takes a much simpler form. In particular the sequences for $\vec{\mathbf{z}}^n$ and $\vec{\mathbf{w}}^n$ are not needed. The resulting method is as follows:

Given $\vec{\mathbf{u}}^0$, $\vec{\boldsymbol{\phi}}^0$, determine for $n \geq 0$:

$$\begin{aligned}
\mathbf{M}(\vec{\boldsymbol{\phi}}^{n+1})\frac{\vec{\mathbf{u}}^{n+1}}{\Delta t} + \big[\mathbf{A}(\vec{\boldsymbol{\phi}}^{n+1})\vec{\mathbf{u}}^{n+1} + \mathbf{N}(\vec{\boldsymbol{\phi}}^{n+1}, \vec{\mathbf{u}}^{n+1})\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{g}}(\vec{\boldsymbol{\phi}}^{n+1}) - \vec{\mathbf{f}}_{\Gamma_h}(\vec{\boldsymbol{\phi}}^{n+1})\big] + \mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\
= \mathbf{M}(\vec{\boldsymbol{\phi}}^{n+1})\frac{\vec{\mathbf{u}}^n}{\Delta t} \\
\mathbf{B}\vec{\mathbf{u}}^{n+1} = 0 \\
\mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\boldsymbol{\phi}}^{n+1}}{\Delta t} + \mathbf{H}(\vec{\mathbf{u}}^{n+1})\vec{\boldsymbol{\phi}}^{n+1} = \mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\boldsymbol{\phi}}^n}{\Delta t}.
\end{aligned} \tag{4.42}$$

4.3.2 The generalized θ-scheme for a time dependent B

In this section we allow both $\mathbf{M}$ and $\mathbf{B}$ to be time dependent. We use the same notation as in the previous section. For the derivation of a generalized θ-scheme we apply an approach that is

standard in the field of differential algebraic equations, cf. [**?**].

We introduce

$$\vec{\mathbf{y}} := \begin{pmatrix} \vec{\mathbf{u}} \\ \vec{\phi} \\ \vec{\mathbf{p}} \end{pmatrix}, \quad \mathbf{C}(\vec{\mathbf{y}}) = \begin{pmatrix} \mathbf{M}(\vec{\phi}) & & \\ & \mathbf{E}(\vec{\mathbf{u}}) & \\ & & 0 \end{pmatrix}, \quad \mathbf{F}(\vec{\mathbf{y}}) = \begin{pmatrix} \mathbf{G}(\vec{\mathbf{u}}, \vec{\phi}, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) - \mathbf{B}^T\vec{\mathbf{p}} \\ -\mathbf{H}(\vec{\mathbf{u}})\vec{\phi} \\ \mathbf{B}\vec{\mathbf{u}} \end{pmatrix} =: \begin{pmatrix} \mathbf{F}_1(\vec{\mathbf{y}}) \\ \mathbf{F}_2(\vec{\mathbf{y}}) \\ \mathbf{F}_3(\vec{\mathbf{y}}) \end{pmatrix}.$$

The Navier-Stokes equations coupled with the level set equation can be written as

$$\mathbf{C}(\vec{\mathbf{y}})\frac{d\vec{\mathbf{y}}}{dt}(t) = \mathbf{F}(\vec{\mathbf{y}}),$$

with initial condition for $\vec{\mathbf{u}}(t_0)$ and $\vec{\phi}(t_0)$. For the numerical treatment of this DAE system we introduce the variable $\vec{\mathbf{q}} := \frac{d\vec{\mathbf{y}}}{dt}$ and thus we obtain the coupled DAE system

$$\begin{aligned} \frac{d\vec{\mathbf{y}}}{dt}(t) &= \vec{\mathbf{q}} \\ \mathbf{C}(\vec{\mathbf{y}})\vec{\mathbf{q}} - \mathbf{F}(\vec{\mathbf{y}}) &= 0. \end{aligned} \tag{4.43}$$

The θ-scheme, with $\theta \neq 0$, applied to this takes the form:

$$\begin{aligned} \vec{\mathbf{y}}^{n+1} &= \vec{\mathbf{y}}^n + \Delta t\big(\theta\vec{\mathbf{q}}^{n+1} + (1-\theta)\vec{\mathbf{q}}^n\big) \\ \mathbf{C}(\vec{\mathbf{y}}^{n+1})\vec{\mathbf{q}}^{n+1} - \mathbf{F}(\vec{\mathbf{y}}^{n+1}) &= 0. \end{aligned} \tag{4.44}$$

We reformulate this method as follows. We decompose $\vec{\mathbf{q}}^n$ as $\vec{\mathbf{q}}^n = (\vec{\mathbf{z}}^n, \vec{\mathbf{w}}^n, \vec{\mathbf{r}}^n)^T$. We only consider $\theta \neq 0$. From $\mathbf{C}(\vec{\mathbf{y}}^{n+1})\theta\vec{\mathbf{q}}^{n+1} - \theta\mathbf{F}(\vec{\mathbf{y}}^{n+1}) = 0$ and $\theta\vec{\mathbf{q}}^{n+1} = \frac{1}{\Delta t}(\vec{\mathbf{y}}^{n+1} - \vec{\mathbf{y}}^n) - (1-\theta)\vec{\mathbf{q}}^n$ we get, for $n \geq 0$,

$$\begin{aligned} \mathbf{M}(\vec{\phi}^{n+1})\frac{\vec{\mathbf{u}}^{n+1}}{\Delta t} - \theta\mathbf{F}_1(\vec{\mathbf{y}}^{n+1}) &= \mathbf{M}(\vec{\phi}^{n+1})\frac{\vec{\mathbf{u}}^n}{\Delta t} + (1-\theta)\mathbf{M}(\vec{\phi}^{n+1})\vec{\mathbf{z}}^n, \\ \mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\phi}^{n+1}}{\Delta t} - \theta\mathbf{F}_2(\vec{\mathbf{y}}^{n+1}) &= \mathbf{E}(\vec{\mathbf{u}}^{n+1})\frac{\vec{\phi}^n}{\Delta t} + (1-\theta)\mathbf{E}(\vec{\mathbf{u}}^{n+1})\vec{\mathbf{w}}^n, \\ 0 &= \mathbf{F}_3(\vec{\mathbf{y}}^{n+1}), \end{aligned}$$

and

$$\begin{aligned} \theta\vec{\mathbf{z}}^{n+1} &= \frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} - (1-\theta)\vec{\mathbf{z}}^n \\ \theta\vec{\mathbf{w}}^{n+1} &= \frac{\vec{\phi}^{n+1} - \vec{\phi}^n}{\Delta t} - (1-\theta)\vec{\mathbf{w}}^n. \end{aligned}$$

In this form we have *exactly the same scheme as in* (4.41). For $n = 0$ we need starting values $\vec{\mathbf{z}}^0 = \frac{d\vec{\mathbf{u}}}{dt}(t_0)$, $\vec{\mathbf{w}}^0 = \frac{d\vec{\phi}}{dt}(t_0)$. From $\mathbf{E}(\vec{\mathbf{u}})\frac{d\vec{\phi}}{dt} = \mathbf{H}(\vec{\mathbf{u}})\vec{\phi}$ we get $\mathbf{E}(\vec{\mathbf{u}}^0)\vec{\mathbf{w}}^0 = \mathbf{H}(\vec{\mathbf{u}}^0)\vec{\phi}^0$ and thus the same starting value for $\vec{\mathbf{w}}^0$ as in (4.40). For $\vec{\mathbf{z}}^0$ we consider (4.31) and obtain

$$\frac{d\mathbf{B}}{dt}\vec{\mathbf{u}} + \mathbf{B}\frac{d\vec{\mathbf{u}}}{dt} = 0$$

and thus (with the same notation as in the previous section)

$$\mathbf{S}(t)\vec{\mathbf{p}}(t) = \mathbf{B}(t)\mathbf{M}(t)^{-1}\mathbf{G}(\vec{\mathbf{u}}, t) + \frac{d\mathbf{B}}{dt}\vec{\mathbf{u}}(t)$$

From this we obtain the system

$$\mathbf{S}(t_0)\vec{\mathbf{p}}^0 = \mathbf{B}(t_0)\mathbf{M}(\vec{\boldsymbol{\phi}}^0)^{-1}\mathbf{G}(\vec{\mathbf{u}}^0, \vec{\boldsymbol{\phi}}^0, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) + \frac{d\mathbf{B}}{dt}(t_0)\vec{\mathbf{u}}(t_0). \tag{4.45}$$

Note that if $\mathbf{B}$ does *not* depend on t this system is the same as the one in (4.40). Given this $\vec{\mathbf{p}}^0$ the starting value $\vec{\mathbf{z}}^0$ can be determined from

$$\mathbf{M}(\vec{\boldsymbol{\phi}}^0)\vec{\mathbf{z}}^0 = \mathbf{G}(\vec{\mathbf{u}}^0, \vec{\boldsymbol{\phi}}^0, \vec{\mathbf{g}}, \vec{\mathbf{f}}_{\Gamma_h}) - \mathbf{B}^T\vec{\mathbf{p}}^0,$$

which is the same as in (4.40). We see that if $\mathbf{B}$ is *in*dependent of t then we obtain the same starting values as in (4.40) and hence the method based on the DAE system (4.43) is *exactly the same* as the one derived in the previous section (4.40)-(4.41). In that case the consistency and stability properties can be derived from the fact that the method is a reformulation of a θ-scheme applied to the system of ODEs (4.34), cf. Remark 6. If $\mathbf{B}$ depends on t then the scheme (4.41)-(4.40) with a modified starting value for $\vec{\mathbf{p}}$ as in (4.45) results from a formal application of the θ-scheme to the DAE system (4.43) and an analysis of the accuracy and stability properties of this method is not available, yet.

4.3.3 An implicit Euler type of method

We present a variant of an implicit Euler method which is particularly attractive due to its simplicity. Starting point is the ODE system for the Navier-Stokes part in (4.34):

$$\frac{d\vec{\mathbf{u}}}{dt}(t) = \mathbf{Q}(t)\mathbf{M}(t)^{-1}\mathbf{G}(\vec{\mathbf{u}}, t).$$

For the discretization the projection operator is treated explicitly, whereas the operator $\mathbf{G}(\vec{\mathbf{u}}, t)$ (which causes the stiffness in this problem) is treated semi-implicitly. If for the latter we apply an implicit Euler method we obtain the discretization

$$\frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} = \mathbf{Q}(t_n)\mathbf{M}(t_n)^{-1}\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_n).$$

By introducing a Lagrange multiplier $\vec{\mathbf{p}}^{k+1} := \mathbf{S}(t_k)^{-1}\mathbf{B}\mathbf{M}(t_k)^{-1}\mathbf{G}(\vec{\mathbf{u}}^{k+1}, t_k)$ the projection $\mathbf{Q}(t_n)$ can be eliminated and we obtain the saddle point form

$$\begin{aligned} \frac{\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{u}}^n}{\Delta t} &= \mathbf{M}(t_n)^{-1}\big(\mathbf{G}(\vec{\mathbf{u}}^{n+1}, t_n) - \mathbf{B}^T\vec{\mathbf{p}}^{n+1}\big) \\ \mathbf{B}\vec{\mathbf{u}}^{n+1} &= 0. \end{aligned} \tag{4.46}$$

The same idea can be applied to the level set equation and thus we get the following time integration scheme for the coupled problem

Given $\vec{\mathbf{u}}^0$, $\vec{\boldsymbol{\phi}}^0$, determine for $n \geq 0$:

$$\begin{aligned} \mathbf{M}(\vec{\boldsymbol{\phi}}^n)\frac{\vec{\mathbf{u}}^{n+1}}{\Delta t} + \big[\mathbf{A}(\vec{\boldsymbol{\phi}}^n)\vec{\mathbf{u}}^{n+1} + \mathbf{N}(\vec{\boldsymbol{\phi}}^n, \vec{\mathbf{u}}^{n+1})\vec{\mathbf{u}}^{n+1} - \vec{\mathbf{g}}(\vec{\boldsymbol{\phi}}^n) - \vec{\mathbf{f}}_{\Gamma_h}(\vec{\boldsymbol{\phi}}^n)\big] + \mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\ = \mathbf{M}(\vec{\boldsymbol{\phi}}^n)\frac{\vec{\mathbf{u}}^n}{\Delta t} \\ \mathbf{B}\vec{\mathbf{u}}^{n+1} = 0 \\ \mathbf{E}(\vec{\mathbf{u}}^n)\frac{\vec{\boldsymbol{\phi}}^{n+1}}{\Delta t} + \mathbf{H}(\vec{\mathbf{u}}^n)\vec{\boldsymbol{\phi}}^{n+1} = \mathbf{E}(\vec{\mathbf{u}}^n)\frac{\vec{\boldsymbol{\phi}}^n}{\Delta t}. \end{aligned} \tag{4.47}$$

This scheme is similar to the one in remark 7, but now the mass matrices $\mathbf{M}$ and $\mathbf{E}$ are treated explicitly (i.e. evaluated at t_n instead of t_{n+1}), in the Navier-Stokes equations the level set function is treated explicitly and in the level set equation the velocity is treated explicitly. Due to this, per time step there is a *decoupling* between the Navier-Stokes and level set equation.

4.3.4 The generalized fractional-step θ-scheme

We generalize the method from section 3.3.2 to the two-phase flow system (4.7)-(4.9). Forthcoming.

4.4 Decoupling and linearization

In the two-phase flow case, besides the nonlinear system for the velocity $\vec{\mathbf{u}}$, we also have the nonlinear coupling between the flow variables $(\vec{\mathbf{u}}, \vec{\mathbf{p}})$ and the level set one $\vec{\boldsymbol{\phi}}$. A time step in the θ-scheme (4.41) is given by

$$\begin{cases} [\frac{1}{\Delta t}\mathbf{M} + \theta\mathbf{A}](\vec{\boldsymbol{\phi}}^{n+1})\vec{\mathbf{u}}^{n+1} + \theta\mathbf{N}(\vec{\boldsymbol{\phi}}^{n+1}, \vec{\mathbf{u}}^{n+1})\vec{\mathbf{u}}^{n+1} + \theta\mathbf{B}^T\vec{\mathbf{p}}^{n+1} \\ \quad = \theta[\vec{\mathbf{g}} + \vec{\mathbf{f}}_{\Gamma_h}](\vec{\boldsymbol{\phi}}^{n+1}) + \mathbf{M}(\vec{\boldsymbol{\phi}}^{n+1})\big(\frac{\vec{\mathbf{u}}^n}{\Delta t} + (1-\theta)\vec{\mathbf{z}}^n\big) \\ \mathbf{B}\vec{\mathbf{u}}^{n+1} = 0 \\ [\frac{1}{\Delta t}\mathbf{E} + \theta\mathbf{H}](\vec{\mathbf{u}}^{n+1})\vec{\boldsymbol{\phi}}^{n+1} = \mathbf{E}(\vec{\mathbf{u}}^{n+1})\big(\frac{\vec{\boldsymbol{\phi}}^n}{\Delta t} + \vec{\mathbf{w}}^n\big). \end{cases}$$

The nonlinear coupling between $(\vec{\mathbf{u}}^{n+1}, \vec{\mathbf{p}}^{n+1})$ and $\vec{\boldsymbol{\phi}}^{n+1}$ is decoupled by a fixed point iteration, in each iteration which results in a linear system for the level set function and a nonlinear system for the velocity and pressure. The fixed point iteration reads:

- Set

$$\vec{\mathbf{g}}^n := \frac{\vec{\mathbf{u}}^n}{\Delta t} + (1-\theta)\vec{\mathbf{z}}^n$$
$$\vec{\mathbf{h}}^n := \frac{\vec{\boldsymbol{\phi}}^n}{\Delta t} + \vec{\mathbf{w}}^n.$$

- Initialize $\vec{\mathbf{u}}_0^{n+1}$ and $\vec{\boldsymbol{\phi}}_0^{n+1}$ with the value $\vec{\mathbf{u}}^n$, $\vec{\boldsymbol{\phi}}^n$ from the previous time step. Iterate for $k = 0, 1, \ldots$

  - Compute the level set vector $\vec{\boldsymbol{\phi}}_{k+1}^{n+1}$ from the linear system

$$\big[\frac{1}{\Delta t}\mathbf{E} + \theta\mathbf{H}\big](\vec{\mathbf{u}}_k^{n+1})\vec{\boldsymbol{\phi}}_{k+1}^{n+1} = \mathbf{E}(\vec{\mathbf{u}}_k^{n+1})\vec{\mathbf{h}}^n. \tag{4.48}$$

  - Solve the following equations for $(\vec{\mathbf{u}}_{k+1}^{n+1}, \vec{\mathbf{p}}_{k+1}^{n+1})$

$$\begin{cases} \big[\frac{1}{\Delta t}\mathbf{M} + \theta\mathbf{A}\big](\vec{\boldsymbol{\phi}}_{k+1}^{n+1})\vec{\mathbf{u}}_{k+1}^{n+1} + \theta\mathbf{N}(\vec{\boldsymbol{\phi}}_{k+1}^{n+1}, \vec{\mathbf{u}}_{k+1}^{n+1})\vec{\mathbf{u}}_{k+1}^{n+1} + \theta\mathbf{B}^T\vec{\mathbf{p}}_{k+1}^{n+1} \\ \qquad = \theta\big[\vec{\mathbf{g}} + \vec{\mathbf{f}}_{\Gamma_h}\big](\vec{\boldsymbol{\phi}}_{k+1}^{n+1}) + \mathbf{M}(\vec{\boldsymbol{\phi}}_{k+1}^{n+1})\vec{\mathbf{g}}^n \\ \mathbf{B}\vec{\mathbf{u}}_{k+1}^{n+1} = 0 \end{cases} \tag{4.49}$$

The nonlinear system (4.49) is very similar to the discrete Navier-Stokes equations (3.17) of the one-phase flow problem. Note, however, that the matrices in (4.49) are different due to the fact that the viscosity and density are not constant over the whole domain. This nonlinear system has the form

$$\begin{aligned} \tilde{\mathbf{A}}\mathbf{x} + \mathbf{N}(\mathbf{x})\mathbf{x} + \mathbf{B}^T\mathbf{y} &= \mathbf{b} \\ \mathbf{B}\mathbf{x} &= \mathbf{c} \end{aligned} \tag{4.50}$$

and can be solved with the fixed point defect correction method explained in section 3.4. This results in saddle point problems of the form

$$\mathbf{K}\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} := \begin{pmatrix}\hat{\mathbf{A}} & \mathbf{B}^T\\ \mathbf{B} & 0\end{pmatrix}\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} = \begin{pmatrix}\mathbf{f}_1\\ \mathbf{f}_2\end{pmatrix} \ , \quad \hat{\mathbf{A}} := \mathbf{A} + \mathbf{N}(\mathbf{x}^{\text{old}}) + \beta\mathbf{M} \ . \tag{4.51}$$

that have a very similar structure as the ones in section 3.5.3.

4.5 Iterative solvers

As can be seen from section 4.4 the decoupling and linearization method results in two types of linear problems, namely an Oseen (i.e., linearized Navier-Stokes) problem and a linear discrete hyperbolic problem, cf. (4.51) and (4.48). The latter can be solved by, for example, a preconditioned GMRES or BiCGSTAB method, cf. section 3.5.1. For the Oseen equations one can use the methods discussed in section 3.5.3.

In the two-phase flow applications that we consider (chapter **??**) we typically have a (very) small time step. Due to this the weighting of the mass matrix part in the Oseen problem (which scales like $1/\Delta t$) is relatively large compared to the diffusion and convection parts. Therefore systems with the matrix $\hat{\mathbf{A}}$ are not very hard to solve. Simple preconditioners $\mathbf{Q}_A$ suffice to obtain a reasonably efficient method. For the Schur complement preconditioner we can use the same one as for the one-phase Navier-Stokes equations.

If one considers a two-phase flow problem in which the two phases are gas and liquid, then the differences in density and viscosity between the two phases is very large. In such a case it might be necessary to modify the Schur complement preconditioner such that it takes these large density and viscosity jumps into account. Such a preconditioner was introduced in [7].

Chapter 5

Two-phase flow with transport of a dissolved species (NS2+M)

5.1 Weak formulation of the transport equation

We recall the strong formulation of the two-phase flow problem coupled with a transport equation as described in section 1.1.

The two-phase flow is modeled by (cf. (1.4)):

$$\begin{aligned} \rho(\phi)\Big(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\Big) &= -\nabla p + \rho(\phi)\mathbf{g} + \operatorname{div}(\mu(\phi)\mathbf{D}(\mathbf{u})) + \tau\mathcal{K}\delta_\Gamma \mathbf{n}_\Gamma \\ \operatorname{div}\mathbf{u} &= 0 \\ \phi_t + \mathbf{u}\cdot\nabla\phi &= 0, \end{aligned} \tag{5.1}$$

together with suitable initial and boundary conditions. The transport equation for the concentration of a dissolved species is given by (cf. (1.5))

$$\frac{\partial c}{\partial t} + \mathbf{u}\cdot\nabla c = D(\phi)\Delta c, \tag{5.2}$$

$$[D(\phi)\nabla c\cdot\mathbf{n}] = 0 \quad \text{at the interface,} \tag{5.3}$$

$$c_1 = C_H c_2 \quad \text{at the interface.} \tag{5.4}$$

The diffusion coefficient is piecewise constant: $D(\phi) = D_1 + (D_2 - D_1)H(\phi)$. In the interface condition we use the notation c_i for $c_{|\Omega_i}$ restricted to the interface. The constant $C_H > 0$ is given (Henry's constant). The model has to be combined with suitable initial and boundary conditions.

Note that the transport equation needs as input the flow field $\mathbf{u}$ that results from the Navier-Stokes equations. If the surface tension coefficient τ is assumed to be *independent* of the concentration c there is no dependence of the Navier-Stokes equation (5.1) on the transport problem (5.2). Due to this we can use a simple "one direction decoupling" between the Navier-Stokes equation and the transport problem as described in section 5.4.

The weak formulation of the Navier-Stokes equations for two-phase flow is given in section 4.1. Below we describe a weak formulation of the convection-diffusion equation (5.2). Note that due to the Henry interface condition in (5.4) the concentration c is *discontinuous* across the interface (for $C_H \neq 1$). To eliminate the discontinuity we introduce a piecewise constant weighting

function

$$C_H(\phi(t)) = \begin{cases} 1 & \text{in } \Omega_1 \text{ (i.e., if } \phi(x,t) \le 0), \\ C_H & \text{in } \Omega_2 \text{ (i.e., if } \phi(x,t) > 0). \end{cases}$$

We define the scaled function $\tilde{c} = C_H(\phi)c$. In view of (5.4) this function should be continuous (maybe in some weak sense) across the interface. For this transformed concentration we derive a weak formulation. Define

$$\tilde{D}(\phi) = C_H(\phi)^{-1} D(\phi).$$

For simplicity we assume homogeneous zero boundary conditions for $\tilde{c}$ on $\partial\Omega$. Furthermore, for a given function $\tilde{c}_0$ we assume an initial condition $\tilde{c}(0) = \tilde{c}_0$. The weak formulation is as follows:

Determine $\tilde{c}(t) \in H_0^1(\Omega)$ such that for all $t \in [0,T]$:

$$\big(C_H(\phi(t))^{-1}\tilde{c}_t(t), v\big) + \big(C_H(\phi(t))^{-1}\mathbf{u}\cdot\nabla\tilde{c}(t), v\big) = -\big(\tilde{D}(\phi(t))\nabla\tilde{c}(t), \nabla v\big) \;\; \forall\, v \in H_0^1(\Omega). \quad (5.5)$$

Here $(c,v) = \int_\Omega c(x)v(x)\,dx$ denotes the L^2-scalar product. Note that in our application the *velocity field* $\mathbf{u}$ *is time-dependent, i.e.* $\mathbf{u} = \mathbf{u}(t)$. A solution $\tilde{c}$ of this problem satisfies $[\tilde{c}]_\Gamma = 0$ (in trace sense) and thus $c = C_H(\phi)^{-1}\tilde{c}$ satisfies the Henry interface condition in (5.4). Furthermore, a solution $\tilde{c}$ satisfies $[\tilde{D}(\phi(t))\nabla\tilde{c}\cdot\mathbf{n}]_\Gamma = 0$ and thus $[D(\phi)\nabla c\cdot\mathbf{n}]_\Gamma = 0$, which is the flux continuity condition in (5.3). If the solution $\tilde{c}$ it sufficiently smooth then it satisfies the differential equation

$$C_H(\phi(t))^{-1}\frac{\partial\tilde{c}}{\partial t}(t) + C_H(\phi(t))^{-1}\mathbf{u}\cdot\nabla\tilde{c}(t) = \tilde{D}(\phi(t))\Delta\tilde{c}(t) = C_H(\phi(t))^{-1}D(\phi(t))\Delta\tilde{c}(t)$$

in the two subdomains Ω_1 and Ω_2. This can be rewritten as

$$\frac{\partial c}{\partial t} + \mathbf{u}\cdot\nabla c = D(\phi)\Delta c,$$

in the two subdomains and for all $t \in [0,T]$ and thus we obtain the convection-diffusion equation (5.2). This shows that the problem (5.5) is an appropriate weak formulation of the transport problem. Note that to obtain the original physical quantity one has to rescale:

$$c = C_H(\phi)^{-1}\tilde{c}.$$

5.2 Spatial discretization

We describe a finite element discretization of the problem in (5.5). Due to the fact that the diffusion coefficient $D(\phi)$ is discontinuous across the interface Γ the solution $\tilde{c}$ has low regularity across Γ. From this and from the fact that the interface is not aligned with the faces in the triangulation is follows that standard finite elements will *not have optimal approximation quality* for this type of problem, cf. [55]. An alternative is to use the XFEM method explained in section 4.2.4. For the latter method, however, one needs a modified weak formulation of the transport problem that results in additional technical complications in the implementation. In section 5.2.1 we first consider the standard finite element method applied to (5.5). This method is easy to implement but suboptimal. In section 5.2.2 it is explained how the XFEM method can be applied to (5.5) resulting in a spatial discretization with a higher order of accuracy than the standard FE method.

5.2.1 Standard linear finite element space

Let $V_h \subset V := H_0^1(\Omega)$ be the subspace of continuous linear finite elements. The Galerkin discretization of (5.5) reads: Find $\tilde{c}_h(t) \in V_h$ such that for all $t \in [0,T]$:

$$\big(C_H(\phi(t))^{-1}(\tilde{c}_h)_t(t), v_h\big) + \big(C_H(\phi(t))^{-1}\mathbf{u}\cdot\nabla\tilde{c}_h(t), v_h\big) = -\big(\tilde{D}(\phi(t))\nabla\tilde{c}_h(t), \nabla v_h\big) \quad \forall\ v_h \in V_h.$$

In addition we need a initial condition $\tilde{c}_h(0)$ that is obtained from (interpolation of) $\tilde{c}(0) = \tilde{c}_0$. The standard nodal basis in V_h is denoted by $\{\psi_j\}_{1\le j\le K}$. We introduce matrices

$$\begin{aligned}
\mathbf{M}(\phi) \in \mathbb{R}^{K\times K}, &\quad \mathbf{M}(\phi)_{ij} = \int_\Omega C_H(\phi)^{-1}\psi_i\psi_j\ dx \\
\mathbf{H}(\phi,\mathbf{u}) \in \mathbb{R}^{K\times K}, &\quad \mathbf{H}(\phi,\mathbf{u})_{ij} = \int_\Omega C_H(\phi)^{-1}(\mathbf{u}\cdot\nabla\psi_j)\psi_i\ dx \\
\mathbf{A}(\phi) \in \mathbb{R}^{K\times K}, &\quad \mathbf{A}(\phi)_{ij} = \int_\Omega \tilde{D}(\phi)\nabla\psi_j\cdot\nabla\psi_i\ dx,
\end{aligned}$$

and the representation

$$\tilde{c}_h(t) = \sum_{j=1}^{K} \tilde{c}_j(t)\psi_j.$$

The unknown functions $\tilde{c}_j(t)$ ($t \in [0,T]$) are collected in the vector function

$$\vec{\mathbf{c}}(t) := \big(\tilde{c}_1(t), \ldots, \tilde{c}_K(t)\big).$$

Using this notation we obtain the following system of ordinary differential equations:

Determine $\vec{\mathbf{c}}(t)$ such that $\vec{\mathbf{c}}(0)$ is given and

$$\mathbf{M}(\phi(t))\frac{d\vec{\mathbf{c}}}{dt}(t) + \mathbf{H}\big(\phi(t),\mathbf{u}(t)\big)\vec{\mathbf{c}}(t) = -\mathbf{A}(\phi(t))\vec{\mathbf{c}}(t) \tag{5.6}$$

for all $t \in [0,T]$.

To determine the entries of the matrices $\mathbf{M}, \mathbf{H}$ and $\mathbf{A}$ one has to compute integrals over tetrahedra in which there are discontinuities in the coefficient functions C_H and $\tilde{D}$ across the interface (which is approximated by Γ_h). For this the methods discussed in section 4.2.5 are used.

5.2.2 Nitsche's method combined with XFEM

In this section we use a technique from [55] that results in a finite element discretization method with an optimal order of convergence.
Forthcoming.

5.3 Time integration

We recall the θ-scheme from section 3.3.1:

$$\frac{u^{\text{new}} - u^{\text{old}}}{\Delta t} = (1-\theta)F(u^{\text{old}}) + \theta F(u^{\text{new}}), \quad \theta \in [0,1].$$

If this method is applied to the discrete transport equation in (5.6) with

$$F(\vec{\mathbf{c}}) = -\mathbf{M}(\phi(t))^{-1}\big[\mathbf{A}(\phi(t)) + \mathbf{H}\big(\phi(t), \mathbf{u}(t)\big)\big]\vec{\mathbf{c}}$$

we obtain the following fully discrete problem, with $\phi^k := \phi(t_k)$, $\mathbf{u}^k := \mathbf{u}(t_k)$:

$$\begin{aligned}\vec{\mathbf{c}}^{n+1} - \vec{\mathbf{c}}^{n} = &-\Delta t(1-\theta)\mathbf{M}(\phi^n)^{-1}\big[\mathbf{A}(\phi^n) + \mathbf{H}(\phi^n, \mathbf{u}^n)\big]\vec{\mathbf{c}}^{n} \\ &- \Delta t\theta\mathbf{M}(\phi^{n+1})^{-1}\big[\mathbf{A}(\phi^{n+1}) + \mathbf{H}(\phi^{n+1}, \mathbf{u}^{n+1})\big]\vec{\mathbf{c}}^{n+1}.\end{aligned} \tag{5.7}$$

This can be rewritten as

$$\begin{aligned}&\big[\mathbf{M}(\phi^{n+1}) + \Delta t\theta\big(\mathbf{A}(\phi^{n+1}) + \mathbf{H}(\phi^{n+1}, \mathbf{u}^{n+1})\big)\big]\vec{\mathbf{c}}^{n+1} \\ &= \mathbf{M}(\phi^{n+1})\Big(\mathbf{I} - \Delta t(1-\theta)\mathbf{M}(\phi^n)^{-1}\big[\mathbf{A}(\phi^n) + \mathbf{H}(\phi^n, \mathbf{u}^n)\big]\Big)\vec{\mathbf{c}}^{n}.\end{aligned} \tag{5.8}$$

Thus per time step we have to solve two linear systems if $\theta \in (0, 1)$ and only one linear system if $\theta = 1$.

5.4 Decoupling and linearization

As can be seen from the model (5.1)-(5.4), there is only a coupling in *one* direction between the two-phase flow problem (5.1) and the transport equation (5.2)-(5.4). This allows the following obvious decoupling strategy. Given values for $\mathbf{u}(t_n)$, $\phi(t_n)$ and $\vec{\mathbf{c}}^n \approx c(t_n)$ we first apply a time integration step $t_n \to t_{n+1}$ to the two-phase flow model (5.1). For this we can use the methods treated in section 4.3 and 4.4. This results in approximations for $\mathbf{u}(t_{n+1})$ and $\phi(t_{n+1})$. These can be used in a time integration step $c(t_n) \to c(t_{n+1})$, for example the θ-scheme given in (5.8).

Chapter 6

Two-phase flow with transport of a surfactant (NS2+S)

We assume that in the two-phase flow problem there is a species (called tenside or surfactant) which adheres to the interface and that the concentration of this surfactant in the two phases is so small that it can be neglected in the model. The concentration of this surfactant is denoted by $S(x,t)$, $x \in \Gamma$. We introduce the orthogonal projection $P = I - \mathbf{n}\mathbf{n}^T$ ($\mathbf{n}$: normal on Γ). Correspondingly, for $x \in \Gamma$ we have an orthogonal decomposition $\mathbf{u}(x,t) = P\mathbf{u}(x,t) + (I-P)\mathbf{u}(x,t) =: \mathbf{u}_\Gamma(x,t) + \mathbf{u}_\perp(x,t)$. The tangential gradient is defined by $\nabla_\Gamma := P\nabla$, and $\mathrm{div}_\Gamma := \nabla_\Gamma^T$, $\Delta_\Gamma := \mathrm{div}_\Gamma \nabla_\Gamma$. The transport of surfactants at the interface is modeled by a convection-diffusion equation, cf. [31, 58]:

$$\partial_{t,n} S + \mathrm{div}_\Gamma(S\mathbf{u}_\Gamma) + S\mathcal{K}\mathbf{u}\cdot\mathbf{n} = D_\Gamma \Delta_\Gamma S. \tag{6.1}$$

The derivative $\partial_{t,n} S$ stands for the time derivative of S along a normal path. For the numerical treatment of such a problem we introduced a new technique, cf. [9].

To explain the main idea of this method we first consider a *stationary* model problem, namely the Laplace-Beltrami equation, on a given fixed interface Γ. In weak form this problem is as follows: For given $f \in L^2(\Gamma)$ with $\int_\Gamma f \mathrm{d}\mathbf{s} = 0$, determine $u \in H^1(\Gamma)$ with $\int_\Gamma u \,\mathrm{d}\mathbf{s} = 0$ such that

$$\int_\Gamma \nabla_\Gamma u \nabla_\Gamma v \,\mathrm{d}\mathbf{s} = \int_\Gamma f v \,\mathrm{d}\mathbf{s} \qquad \text{for all } \; v \in H^1(\Gamma). \tag{6.2}$$

The solution u is unique and satisfies $u \in H^2(\Gamma)$ with $\|u\|_{H^2(\Gamma)} \leq c\|f\|_{L^2(\Gamma)}$ and a constant c independent of f.

For the discretization of this problem one needs an approximation Γ_h of Γ. We assume that this approximate manifold is constructed as follows. Let $\{\mathcal{T}_h\}_{h>0}$ be a family of tetrahedral triangulations of a *fixed* domain $\Omega \subset \mathbb{R}^3$ that contains Γ. These triangulations are assumed to be regular, consistent and stable. Take $\mathcal{T}_h \in \{\mathcal{T}_h\}_{h>0}$ and denote the set of tetrahedra that form $\mathcal{T}_h$ by $\{S\}$. We assume that Γ_h is a closed manifold such that

- Γ_h can be decomposed as
$$\Gamma_h = \cup_{T\in\mathcal{F}_h} T, \tag{6.3}$$
where for each T there is a corresponding tetrahedron $S_T \in \mathcal{T}_h$ with $T = S_T \cap \Gamma_h$ and $\mathrm{meas}_2(T) > 0$. To avoid technical complications we assume that this S_T is unique, i.e., T does not coincide with a face of a tetrahedron in $\mathcal{T}_h$.

- Each T from the decomposition in (6.3) is *planar*, i.e., either a triangle or a quadrilateral.

Note that the construction described in section 4.2.2 results in approximate interfaces that satisfy these conditions if for the tetrahedral family we take $\{\mathcal{T}_h'\}_{h>0}$ as explained in section 4.2.2.

The main new idea of our approach is that for discretization of the problem (6.2) *we use a finite element space induced by the continuous linear finite elements on* $\mathcal{T}_h$. This is done as follows. We define a subdomain that contains Γ_h:

$$\omega_h := \cup_{T\in\mathcal{F}_h} S_T. \tag{6.4}$$

This subdomain in $\mathbb{R}^3$ is connected and partitioned in tetrahedra that form a subset of $\mathcal{T}_h$. We introduce the finite element space

$$V_h := \{\, v_h \in C(\omega_h) \mid v_{|S_T} \in P_1 \quad \text{for all} \quad T \in \mathcal{F}_h \,\}. \tag{6.5}$$

This space induces the following space on Γ_h:

$$V_h^\Gamma := \{\psi_h \in H^1(\Gamma_h) \,|\, \exists\ v_h \in V_h : \psi_h = v_h|_{\Gamma_h}\,\}. \tag{6.6}$$

This space is used for a Galerkin discretization of (6.2): determine $u_h \in V_h^\Gamma$ with $\int_{\Gamma_h} u_h \mathrm{d}\mathbf{s}_h = 0$ such that

$$\int_{\Gamma_h} \nabla_{\Gamma_h} u_h \nabla_{\Gamma_h} \psi_h \,\mathrm{d}\mathbf{s}_h = \int_{\Gamma_h} f_h \psi_h \,\mathrm{d}\mathbf{s}_h \qquad \text{for all} \quad \psi_h \in V_h^\Gamma, \tag{6.7}$$

with f_h a suitable extension of f such that $\int_{\Gamma_h} f_h \mathrm{d}\mathbf{s}_h = 0$. Due the Lax-Milgram lemma this problem has a unique solution u_h. In [9] we present a discretization error analysis of this method that shows that under reasonable assumptions we have optimal error bounds.

Remark 8 As far as we know this method for discretization of a partial differential equation on a surface is new. We give some comments related to this approach:

- The family $\{\mathcal{T}_h\}_{h>0}$ is *shape-regular* but the family $\{\Gamma_h\}_{h>0}$ in general is *not shape-regular*. In our applications, cf. chapters 12 and 13, Γ_h contains a significant number of strongly deteriorated triangles that have very small angles. Moreover, neighboring triangles can have very different areas. As is shown in [9], optimal discretization bounds hold if $\{\mathcal{T}_h\}_{h>0}$ is shape-regular; for $\{\Gamma_h\}_{h>0}$ shape-regularity is *not* required.

- Each quadrilateral in $\mathcal{F}_h$ can be subdivided into two triangles. Let $\tilde{\mathcal{F}}_h$ be the induced set consisting of *only* triangles and such that $\cup_{T\in\tilde{\mathcal{F}}_h} T = \Gamma_h$. Define

  $$W_h^\Gamma := \{\, \psi_h \in C(\Gamma_h) \mid \psi_h|_T \in P_1 \quad \text{for all} \quad T \in \tilde{\mathcal{F}}_h \,\}.$$

  The space W_h^Γ is the space of continuous functions that are piecewise linear on the triangles of Γ_h. Clearly $V_h^\Gamma \subset W_h^\Gamma$ holds. There are, however, situations in which $V_h^\Gamma \neq W_h^\Gamma$.

- Let $(\xi_i)_{1\leq i\leq m}$ be the collection of all vertices of all tetrahedra in ω_h and ϕ_i the nodal linear finite element basis function corresponding to ξ_i. Then V_h^Γ is spanned by the functions $\phi_i|_{\Gamma_h}$, $1 \leq i \leq m$. These functions, however, are *not* necessarily independent. In computations we use this generating system $\phi_i|_{\Gamma_h}$, $1 \leq i \leq m$, for solving the discrete problem (6.7). Properties that are of interest for the numerical solution of the resulting linear system, such as conditioning of the mass and stiffness matrix are analyzed in a forthcoming paper.

- In the implementation of this method one has to compute integrals of the form

$$\int_T \nabla_{\Gamma_h}\phi_j \nabla_{\Gamma_h}\phi_i \,\mathrm{d}\mathbf{s}, \quad \int_T f_h\phi_i \,\mathrm{d}\mathbf{s} \quad \text{for } T \in \mathcal{F}_h.$$

  The domain T is either a triangle or a quadrilateral. The first integral can be computed exactly. For the second one standard quadrature rules can be applied, for example the one discussed in section 4.2.5.

First results of this finite element method applied to the Laplace-Beltrami equation on a given surface Γ are presented in [9]. The method will also be used for the spatial discretization of the convection-diffusion problem (6.1). Clearly for the numerical treatment of this problem we also need a time discretization method.
Remainder: forthcoming.

Chapter 7

Two-phase flow with transport of both a dissolved species and a surfactant at the interface

Forthcoming.

Part II

Implementation in DROPS

In this second part we discuss some implementation issues related to the numerical treated in part I. In chapter 8 we describe some fundamental concepts and the most important classes of DROPS. In chapter 9 we give a brief introduction to the parallel version.

Chapter 8

Fundamental concepts and data structures

In this chapter important data structures and algorithms implemented in DROPS are presented. Figure 8.1 gives an overview of the main components of the software. The different modules are arranged in a diagram such that has two levels of structuring, namely in vertical and horizontal direction.

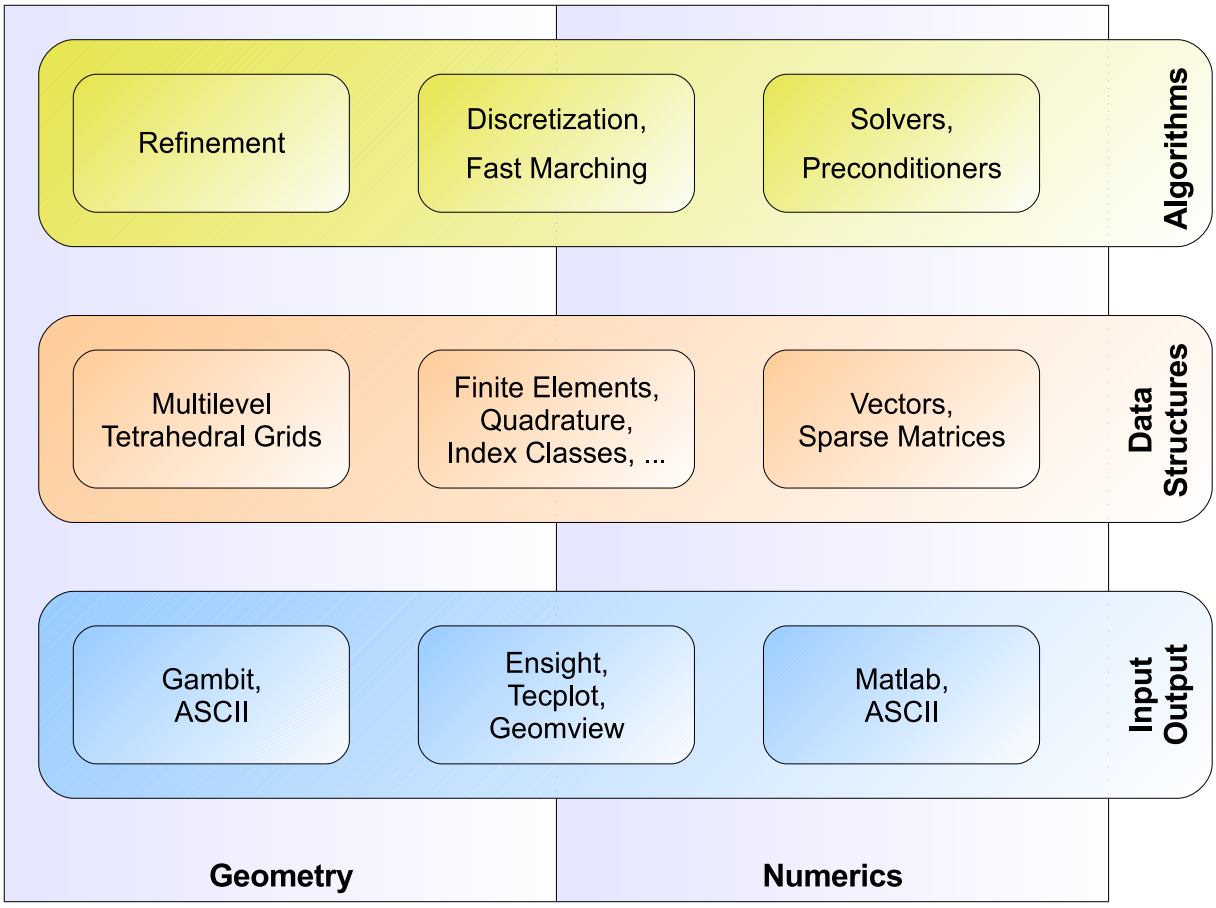

Figure 8.1: Overview of modules and structure of DROPS.

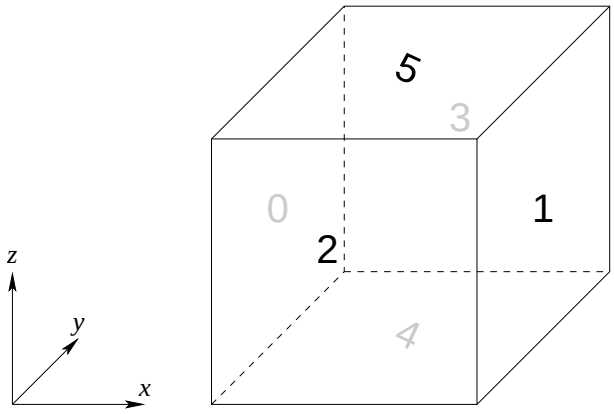

Figure 8.2: A cube and its 6 boundary segments $\Sigma_0, \ldots, \Sigma_5$.

The vertical structure of the figure distinguishes between input and output routines, data structures and algorithms. The different methods related to input and output are described in Section 8.8. In part I we treated many numerical methods from a numerical analysis point of view. In this chapter we discuss implementation issues of these algorithms, in particular their relation with certain data structures. This corresponds to the object-oriented perspective of C++ classes, where data structures (as data members) and functionality (as member functions) are combined with each other.

The horizontal structure in Figure 8.1 shows a classification of the different modules based on the categories 'geometry'and 'numerics', emphasizing the fact that we tried to *decouple geometrical data* such as the grid *from numerical data* such as vectors and matrices. Some tasks, however, require geometrical as well as numerical information and are therefore located in the middle column. For example, the discretization routines for setting up stiffness matrices, where in a loop over all tetrahedra the corresponding matrix entries are determined. In these routines the geometrical and numerical data are coupled. The geometrical and numerical data structures are described in Sections 8.1 and 8.2, respectively.

8.1 Geometrical objects: multilevel triangulation and simplices

In this section we discuss the data structures that represent geometrical objects such as vertices, edges, faces, tetrahedra, the boundary and the multilevel grid. The corresponding data structures are called `VertexCL`, `EdgeCL`, `FaceCL`, `TetraCL`, `BoundaryCL` and `MultiGridCL`, respectively. Note that all C++ classes in DROPS have a suffix `CL` to distinguish data type identifiers from object identifiers.

Boundary and boundary segments

We assume that the boundary $\Sigma = \partial\Omega$ is partitioned into elementary boundary segments Σ_j, $j = 0, \ldots, N_\Sigma - 1$. Note that we use a C style numbering starting with zero. To give an example, if Ω is a cube, then Σ can be partitioned into $N_\Sigma = 6$ boundary segments $\Sigma_0, \ldots, \Sigma_5$, cf. Figure 8.2. Each boundary segment is represented by a `BndSegCL` object. Up to now DROPS can only handle boundaries which are piecewise planar. The class `BoundaryCL` contains an array of all `BndSegCL` objects.

Simplices

In the following we describe the representation of the simplices.

VertexCL. Each vertex V stores its coordinates $\mathbf{x}_V \in \mathbb{R}^3$ as a `Point3DCL` object. If V is located on the boundary Σ, it stores a list of `BndPointCL` objects, each containing the index j of the boundary segment Σ_j with $\mathbf{x}_V \in \Sigma_j$ and the 2D coordinate in the local reference frame. Note that V may be lie on multiple boundary segments. For the example in Figure 8.2 a vertex may be part of up to 3 boundary segments.

EdgeCL. Each edge E is linked to the two vertices V_1, V_2 which are connected by E. If the edge is further refined into two sub-edges E_1, E_2, then there is also a link to the midpoint vertex V_m. Note that $E_1 = V_1V_m$ and $E_2 = V_mV_2$. If E is located on the boundary, then it stores the indices j of the boundary segments with $\{\mathbf{x}_{V_1}, \mathbf{x}_{V_2}\} \subset \Sigma_j$. Note that an edge can be located on at most 2 boundary segments.

FaceCL. Each face F is linked to its neighboring tetrahedra. For a boundary face the index j of the corresponding unique boundary segment Σ_j is stored. A face F may possess up to 4 neighboring tetrahedra. This is the case if F is an inner face connecting two tetrahedra T_1 and T_2 which are irregularly refined such that F is not subdivided by the corresponding green refinement rule. Then there are two green children $T_1' \in \mathcal{K}(T_1)$ and $T_2' \in \mathcal{K}(T_2)$ also sharing F as a common face.

TetraCL. Each tetrahedron T is linked to its 4 vertices, 6 edges and 4 faces. If $\ell(T) > 0$, i. e., T is not stored in the initial triangulation $\mathcal{T}_0$, then T is linked to its parent tetrahedron. If T is refined, then it is also linked to its children $T' \in \mathcal{K}(T)$. T stores the integer values $\mathsf{mark}(T)$ (the refinement mark) and $\mathsf{status}(T)$ (the actual refinement rule). These are used in the refinement algorithm, cf. [1].

Furthermore, each simplex class contains an `UnknownHandleCL` object which stores the indices of unknowns belonging to this simplex, cf. Section 8.3.

Multilevel triangulation

The class `MultiGridCL` represents a multilevel triangulation $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$, cf. Definition 5. The data structure is based on the corresponding hierarchical decomposition $\mathcal{H} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$, cf. Definition 6. The tetrahedra are stored in $J + 1$ lists, each one for the hierarchical surplus $\mathcal{G}_j$ of a different level. The vertices, edges and faces are stored in a similar manner, where the level of such a sub-simplex S is defined as

$$\ell(S) := \min\{\, \ell(T) : T \in \mathcal{H} \text{ contains } S \text{ as sub-simplex} \,\}.$$

Furthermore, `MultiGridCL` contains a `BoundaryCL` object in which all boundary segments are stored.

The `MultiGridCL` constructor takes a `MGBuilderCL` object as input argument which creates the initial triangulation $\mathcal{T}_0$. `MGBuilderCL` serves as an abstract base class from which specific classes can be derived. For instance, the derived class `BrickBuilderCL` can be used to generate an initial triangulation of a cuboid-shaped domain.

The member function `Refine()` calls the refinement algorithm described in [1]. It requires that the tetrahedra $T \in \mathcal{T}_J$ in the input multilevel triangulation are marked for refinement or

for coarsening. This can be achieved by calling the member functions `SetRegRefMark()` or `SetRemoveMark()` of the corresponding `TetraCL` objects.

There are different kinds of iterators to access the simplices in the multilevel triangulation. The `MultiGridCL` member functions `GetTriangTetraBegin(L)` and `GetTriangTetraEnd(L)` return iterators to cycle through all tetrahedra

$$T \in \mathcal{T}_{\mathtt{L}}$$

of a triangulation. Similarly the member functions `GetAllTetraBegin(L)` and `GetAllTetraEnd(L)` can be used to iterate over all

$$T \in \bigcup_{j=0}^{\mathtt{L}} \mathcal{G}_j,$$

where the level $\ell(T)$ of the iterated tetrahedra T is increasing from 0 to `L`. Similar iterators exist for vertices, edges and faces as well.

The iterators are implemented such that a corresponding `for` loop can be executed by multiple OpenMP threads in parallel [66]. This allows for faster computations on shared memory machines. As the importance and availability of multicore architectures is growing nowadays and will grow further in the future, this is a relevant advantage regarding computational efficiency.

8.2 Numerical objects: vectors and sparse matrices

Vectors

In DROPS there are two different type of vectors: `SVectorCL` for short vectors with a handful of entries and `VectorCL` for vectors with a large number of entries. Throughout this chapter we assume that indices always start with the number zero (C style numbering).

`SVectorCL<RowsN>` is a template class with template parameter `RowsN` for vectors $\mathbf{x} \in \mathbb{R}^{\mathtt{RowsN}}$ with a fixed dimension `RowsN`. It is mostly used for storing coordinates. For this purpose we defined the typedefs `Point2DCL`, `Point3DCL` and `BaryCoordCL` which are identical to `SVectorCL<2>`, `SVectorCL<3>` and `SVectorCL<4>`, respectively.

The data type `VectorCL` is used for storing vectors $\vec{x} \in \mathbb{R}^N$ where N is large and may differ from object to object. It is a typedef for `VectorBaseCL<double>`. The class `VectorBaseCL<RealT>` is a template class for vectors with entry type `RealT` and is an ancestor of `std::valarray<RealT>`. Thus `VectorCL` derives the benefits of the efficient expression template mechanisms available for arithmetical operations involving `valarray` objects. By setting a debug flag `DebugNumericC` range checking and other debug features can be enabled which are switched off by default for performance reasons.

Matrices

There are two different types of matrices in DROPS, `SMatrixCL` for small matrices and `MatrixCL` for large sparse matrices.

The template class `SMatrixCL<RowsN,ColsN>` is used for small matrices $M \in \mathbb{R}^{\mathtt{RowsN}\times\mathtt{ColsN}}$ with fixed dimensions.

Sparse matrices are stored in objects of the type `SparseMatBaseCL<RealT>` where `RealT` indicates the type of the entries. For convenience, we introduced a typedef `MatrixCL` for `SparseMatBaseCL<double>`.

We use the *compressed row storage* format (CSR) which is described in the following. For a sparse matrix with m rows and N non-zero entries, `SparseMatBaseCL` contains a vector `RowBegin`

with $m+1$ integer entries, a vector `ColIndex` with N integer entries and a vector `Val` with N entries of type `RealT`. For a row `i` the indices from `RowBegin[i-1]` to `RowBegin[i]-1` indicate the range in `Val` where the values of the non-zero entries are stored. The column indices of the corresponding values are stored in `ColIndex`.

As it is tedious task to compute the sparsity pattern stored in `RowBegin` and `ColIndex`, we use an intermediate storage format called `SparseMatBuilderCL<RealT>` when setting up a new sparse matrix M. The `SparseMatBuilderCL` first collects and accumulates all entries in a `std::map` based data structure. After that a call of the member function `Build()` automatically creates the corresponding `SparseMatBaseCL` object M and deletes the `map`s afterwards.

As `map`s are often too memory consuming we use them only for initializing M. When updating M in subsequent steps the sparsity pattern is reused by default, i. e., access to `SparseMatBuilderCL` entries directly returns the corresponding `SparseMatBaseCL` entries in `Val`. If the sparsity pattern should not be reused (for example when the extended pressure space Q_h^Γ changed because the interface Γ has moved) all matrix entries should be deleted by a call to the member function `clear()` to force a complete initialization of the matrix.

8.3 The connnection between grid and unknowns: indices

As mentioned before we decided to decouple the geometrical data (grid) from the numerical data (matrices, vectors). This is advantageous, because then the iterative solvers only have to deal with matrices and vectors and not with the grid. Then a matrix-vector multiplication does not require a loop over all grid entities which saves substantial computational time. For the interpretation of a solution vector $\vec{\mathbf{u}}$, however, it is necessary to know which vector entries are associated with a certain vertex V, for example. Here the concept of indices comes into play.

Index descriptions and numberings

For each finite element type used in a solution strategy there exists an associated index. An index $\mathcal{J}$ is described by an `IdxDescCL` object. It contains the number of degrees of freedom (DoF) for each simplex type, n_V, n_E, n_F, n_T, and the overall number of unknowns, $N_\mathcal{J}$. To give an example, a P_1-index has $n_V = 1$ DoF per vertex (and $n_E = n_F = n_T = 0$), an index for vector-valued P_2-FE has $n_V = n_E = 3$ DoFs for each vertex and edge (and $n_F = n_T = 0$).

As a next step we have to create a *numbering* of all degrees of freedom which belong to the index $\mathcal{J}$, where degrees of freedom on Dirichlet boundaries are omitted. This is done by a function `CreateNumbering(...)`, which is usually a member function of the problem class (cf. Section 8.4). By this we also obtain the total number of unknowns, $N_\mathcal{J}$, which is equal to the dimension of the vectors associated with $\mathcal{J}$. Thus at the end `CreateNumbering(...)` sets the value $N_\mathcal{J}$ in the corresponding `IdxDesc` object.

The numbering is stored by `UnknownHandleCL` objects contained in the corresponding `VertexCL`, `EdgeCL`, `FaceCL` and `TetraCL` objects. Note that for a single simplex maybe multiple such numbers have to be stored, namely one for each index or, in other words, one for each finite element type.

For an extended finite element space a call to `UpdateXNumbering(...)` augments the usual numbering, also called *base numbering*, by a numbering for the new degrees of freedom (induced by extension of the finite element space). These numbers are not stored in the `UnknownHandleCL` objects, but in a seperate `ExtendedIdxCL` object. It contains a vector `xidx` $\in \mathcal{N}^{N_\mathcal{J}}$ where the entry `xidx[j]` either stores the number of the extended DoF belonging to the base DoF `j` or it

contains a flag that the DoF `j` is not extended. Note that `UpdateXNumbering(...)` has to be called each time the interface has moved to account for the changed extended DoFs.

Vector and matrix descriptions

A `VecDescCL` object contains a vector `Data` of type `VectorCL` and a pointer `RowIdx` to the associated index of type `IdxDescCL`. Calling the member function `SetIdx(idx)` sets the pointer and resizes the vector to the right dimension. Similarly, a `MatDescCL` object contains a sparse matrix `Data` and pointers `RowIdx` and `ColIdx` to the associated row and column indices, respectively. A call of the member funtion `SetIdx(ridx,cidx)` sets the pointers and deletes all matrix entries. The right dimension of the matrix are set later by `SparseMatBuilderCL`, cf. Section 8.2.

8.4 Problem classes

There are several problem classes in DROPS representing different types of partial differential equations. For example, we have problem classes for the Poisson, Stokes and Navier-Stokes problem (one-phase), the level-set equation and the two-phase Stokes and Navier-Stokes problem. For example the class for the two-phase Stokes problem is called `InstatStokes2PhaseP2P1CL`. All problem classes are derived from a common base class `ProblemCL` which contains three objects constituting a problem:

- the domain Ω, given by a multilevel triangulation (`MultiGridCL`),
- the boundary conditions and boundary values, given by a `BndDataT` object,
- the coefficients and right hand-side of the partial differential equation, given by a `CoeffT` object.

`BndDataT` and `CoeffT` are template parameters of the template class `ProblemCL` as their specific structure may vary among different problem types. Their meaning is discussed in the subsequent sections.

A specific problem class usually contains the index descriptions of the applied finite element types and several matrix and vector descriptions. Among the member functions there are `CreateNumbering(...)` procedures for the indices (cf. Section 8.3) and different `Setup...(...)` routines to compute the matrices and the right-hand side vectors constituting the finite element discretization.

Boundary data

The boundary data are represented by a `BndDataCL<BndValT>` object. It contains an array of `BndSegDataCL<BndValT>` objects, one for each boundary segment Σ_j, cf. Section 8.1. Each `BndSegDataCL` object stores the boundary condition and a function pointer for evaluating the corresponding boundary values of type `BndValT`. The choice of the template parameter `BndValT` depends on whether the boundary condition applies to a scalar (`double`) or vector-valued (`Point3DCL`) quantity. The boundary condition of type `BndCondT` can be one of

- `DirBC`, `Dir0BC` for inhomogeneous and homogeneous Dirichlet boundary conditions, respectively,
- `NatBC`, `Nat0BC` for inhomogeneous and homogeneous natural boundary conditions, respectively,

- `Per1BC`, `Per2BC` for periodic boundary conditions denoting corresponding boundaries.

`WallBC` and `OutflowBC` are alias names for `Dir0BC` and `Nat0BC`, respectively.

Coefficients

As an example of how the coefficients of a specific partial differential equation sre represented in the implementation we consider a scalar convection-diffusion problem for the unknown function $u = u(x,t)$,

$$u_t + \mathbf{v}(x,t) \cdot \nabla u - \operatorname{div}(a(x,t)\nabla u) = f(x,t) \qquad \text{in } \Omega \times [t_0, t_f].$$

This type of problem is defined by the problem class `InstatPoissonP1CL`. The corresponding `PoissonCoeffCL` contains the functions $\mathbf{v}(x,t)$, $a(x,t)$ and $f(x,t)$ as static member functions.

For the two-phase flow problem (4.1)-(4.2) the corresponding coefficient class stores quantities such as densities ρ_i and dynamic viscosities μ_i of the phases Ω_i, $i = 1, 2$, the surface tension coefficient τ and the vector of gravitational acceleration $\mathbf{g}$.

8.5 Tools for spatial discretization

In the discretization procedures `Setup...(...)` of the problem classes several sparse matrices representing the discrete differential operators and vectors for the right-hand side have to be constructed. This is done by iterating over all tetrahedra $T \in \mathcal{T}_h$, where for a single tetrahedron T contributions to the matrix and vector entries are computed. These contributions are integrals over T and the integrands are functions which can be defined locally on T, e. g., basis functions or gradients of basis functions.

Grid functions

For representing the integrands and computing the integrals over T we use `LocalP1CL` and `LocalP2CL` objects (for linear and quadratic functions, respectively) and quadrature rules `Quad2CL`, `Quad5CL` (exact for polynomials up to degree 2 or 5, respectively). All these classes have a template parameter `ValT` for the function values and are derived from a common base class `GridFunctionCL<ValT,PointsN>`. This class stores `PointsN` values of type `ValT` which are associated to distinct nodes in a tetrahedron described by barycentric coordinates (`BaryCoordCL`, cf. Section 8.2). For a `LocalP1CL` object these nodes are the 4 vertices of the tetrahedron, for a `LocalP2CL` object the 6 midpoints of the edges are added. For the `Quad...CL` objects the nodes are defined by the quadrature points of the corresponding quadrature rule

Arithmetic operations such as `+`, `-`, `*`, `/` for `GridFunctionCL` objects are defined pointwise. In the same way functions can be applied to `GridFunctionCL` objects using the member function `apply(...)`. Due to inheritence all this functionality is also provided for the derived `LocalP...CL` and `Quad...CL` classes. This is very useful when treating complex integrands like $(\mathbf{u} \cdot \nabla v_j)\, v_i$.

Several variants of `assign(...)` member functions enable the initialization of the `LocalP...CL` and `Quad...CL` objects. Additionally, `LocalP...CL` objects can be evaluated in an arbitrary point $\mathbf{x} \in T$ given by its barycentric coordinates. The `Quad...CL` objects have a member function `quad(...)` which applies the quadrature rule and returns the result of the numerical integration.

Local numberings

A `LocalNumbCL` object is initialized with an index description of index $\mathcal{J}$, the corresponding boundary data object and a tetrahedron T. It collects the numbering of the local degrees of freedom of T according to the index $\mathcal{J}$, cf. Section 8.3. If a degree of freedom is on a boundary it also provides the associated boundary condition and the number j of the corresponding boundary segment Σ_j. Up to now `LocalNumbCL` can only be used for P_2 finite elements.

Integration over interface patches or parts of a tetrahedron

An `InterfacePatchCL` object is initialized by a tetrahedron T and the level set function φ_h given by an P_2-FE `VecDescCL` object. It extracts the `LocalP2CL` object corresponding to φ_h, decides whether $\Gamma_h \cap T \neq$ and provides information about the sign ($\in \{+, -, 0\}$) of each degree of freedom.

The member function `ComputeForChild(i)` computes the planar interface patch $\Gamma_{T'} = \Gamma_h \cap T'$ for the `i`th regular child $T' \in \mathcal{K}(T)$, `i` $= 0, \ldots, 7$. $\Gamma_{T'}$ is represented by the coordinates of its vertices, which are given in terms of barycentric coordinates with respect to the parent T, cf. Fig. 4.1 and Fig. 4.2. Note that for the computation of the patches the regular refinement of T is not really constructed in the sense that geometrical data structures are changed.

After calling `ComputeCutForChild(i)` the member function `quad(...)` can be used to compute the integral over the subdomain $T' \cap \Omega_1$ or $T' \cap \Omega_2$, where the integrand is an arbitrary quadratic function f given by a `LocalP2CL` object. The additional member function `quadBothParts(...)` provides the integrals over the union of the subdomains $(T' \cap \Omega_1) \cup (T' \cap \Omega_2)$.

8.6 Time discretization and coupling

For the one-phase Stokes and Navier-Stokes problem the one-step θ-scheme (cf. section 3.3.1) is represented by the classes `InstatStokesThetaSchemeCL` and `InstatNavStokesThetaSchemeCL`, respectively. Both classes have a template parameter `SolverT` for the type of the solver used in each time step. The computation of one time step is performed by the member function `DoStep(...)`.

For the two-phase Stokes and Navier-Stokes problem we have to consider a coupled system for velocity $\mathbf{u}$, pressure p and level set function φ, cf. section 4.4. During the implementation it turned out that the coupling and time discretization should be combined in one class as they are closely connected to each other. However, the different coupling classes all have a similar structure, thus we decided to derive them from a base class `CouplLevelsetBaseCL` which stores common data members and defines a common abstract interface by means of virtual member functions such as `DoStep(...)`.

For the two-phase Stokes problem the class `CouplLevelsetStokesCL` represents a coupled one-step θ-scheme. For the two-phase Navier-Stokes problem we implemented several methods, for example, the following classes:

- `CouplLevelsetNavStokes2PhaseCL`: generalized θ-scheme as explained in sections 4.3 and 4.4.

- `CouplLsNsBaenschCL`: coupled fractional-step scheme with operator splitting. This method generalizes the method explained in section 3.3.3 to the two-phase flow problem, cf. [2].

In all these methods result in linear systems of equations that have to be solved. All these classes have a template parameter `SolverT` controlling the type of the iterative solver used in each time step.

8.7 Iterative solvers and preconditioners

For the implementation of iterative solvers we tried to use a software design that accounts for the nested hierarchy of the solution methods. For example, the iterative linearization (by a fixed point method) of the one-phase Navier-Stokes equations as described in section 3.4 results in a linear saddle point problem (Oseen equation). For this linear Oseen problem one can use a preconditioned MINRES method. In the preconditioner an inner iterative solver (for example multigrid) may be used. Thus, in this example one has three nested iterative methods: a fixed point linearization method, an Oseen solver and an inner preconditioner. In view of this we use a template mechanism to specify the (inner) solution components as template parameters. This enables an easy plug-in of different solution components to test and compare reasonable combinations of solvers available from the DROPS solver toolbox. Furthermore, this technique assures efficient code since the compiler can perform full code optimization for the template specialization which is known at the moment of compilation.

Example 8.1 As an illustrative example for the template plug-in mechanism we give a piece of code for the definition of a Stokes solver, cf. section 3.5.2:

```
// preconditioner for upper left block preconditioner
typedef SSORPcCL ULPcPcT;
ULPcPcT ULPcPc(...);

// preconditioner for upper left block
typedef PCGSolverCL<ULPcPcT> ULSolverT;
ULSolverT ULsolver( ULPcPc, ...);
typedef SolverAsPreCL<ULSolverT> ULPcT;
ULPcT ULPc( ULsolver);

// Schur complement preconditioner
typedef ISPreCL SchurPcT;
SchurPcT SchurPc( ...);

// Stokes solver
typedef InexactUzawaCL<ULPcT, SchurPcT> StokesSolverT;
StokesSolverT StokesSolver( ULPc, SchurPc, ...);
```

Hence, the object `StokesSolver` represents an inexact Uzawa method. For $\mathbf{Q}_A$ we chose some iterations of an SSOR-preconditioned CG method (`ULPc`) applied to the upper left block of the saddle point matrix. The Schur complement preconditioner $\mathbf{Q}_S$ is given by `SchurPc`.

We emphasize that there is a conceptual difference between solver objects and preconditioner objects. Solver classes are derived from a common base class `SolverBaseCL` storing the tolerance and the maximum number of iterations, i. e., the stopping criterion, as well as the norm of the residual and number of iterations used after the last execution of the solver. Each solver class comprises a member function `Solve(...)` calling the routine of the iterative solver for a given initial guess. In contrast, each preconditioner class contains the analogon `Apply(...)` calling the preconditoner for the initial guess 0.

In the following we list the most important solvers and preconditioners available from the DROPS solver toolbox.

Solvers

Navier-Stokes linearization methods, cf. section 3.4.

- `FixedPtDefectCorrCL`: Algorithm (1) with step length $\omega^k = 1$,
- `AdaptFixedPtDefectCorrCL`: Algorithm 1 with step length ω^k as in (3.19).

Both are template classes where the template parameter `SolverT` determines the type of the Oseen solver. Concerning the iterative solvers for linear equations (Oseen problem and level set equation) we distinguish between Schur complement methods (only for saddle point problems), Krylov subspace methods and multigrid solvers.

Schur complement methods, cf. sections 3.5.2 and 3.5.3.

- `InexactUzawaCL`: Algorithm 3.28 an a variant of this, introduced by Bramble and Pasciak in [**?**], that is implemented in `UzawaCL`
- `SchurSolverCL`: a variant of algorithm (3.21)–(3.23).

Some of the classes provide template parameters `ULPcT`, `SchurPcT` to determine the type of the preconditioners $\mathbf{Q}_A$, $\mathbf{Q}_S$, for the upper left block in the saddle poiint matrix and its Schur complement, respectively.

Krylov subspace methods
For the application of a general Krylov subspace method to the saddle point matrix K one can use the class `BlockMatrixSolverCL<SolverT>` where the template parameter `SolverT` specifies the type of the Krylov solver. The following methods are available in DROPS

- `PMResSPCL`: preconditioned MINRES solver for the Stokes problem.
- `CGSolverCL`, `PCGSolverCL`: CG method and preconditioned variant,
- `MResSolverCL`, `PMResSolverCL`: MINRES method and preconditioned variant.
- `GMResSolverCL`, `GMResRSolverCL`: GMRES and GMRES-Recursive method with left or right preconditioning,
- `BiCGStabSolverCL`: preconditioned BiCGSTAB method,
- `GCRSolverCL`: preconditioned GCR method.

The classes representing preconditioned Krylov subspace methods have a template parameter `PcT` designating the type of the preconditioner.

Multigrid method
The `MGSolverBaseCL` represents a multigrid solver (V-cycle) with a fixed number of smoothing steps. There are two template parameters `SmootherT` and `SolverT` which control the type of the smoother and the coarse grid solver, respectively. The multigrid method requires a hierarchy of linear systems

$$\mathbf{A}_\ell \mathbf{x}_\ell = \mathbf{b}_\ell, \qquad \ell = 0, 1, \ldots, L$$

and prolongations $\mathbf{P}_\ell$ and restrictions $\mathbf{R}_\ell = \mathbf{P}_\ell^T$ to transfer information between a finer and a coarser level. For each level the corresponding system and prolongation matrices $\mathbf{A}_\ell, \mathbf{P}_\ell$ and right-hand side vector $\mathbf{b}_\ell$ are stored in a `MGLevelDataCL` object. The hierarchy of matrices and vectors is represented by the data structure `MGDataCL` which is simply a list of `MGLevelDataCL` objects.
For the smoothers we implemented methods that are suitable for scalar (convection-diffusion) problems such as damped Jacobi and Gauss-Seidel and methods that can be used for Stokes and Oseen equations, for example Vanka- and Braess-Sarazin smoothers.

Preconditioners

The DROPS solver toolbox comprises the preconditioner classes given in the following lists. For a discussion of some of these preconditioners we refer to sections 3.5.2 and 3.5.3.

Iterative method as preconditioners

- `JACPcCL`: one step of the Jacobi preconditioner,
- `GSPcCL`, `SGSPcCL`: one step of the Gauss-Seidel or symmetric Gauss-Seidel preconditioner,
- `SSORPcCL`, `MultiSSORPcCL`: one or multiple steps of the SSOR preconditioner,
- `MGPreCL`: fixed number of multigrid V-cycles with SSOR smoothing,
- `DummyPcCL`: no preconditioning

For the Jacobi and Gauss-Seidel preconditioners there exist variants which can be used as smoother for the multigrid solver.

The wrapper class `SolverAsPreCL` enables the use of a solver object as a preconditioner. That means that the `Apply(...)` member function of the wrapper class calls the `Solve(...)` member function of the solver class with initial guess zero. This mechanism is used in Example 8.1 in the definition of the preconditioner for the upper left block, `ULPc`, which wraps the solver object `ULsolver`.

Schur complement preconditioners $\mathbf{Q}_S$

- `ISPreCL`: the Schur complement preconditioner (3.32) where $\mathbf{M}^{-1}$ and $\mathbf{T}_h^{-1}$ are replaced by one step of the SSOR preconditioner applied to the corresponding pressure matrices,
- `ISNonlinearPreCL`: the same Schur complement preconditioner, but with $\mathbf{M}^{-1}$ and $\mathbf{T}_h^{-1}$ replaced by some iterations of a Krylov subspace method which can be chosen by means of a template argument,
- `ISBBTPreCL`: variant of the Schur complement preconditioner (3.36).

The `DiagBlockPreCL` is used in combination with `BlockMatrixSolverCL` solvers. It combines a preconditioner $\mathbf{Q}_A$ for the upper left block with a preconditioner $\mathbf{Q}_S$ for the Schur complement yielding the diagonal block preconditioner $\tilde{\mathbf{K}}$ as in (3.33). If this preconditioner is used in combination with MINRES it must be symmetric positive definite. If it is combined with a Krylov subspace method for nonsymmetric problems (for example, GMRES) this SPD property is not required.

8.8 Input and output

In this section we describe input and output interfaces for different types of data.

Numerical data

Vectors and sparse matrices can be saved to and restored from files by using the input and output stream operators, `>>` and `<<`, implemented for `VectorCL` and `MatrixCL` objects. The matrix format is MATLAB-compatible which is very useful for computing condition numbers or the spectrum of a matrix.

Geometrical data

The initial triangulation $\mathcal{T}_0$ can be read from a mesh file generated with the mesh generator GAMBIT [44]. To construct the corresponding multilevel triangulation a `ReadMeshBuilderCL` object containing the mesh file name is passed to the constructor of the `MultiGridCL` object. Here the concept of the `MGBuilderCL` class is applied, cf. Section 8.1, from which `ReadMeshBuilderCL` is derived. Other input file formats can be implemented by adding further ancestors of `MGBuilderCL`.

For the input and output of a *hierarchy* of triangulations $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ we use a self-defined file format. For saving a `MultiGridCL` object representing a multilevel triangulation we use a software technique called serialization. For this reason the class representing this task is called `MGSerializationCL`. The deserialization is done by the class `FileBuilderCL`, which is an ancestor of `MGBuilderCL` and is passed to the constructor of `MultiGridCL`. It reads the files written out before by a `MGSerializationCL` object and recreates the corresponding `MultiGridCL` object.

In this way, a simulation run that has stopped can be restarted from the last time step where a serialized multilevel triangulation was saved to the file system. In a first step the geometrical data is deserialized from the file system using the class `FileBuilderCL`. After that the vectors representing the numerical solutions are restored by means of the class `ReadEnsightP2SolCL`, see the subsequent section.

Visualization

For 3D visualization purposes we mainly use the software package Ensight [43]. The class `EnsightP2SolOutCL` writes out the geometrical information (tetrahedra and coordinates of the vertices) and the numerical solutions ($\mathbf{u}_h$, p, φ evaluated in all P_2 degrees of freedom) using a specific Ensight file format. This format can also be read by other visualization packages such as ParaView [70].

The class `ReadEnsightP2SolCL` restores the vectors $\vec{\mathbf{u}}$, $\vec{p}$ and $\vec{\varphi}$ from the files written out by the class `EnsightP2SolOutCL`. However, this only works properly if the multilevel triangulations at the time of storing and restoring are the same.

There are interfaces to some other visualization tools as well.

- `GeomMGOutCL`, `GeomSolOutCL` for visualization of geometry and numerical solution with Geomview [47],
- `TecPlotSolOutCL`, `TecPlot2DSolOutCL` for visualization of geometry and numerical solution (in 3D or on a 2D cut plane, respectively) with TecPlot [89],
- `MapleMGOutCL`, `MapleSolOutCL` for visualization of geometry and numerical solution with Maple.

Chapter 9

Parallelization

In this chapter we consider the main concepts underlying the parallelization of DROPSfor *distributed memory machines* by means of a *message passing interface* (MPI). *Shared memory parallelization* by means of OpenMP [66] has also been applied to some parts of DROPS, cf. [14] for a description of the parallelized routines and some benchmark computations. Both parallelization concepts can be combined when using multicore processors which are connected by a high-speed network. For the parallelization of DROPS we pursue such a hybrid parallelization approach due to the growing importance of multicore architectures.

In Section 9.1 we present a data distribution format for the geometrical data and, based on this, we also derive a distribution format for the numerical data. In Definition 7 below the geometrical data distribution format will be made mathematically precise by a formal specification of a so-called *admissible hierarchical decomposition*. This data distribution format is such that the following holds:

1. Let $T \in \mathcal{G}_k$ be an element from the hierarchical surplus on level k, cf. Definition 6. Then T is stored on one processor, say p, as a so-called master element. In certain cases (explained below) a ghost copy of T is stored on *one* other processor, say q.

2. The children of T (if they exist) are all stored as masters either on processor p or, if T has a ghost copy, on processor q. For $T \in \mathcal{G}_k$, $k > 0$, the parent of T or a copy of it is stored on processor p.

For the multilevel refinement algorithm a crucial point is that for a tetrahedron T one needs information about all children of T, cf. [27, 1]. Due to property 2 this information is available on the local processor (p or q) *without communication*. The first property shows that in a certain sense the overlap of tetrahedra is small.

In a parallel run of a simulation the computational load has to be distributed more or less equally among the processors. Hence, an adaptive finite element solver has to be combined with dynamic load balancing and data migration between the processors. This is the topic of Section 9.2.

The main results concerning the admissible hierarchical decomposition, the parallel multilevel refinement method and the load balancing strategy can be summarized as follows:

- An admissible hierarchical decomposition has the desirable properties 1 (small storage overhead) and 2 (data locality) from above. This result is given in Section 9.1.

- The application of the parallel refinement algorithm to an admissible hierarchical decomposition is well-defined and results in an admissible hierarchical decomposition. This is proved in [1].

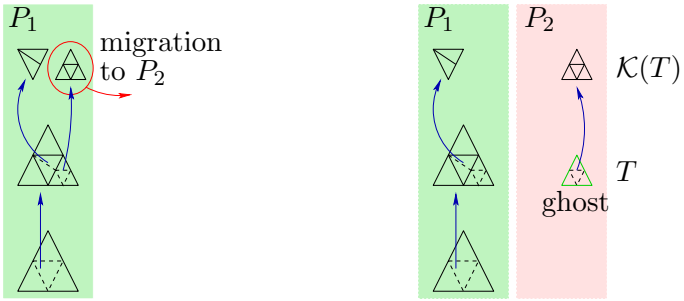

Figure 9.1: Ghost elements are required to represent links between parents and their children since pointers across memory boundaries are not allowed for distributed memory machines.

- Given an admissible hierarchical decomposition one can apply a suitable load balancing and data migration algorithm such that after data migration one still has an admissible hierarchical decomposition. We comment on this in Section 9.2.

9.1 Data distribution

Distribution of geometrical data: admissable hierarchical decomposition

Let the sequence $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ of triangulations be a multilevel triangulation and $\mathcal{H} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$ the corresponding hierarchical decomposition. In this section we introduce a particular format for the distribution of the tetrahedra in $\mathcal{H}$ among processors on a parallel machine. We assume that the processors are numbered by $1, \ldots, P$.

For the set of elements in the hierarchical surplus on level k that are stored on processor p we introduce the notation

$$\mathcal{G}_k(p) := \{\, T \in \mathcal{G}_k : T \text{ is stored on processor } p \,\}$$

and we define

$$\mathcal{H}(p) := \big(\mathcal{G}_0(p), \ldots, \mathcal{G}_J(p)\big).$$

Note that in general $\mathcal{H}(p)$ is not a hierarchical decomposition (in the sense of Definition 7). The sequence

$$\tilde{\mathcal{H}} = (\mathcal{H}(1), \ldots, \mathcal{H}(P)) \tag{9.1}$$

is called a *distributed hierarchical decomposition* (corresponding to $\mathcal{H}$).

In general the intersection $\mathcal{G}_k(p) \cap \mathcal{G}_k(q)$, $p \neq q$, may be nonempty. Note that such an *overlapping distribution of the elements is necessary*, due to the fact that parents and children are linked by pointers. Consider, for example, the situation depicted in Figure 9.1 where a parent T and its child $T' \in \mathcal{K}(T)$ are stored on different processors, say 1 and 2. Since pointers from one local memory to another are not allowed in a distributed memory setting, we have to use a copy to realize this pointer. One could store a copy of T on processor 2 to represent the link between T and T' as a pointer on processor q. If one does not allow such ghost copies, all ancestors and descendants of a tetrahedron must be on the same processor. This would cause very coarse data granularity, poor load balancing and hence low parallel efficiency.

For each level k and processor p we introduce a set of *master elements*, $\mathcal{M}_k(p) \subset \mathcal{G}_k(p)$, and a set of *ghost elements*, $\mathcal{G}_k(p) \subset \mathcal{G}_k(p)$. In the formulation of the conditions below we use the conventions $\mathcal{K}(T) := \emptyset$ if $\mathsf{status}(T) = \mathsf{NoRef}$ and $\mathcal{M}_{J+1}(p) := \emptyset$.

We now formalize the conditions on data distribution as follows.

Definition 7 (Admissible hierarchical decomposition) The distributed hierarchical decomposition $\tilde{\mathcal{H}}$ is called an *admissible hierarchical decomposition* if for all $k = 1, \ldots, J$ the following conditions are fulfilled:

(A1) **Partitioning of $\mathcal{G}_k(p)$:** The sets of masters and ghosts form a disjoint partitioning of $\mathcal{G}_k(p)$:

$$\forall\, p \quad \mathcal{M}_k(p) \cup \mathcal{G}_k(p) = \mathcal{G}_k(p) \quad \text{and} \quad \mathcal{M}_k(p) \cap \mathcal{G}_k(p) = \emptyset$$

(A2) **Existence:** Every element from $\mathcal{G}_k$ is represented as a master element on level k:

$$\mathcal{G}_k = \bigcup_{p=1}^{P} \mathcal{M}_k(p)$$

(A3) **Uniqueness:** Every element from $\mathcal{G}_k$ is represented by at most one master element on level k:

$$\forall\, p_1, p_2 : \qquad \mathcal{M}_k(p_1) \cap \mathcal{M}_k(p_2) \neq \emptyset \;\Rightarrow\; p_1 = p_2$$

(A4) **Child–parent locality:** A child master element and its parent (as master or ghost) are stored on the same processor:

$$\forall\, p \quad \forall\, T \in \mathcal{G}_k \quad \forall\, T' \in \mathcal{K}(T) : \qquad T' \in \mathcal{M}_{k+1}(p) \;\Rightarrow\; T \in \mathcal{G}_k(p)$$

(A5) **Ghosts are parents:** Ghost elements always have children:

$$\forall\, p \quad \forall\, T \in \mathcal{G}_k(p) : \qquad \mathcal{K}(T) \neq \emptyset$$

(A6) **Ghost–children locality:** A ghost element and its children are stored on the same processor:

$$\forall\, p \quad \forall\, T \in \mathcal{G}_k(p) : \qquad \mathcal{K}(T) \subset \mathcal{M}_{k+1}(p)$$

Remark 1 Consider a consistent initial triangulation $\mathcal{T}_0 = \mathcal{G}_0$ with a nonoverlapping distribution of the tetrahedra: $\mathcal{G}_0(p) \cap \mathcal{G}_0(q) = \emptyset$ for all $p \neq q$. In this case all tetrahedra can be stored as masters and there are no ghosts. Then the distributed hierarchical decomposition $\tilde{\mathcal{H}} = ((\mathcal{G}_0(1)), \ldots, (\mathcal{G}_0(P)))$ is obviously admissible.

Two elementary results are given in the following lemma.

Lemma 1 *Let $\tilde{\mathcal{H}}$ as in* (9.1) *be a distributed hierarchical decomposition. The following holds:*

1. *If the conditions* (A3), (A5) *and* (A6) *are satisfied then for any element from $\mathcal{G}_k$ there is at most one corresponding ghost element:*

$$\forall\, T \in \mathcal{G}_k \quad \forall\, p, q : \qquad T \in \mathcal{G}_k(p) \cap \mathcal{G}_k(q) \;\Rightarrow\; p = q$$

2. *If the conditions* (A1), (A2), (A3), (A4) *and* (A6) *are satisfied then all children of a parent are stored as master elements on one processor:*

$$\forall\, T \in \mathcal{G}_k \quad \exists\, p : \qquad \mathcal{K}(T) \subset \mathcal{M}_{k+1}(p)$$

A proof of this result is given in [1].

In [1] a parallel refinement (and coarsening) algorithm is presented which is based on an admissible hierarchical decomposition and is suitable for distributed memory machines. It uses the DDD package [64, 95] for the management of the distributed tetrahedra, faces, edges and vertices. For a given input-multilevel triangulation the parallel method produces the same output-multilevel triangulation as the serial method presented in [27, 28]. In this sense the "computational part" of the algorithm is not changed. It is proven that the application of the parallel refinement algorithm to an admissible hierarchical decomposition is well-defined and results in an admissible hierarchical decomposition.

Remark 2 Let $T \in \mathcal{M}_k(p)$ be a parent master element. From the second result in Lemma 1 and (A4) it follows that either all children are masters on the same processor p as T, or they are masters on some other processor q. In the latter case, the element T has a corresponding ghost element on processor q. Due to this property, in the parallel refinement algorithm we use the strategy:

- If a parent tetrahedron T has a ghost copy then operations that involve children of T are performed on the processor on which the ghost and the children are stored.

From condition (A4) it follows that a child master element has its parent (as ghost or as master) on the same processor. Therefore we use the strategy:

- Operations that involve the parent of T are performed on the processor on which the master element of T and its parent are stored.

The first result in Lemma 1 shows that every $T \in \mathcal{H}$ has at most one ghost copy. Moreover, due to (A5) all leaves ($T \in \mathcal{T}_J$) have no ghost copies. In this sense the overlap of tetrahedra between the processors is small.

Distribution of numerical data

Let $\vec{x} \in \mathbb{R}^N$ a vector and $A \in \mathbb{R}^{N \times N}$ a (sparse) matrix. The numbering $\mathcal{J} = \{1, \ldots, N\}$ is associated with certain degrees of freedom of the hierarchical decomposition $\mathcal{H}$. Based on the distributed hierarchical decomposition $\tilde{\mathcal{H}}$ we will define a corresponding distribution of the numerical data $\vec{x}$ and A. For this purpose we first introduce the notion of a domain decomposition.

Definition 8 (Domain decomposition) Let $\mathcal{H}$ be a hierarchical decomposition and $\tilde{\mathcal{H}}$ its admissible distribution among the processors. Due to the conditions (A2) and (A3) every tetrahedron $T \in \mathcal{H}$ can be assigned a unique processor on which T is stored as a master element. In other words, we have a well-defined function $\mathsf{master} : \mathcal{H} \to \{1, \ldots, P\}$ that is given by

$$\mathsf{master}(T) = p \quad \Leftrightarrow \quad T \in \mathcal{M}_{\ell(T)}(p).$$

For $0 \le j \le J$ and $1 \le p \le P$ we define

$$\mathcal{T}_j(p) := \{\, T \in \mathcal{G}_j : \mathsf{master}(T) = p \,\} \qquad \text{and} \qquad \Omega_j(p) := \bigcup_{T \in \mathcal{T}_j(p)} T.$$

Then for each $0 \le j \le J$ the sequence $(\mathcal{T}_j(1), \ldots, \mathcal{T}_j(P))$ is a partition of the triangulation $\mathcal{T}_j$ (due to (A2), (A3)) and is called the *domain decomposition of level* j corresponding to the admissible hierarchical decomposition $\tilde{\mathcal{H}}$.

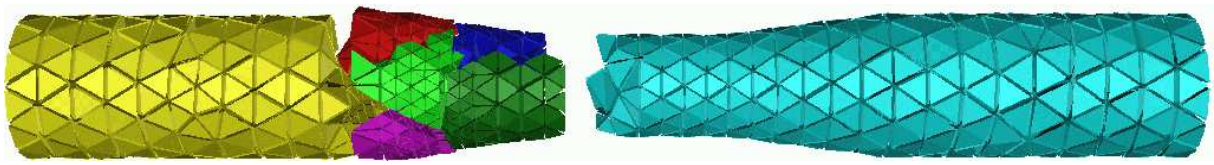

Figure 9.2: Domain decomposition for $P = 8$ processors.

Figure 9.2 shows a domain decomposition for $P = 8$ processors.
A domain decomposition of level j automatically induces a distribution of the numerical data on level j. Without loss of generality we assume that the (global) numbering $\mathcal{J} = \{1, \ldots, N\}$ is associated with the finest level J. Let $\mathcal{J}(p) = \{1, \ldots, N_p\}$ be a (local) numbering of the degrees of freedom of the local triangulation $\mathcal{T}_J(p)$ on processor p, $1 \leq p \leq P$. Then the relation between a local number $i \in \mathcal{J}(p)$ and its global counterpart $j \in \mathcal{J}$ is given by the coincidence matrix $I_p \in \mathbb{R}^{N \times N_p}$,

$$(I_p)_{i,j} := \begin{cases} 1 & \text{if degree of freedom with global number } j \in \mathcal{J} \text{ exists} \\ & \text{on processor } p \text{ with local number } i \in \mathcal{J}(p), \\ 0 & \text{else.} \end{cases}$$

Degrees of freedom which are located on multiple processors form the so called *processor boundary*.

Definition 9 (Accumulated and distributed storage) For a (global) vector $\vec{x} \in \mathbb{R}^N$ the sequence

$$\vec{x}_a = (I_1\vec{x}, \ldots, I_P\vec{x}) \in \mathbb{R}^{N_1} \times \ldots \times \mathbb{R}^{N_P}$$

is called the corresponding *accumulated vector*. That means that for unknowns on a processor boundary each adjacent processor stores the same global value.

The sequence $\vec{x}_d = (\vec{x}_1, \ldots, \vec{x}_P)$ of vectors $\vec{x}_p \in \mathbb{R}^{N_p}$ is called the *distributed vector* corresponding to $\vec{x}$, if

$$\vec{x} = \sum_{p=1}^{P} I_p^T \vec{x}_p.$$

In this case the global value of an unknown on a processor boundary is the sum of all local values stored on the adjacent processors. The same holds for entries of a *distributed matrix* $A_D = (A_1, \ldots, A_P)$ with

$$A = \sum_{p=1}^{P} I_p^T A_p I_p.$$

Remark 3 (Computation of distributed stiffness matrix) For a stiffness matrix $A \in \mathbb{R}^{N \times N}$ the local distributed matrix $A_p \in \mathbb{R}^{N_p \times N_p}$ coincides with the stiffness matrix corresponding to the subdomain $\Omega_J(p)$ with triangulation $\mathcal{T}_J(p)$. Thus the local matrices M_p can be set up independently by the different processors $p = 1, \ldots P$ without any communication. Furthermore, the parallelization of the `Setup` routines (cf. Section 8.4) is a trivial task.

The conversion of a distributed into an accumulated vector is achieved by summing up the vector entries on processor boundaries which requires communication between adjacent processors. Obviously, the conversion in the other direction is not unique. For computing the matrix-vector

multiplication $\vec{y} = A\,\vec{x}$ we use the accumulated storage $\vec{x}_a$ as input and obtain the result $\vec{y}_D$ in a distributed fashion:

$$A\,\vec{x} = \left(\sum_{p=1}^{P} I_p^T A_p I_p \right) \vec{x} = \sum_{p=1}^{P} I_p^T \underbrace{A_p (I_p \vec{x})}_{=:\vec{y}_p} = \vec{y}.$$

Hence, the computation of the matrix-vector multiplication does not require any communication. The scalar product of two vectors $\vec{x}, \vec{y}$ can be computed efficiently if one of them is stored accumulated, for example $\vec{x}_a$, and the other one distributed, $\vec{y}_d$. Then the computation of

$$(\vec{x},\, \vec{y}) = \vec{x}^T \sum_{p=1}^{P} I_p^T \vec{y}_p = \sum_{p=1}^{P} (I_p \vec{x})^T \vec{y}_p = \sum_{p=1}^{P} (I_p \vec{x},\, \vec{y}_p)$$

only requires the global summation of P real numbers (`MPI::AllReduce(...)`).

9.2 Distribution of work load

In the simulation of a rising bubble, for example, the multilevel triangulation $\mathcal{M}$ will change as the refinement zone is moving upwards following the bubble geometry. Hence, the distributed hierarchical decomposition $\tilde{H}$ and the numerical data have to be redistributed from time to time to ensure a reasonable balance of the computational load. Otherwise the situation may occur that almost all unknowns are stored on one processor, say p, while the others only have to solve problems of small size. On the one hand this leads to an inefficient usage of the overall memory. On the other hand runtime scalability severely decreases since all processors have to wait at synchronization points such as `MPI::AllReduce(...)` until processor p has finished its work.

The challenge of the so-called *load balancing* is to find a mapping

$$m : \mathcal{T} \to \{1, \ldots, P\}$$

describing the distribution of the tetrahedra among the processors such that

a) all processors have almost the same number of tetrahedra and

b) the corresponding processor boundary is as small as possible.

This problem statement is equivalent to a *graph partitioning problem* which will be stated in Definition 11. For this reason, m is also called a *partitioning* of $\mathcal{T}$. We now introduce the notion of a weighted dual graph.

Definition 10 (Weighted dual graph) For a triangulation $\mathcal{T}$ the corresponding *dual graph* $G(\mathcal{T}) = (V, E)$ is given by the node set $V = \mathcal{T}$ and the edge set $E \subset \mathcal{T} \times \mathcal{T}$, where $(T_1, T_2) \in E$ iff the tetrahedra T_1, T_2 share a common face.

By introducing *weight functions* $\alpha : V \to \mathbb{R}_+$ for nodes and $\beta : E \to \mathbb{R}_+$ for edges of the graph the computational load $\alpha(v_T)$ of the corresponding tetrahedron T and the amount of communication $\beta(e_F)$ for the corresponding face F can be described. $G_w(\mathcal{T}) = (V, E, \alpha, \beta)$ is called a *weighted dual graph.*

Figure 9.3 shows a 2D example for a dual graph. For a subset $\tilde{V} \subset V$ we define $\alpha(\tilde{V}) := \sum_{v \in \tilde{V}} \alpha(v)$ corresponding to the total load of $\tilde{V}$. For a given partitioning m the set

$$E_{\text{cut}}(m) := \{\, (T_1, T_2) \in E :\, m(T_1) \neq m(T_2) \,\}$$

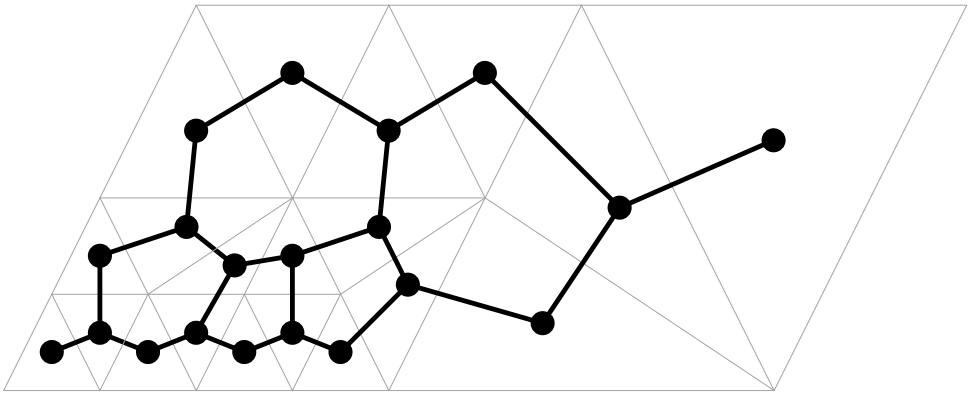

Figure 9.3: Dual graph for 2D triangulation.

corresponds to the faces forming the processor boundary where communication takes place.

The graph partitioning problem is given by the following definition:

Definition 11 (Generalized graph partitioning problem) For a constant $C > 1$ and a given weighted dual graph (V, E, α, β) find a partitioning $m : V \to \{1, \ldots, P\}$ such that

$$\mathrm{cost}_{\mathrm{comm}}(m) := \sum_{e \in E_{\mathrm{cut}}(m)} \beta(e) \quad \to \quad \min$$

and

$$\alpha(V_p) \leq C \frac{\alpha(V)}{P}$$

with $V_p := m^{-1}(p)$.

The graph partitioning problem belongs to the class of NP-hard problems, in this sense an *optimal* partitioning cannot be computed efficiently. Nevertheless, there are a couple of heuristic approaches with polynomial runtime yielding reasonable results. For a survey on this topic we refer to [37]. We use the package ParMETIS [72] which realizes a parallel multilevel graph partitioning algorithm described in [59].

Based on a partitioning m computed by a graph partitioning tool the tetrahedra and numerical data are rearranged among the processors. This phase is called *data migration*. To obtain again an admissible hierarchical decomposition after the migration phase we have to ensure that the properties (A1)–(A5) hold. In particular all children of a common parent have to stay together as masters on a single processor, cf. Lemma 1. Thus in the following we give a definition for a reduced dual graph, where the children of a common parent are represented by a single multi-node. For this purpose we introduce a map

$$P : \bigcup_{k=0}^{J} \mathcal{G}_k \to \bigcup_{k=0}^{J-1} \mathcal{G}_k$$

from a tetrahedron $T \in \mathcal{G}_k$ to its parent tetrahedron $P(T) \in \mathcal{G}_{k-1}$, $k = 1, \ldots, J$, with the convention $P(T) = T$ for all $T \in \mathcal{G}_0$. For $T \in \mathcal{T}$ we define the corresponding equivalence class

$$[T]_P := \{\, S \in \mathcal{T} : \, P(S) = P(T) \,\}.$$

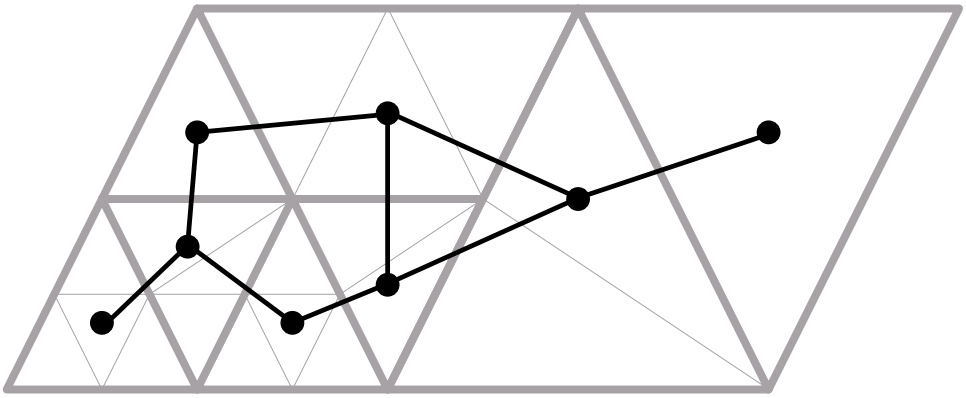

Figure 9.4: Reduced dual graph for 2D triangulation.

Definition 12 (Reduced dual graph) For a triangulation $\mathcal{T}$ let $G_w(\mathcal{T}) = (V, E, \alpha, \beta)$ be the corresponding weighted dual graph. The *reduced dual graph* $G'_w(\mathcal{T}) = (V', E', \alpha', \beta')$ is given by the reduced node set

$$V' := \{\, [T]_P : T \in \mathcal{T} \,\}$$

inducing the reduced edge set

$$E' := \{\, (v'_1, v'_2) : \quad \exists\, v_1 \in v'_1, v_2 \in v'_2 : \ (v_1, v_2) \in E \,\} \setminus \{\, (v', v') : v' \in V' \,\}.$$

The weight functions α', β' are given by

$$\alpha'(v') := \sum_{v \in v'} \alpha(v),$$
$$\beta'((v'_1, v'_2)) := \sum_{e \in E \cap (v'_1 \times v'_2)} \beta(e).$$

Figure 9.4 shows the reduced dual graph corresponding to the dual graph given in Figure 9.3. The tetrahedra forming a multi-node are surrounded by a bold frame. Note that the dual graph $G(\mathcal{T})$ in Figure 9.3 has 20 nodes and 24 edges whereas the reduced dual graph $G'(\mathcal{T})$ in Figure 9.4 has only 8 nodes and 9 edges.

After computing a load balancing partitioning $m' : V' \to \{1, \ldots, P\}$ of the reduced dual graph $G'_w(T)$, for the data migration we use an migration algorithm described in [52]. The migration of the tetrahedra is carried out by means of the DDD package. After the migration for the new distributed hierarchical decomposition $\tilde{\mathcal{H}}$ the property

$$\mathsf{master}(T) = m'([T]_P)$$

holds. In [52] it is shown that for an admissible input hierarchical decomposition the distributed hierarchical decomposition after the migration is again admissible.

Remark 4 (Migration of numerical data) If a tetrahedron T is moving from one processor to another, also certain vector entries corresponding to the degrees of freedom on T have to be migrated. The valid migration of numerical data is a delicate task and will not be discussed in this thesis.

9.3 Current status and outlook

The parallel refinement algorithm and load balancing strategy described in [1] served as a starting point for the parallelization of DROPS. The work on parallelization is done at the Chair of Scientific Computing, RWTH Aachen University.
Currently the parallel version of DROPS can solve the following problem classes

1. Poisson problems
2. Stokes problems
3. Navier-Stokes problems
4. Two-phase flow problems

In all these cases we can handle adaptive grids that change during the time integration.

Current status

In order to be able to solve these problems with DROPS on a distributed memory computer the following methods are parallelized. We use the structure in Figure 8.1 to discuss the parallelized algorithms and data structures and the algorithms and data structures that remain unchanged.

The *refinement algorithm and load balancing strategy* have been parallelized. This implies that the underlying data structure of multilevel tetrahedral grids is distributed. The refinement algorithm and the load balancing algorithm have been extended such that these can handle degrees of freedom on hierarchical tetrahedral grids.

The initial triangulation $\mathcal{T}_0$ can be read from a mesh file generated by the mesh generator GAMBIT [44] and is then distributed among all processors. The geometry as well as the finite element solutions can be transfered to Ensight- [43], Geomview- [47] or to VTK-format [80].

The *setup routines* for matrices and right hand sides of the discretized PDEs were easy to parallelize, cf. section 9.1. The data structures for finite elements, quadrature, index clases, etc. of the sequential version of DROPS are reused in the parallel version with only minor changes.

In an adaptive setting it is essential to be able to perform *interpolation* operations on finite element functions between different grids. Changing the triangulation or performing load balancing operations has to be done carefully in order to keep access to degrees of freedom that are required for the transfer (interpolation) of FE-functions to another triangulation. In the current parallel version all algorithms to create and change adaptive grids in time can handle P_1- and P_2-FE-functions, cf. section 8.3.

Within the parallel solvers it is neccessary to *transform vectors from the distributed storage format into the accumulated one*, see Definition 9. Efficient routines for such transformation operations have been implemented.

A large part of the overall computing time is consumed by *iterative solvers for linear systems.* Parallel versions of CG, GMRES, GCR, QMR and BiCGStab are implemented. For preconditioning these solvers can be combined with a parallel Jacobi- or a blocked SSOR-method as well as with iterative solvers in the form of preconditioners.

In order to solve Stokes- or Oseen type problems one can use a parallel Krylov subspace method. Furthermore, a parallel version of the inexact Uzawa algorithm is available. cf. sections 3.5.2-3.5.3.

To treat the *nonlinearity* in the discretized the Navier-Stokes-equations we use a parallel version of the fixed point defect correction method with step size control, cf. algorithm 1 in section 3.4.

The *time integration methods* methods available in the sequential DROPS version are easy to parallelize. Such parallel variants are available.

Parallelization of the *fast marching method* (FMM), cf. section 4.2.6, is a delicate task, because information from neighboring tetrahedra is needed to update degrees of freedom. This information is used in each update step of the FMM and may be stored on a neighbor processor. This implies a large number of small communications and thus can cause bad parallel efficiency. Due to the inherent sequential structure of this algorithm there is no effective way of parallelizing this algorithm with only minor changes. We have implemented in the current version a (with respect to parallel efficiency) suboptimal strategy in which the update phase of the FMM is realized on a single processor. It turns out that this is still acceptable for the problem sizes and numbers of processors that we have considered so far.

For the *decoupling* (in each time step) between the level set and the Navier-Stokes subproblems in a two-phase flow problem the sequential methods could be parallelized with only minor changes.

Outlook on parallelization

The following concerning the parallelization of DROPS are planned.

- **Hybrid parallelization**. In order to use clusters of multicore architectures a hybrid parallel version of DROPS will be developed. This will be done by combining OpenMP and MPI, to get better parallel efficiency on these high-performance computers. The planned shared memory parallelization will be based on the work that has been done by the Center for Computation and Communication at RWTH [14].

- **Improving parallel efficiency of linear solvers**. The most time-consuming task in solving two-phase flow problems is solving multiple linear systems of equation. We want to use a large number of processors for solving such problems in order to be able to store such large data sets and to have relatively short computing times. Synchronization points will become a bottleneck for the parallel efficiency. Therefore new algorithms and modified Krylov subspace solvers will be developed and implemented in DROPS. A good option may be to apply so-called s-step methods in these solvers. Furthermore, more parallel preconditioners have to be developed and implemented.

- **Load balancing**. The current version of the load balancing technique tries to

  a) uniformly distribute the tetrahedra of a given level (in most cases of the finest triangulation) and

  b) keep the number of faces on the processor boundary as small as possible.

  Both goals are not optimal. In the case of a) the number of tetrahedra does not necessarily represent correctly the amount of work each processor has to do. For example, Dirichlet-boundary conditions do not create degrees of freedom on all vertices of tetrahedra at the domain boundary and/or the XFEM may introduce extra degrees of freedom in a certain tetrahedron. With respect to b) we note that just considering faces may not be a correct measure for the amount of communication between processors. It is known,

that hyper graphs can often model the communication in a better way, cf. [40]. Thus the load balancing algorithm can be improved by a better node weight function $\alpha'(v')$, cf. Definition 12 and by the use of hyper graphs to compute a graph partitioning.

- **Reparametrization of the level-set function**. The currently implemented strategy to reparametrize the level-set function by performing the update phase in the fast-marching algorithm on a single processors is not efficient for a large number of processors (Amdahl's law). A better parallel variant of the FMM has to be developed and implemented.

- **XFEM**. The present parallel version of DROPS cannot handle extended finite element spaces. As is known from the sequential version these spaces result in much smaller discretization errors as the standard finite element spaces. Therefore it is important to be able to use the XFEM technique in the parallel code, too.

Part III

Examples of implementations in DROPS

In this part we present examples of implementations of a few one- and two-phase flow problems in DROPS.

Chapter 10

How to get started

In this chapter we describe how the DROPS package can be obtained and installed.

10.1 How to obtain the code

For access to the DROPS CVS server, it may be necessary to register at TIM (Tivoli Identity Manager) at the Computing Center RWTH Aachen University (see, for details, http://www.rz.rwth-aachen.de/). Then the following should be done:

- Download the register form, fill-in and send it to Computing Center RWTH Aachen University. After that you will receive the login account <User-ID> by TIM and the password.
- Login by TIM and unlock "Hochleistungsrechnen RWTH Aachen".
- Write a mail to <terboven@rz.rwth-aachen.de>[1] and ask to record the user <User-ID> for the DROPS-CVS repository.
- Configure the computer cluster ssh so that not every time the password is demanded. Do the following:
  - Login at the computer cluster at Computing Center RWTH Aachen University
    ```
    ssh User-ID@cluster.rz.rwth-aachen.de
    ```
  - Create the *.ssh* subdirectory
    ```
    mkdir .ssh
    ```
  - Create (with the vi text editor) the *authorized_keys* file
    ```
    vi authorized_keys
    ```
    and copy the contents of the *.ssh/id_rsa.pub* file from you computer into this file.
- Configure the DROPS-CVS repository on your computer
  ```
  export CVS_RSH=ssh

  export CVSROOT=<User-ID>@cluster.rz.rwth-aachen.de:/home/drops/CVS_REPOSITORY
  ```

If the registration is completed successfully, perform the following actions:

[1] This address may change. In this case ask the maintainers of DROPS about the current contact person.

- Create a work directory for installation

  ```
  mkdir your_installation_directory
  ```

  and checkout *drops* from DROPS-CVS repository by

  ```
  cvs co drops
  ```

  Alternatively, you can use a GUI frontend to CVS such as *tkcvs*:

  ```
  tkcvs
  ```

  [Choose the option "Browse Modules" in the menu-icon "File"]

  [In the new window "Module Browser" choose
  the module "drops" and check out them]

  Subdirectory *drops* will be created in your current directory.

- Go to this new subdirectory *drops*:

  ```
  cd drops
  ```

10.2 Compilation of DROPS

You may have to change system-dependent settings. They are contained in the file *drops.conf*. Use a text editor, for example nedit to change the file *drops.conf*.

```
nedit drops.conf
```

[change the file *drops.conf*]

Set your architecture and compiler (option ARCH) to one of the following: to one of

- ARCH = LINUX for gcc compiler;
- ARCH = INTEL for icc (Intel C++) compiler;
- ARCH = SGI for CC compiler in OS Irix;
- ARCH = SUN for CC compiler in OS Solaris,

and correct the translator call (option ARCH_CXX) and its options (option ARCH_CXXFLAGS) in the file *arch/ARCH/mk.conf*. For example, for ARCH = LINUX:

```
cd arch/LINUX
```

```
nedit mk.conf
```

[change the file *mk.conf*]

`cd ../..` (go back into the root directory *drops*).

Now that you have changed the files *drops.conf* and *arch/ARCH/mk.conf*, run the *make* utility at the shell prompt in the current directory with the target *dep*:

```
make dep
```

This will create the corresponding file dependency needed for compiling DROPS.

10.3 Building the binary file

There are four main subdirectories with the program examples for

- a Poisson problem (subdirectory POISSON) ;
- a one-phase Stokes problem (subdirectory STOKES) ;
- a one-phase Navier-Stokes problem (subdirectory NAVSTOKES);
- a two-phase Navier-Stokes problem (subdirectory LEVELSET) .

As an example, go to the subdirectory *navstokes*:

```
cd navstokes
```

Now run the *make* utility at the shell prompt in the current directory:

```
make <name>
```

Here <name> may be one of the following:

- *nsdrops* for an stationary Navier-Stokes problem;
- *insdrops* for an instationary Navier-Stokes problem;
- *insadrops* for an instationary Navier-Stokes problem with adaptive grid refinement;
- *drivcav* for the driven cavity problem described in Chapter 11.

For example, we continue with the option *drivcav*:

```
make drivcav
```

10.4 Running DROPS

If the *make* finished successfully, the executable file *drivcav* has been constructed. The simplest way to run the program is a command call such as

```
./drivcav 1e-7 200 0.1 100 1 200 0 100 0.1 0 1
```

A detailed description of this example is given in Section 11.2.4.

Another way to run a program in DROPS is the usage of the parameter file <name>.*param*. For example, to run the program *risingBubbleAdap* from the $\sim$ */drops/levelset* subdirectory, in which the simulation of a rising bubble is implemented, we call

```
./risingBubbleAdap risingBubbleAdap.param
```

A detailed description of this example is given in Chapter 12.

Chapter 11

Navier-Stokes equations for a one-phase flow

11.1 Introduction

We consider a driven cavity problem of the form

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} - \mu\Delta\mathbf{u} + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p &= 0 \\ \operatorname{div}\mathbf{u} &= 0 \end{aligned}$$

in the unit cube $\Omega = [0,1]^3$ for $t \in [0,1]$ with $\mu = 0.001$. This rather small viscosity coefficient is used to obtain a more interesting (compared to $\mu = 1$) flow field, cf. Figure 11.2. The Dirichlet boundary conditions are taken as $\mathbf{u} = (1,0,0)^T$ on the face $z = 0$ and $\mathbf{u} = (0,0,0)^T$ on the other faces. The initial condition is $\mathbf{u} = (0,0,0)^T$ for $t = 0$.

The following components are used:

- For the triangulation we partition Ω uniformly into $16 \times 16 \times 16$ cubes and then each of them is subdivided into six tetrahedra. Then the subdomains [0, 0.2] × [0, 0.2] × [0, 0.2] and [0.8, 1] × [0, 0.2] × [0, 0.2] are refined once more.

- P_2-P_1 finite element pair for velocity and pressure. The discretization is implemented in the routines `SetupInstatSystem`, `SetupInstatRhs` of the class `StokesP2P1CL` (file `stokes/stokes.h`) and `SetupNonlinear` of the class `NavierStokesP2P1CL` (file `navstokes/navstokes.h`).

- For time discretization and linearization, the θ-scheme with $\theta = 1$ and the fixed point defect correction are used, cf. sections 3.3.1 and 3.4. These are implemented in the classes `InstatNavStokesThetaSchemeCL` (file `navstokes/integrTime.h`) and `AdaptFixedPtDefectCorrCL` (file `num/nssolver.h`), respectively.

- The Oseen problems are solved by a preconditioned block GCR solver, cf. section 3.5.3. In DROPS it is realized by a combination of the classes `GCRSolverCL` (file `num/solver.h`) and `BlockMatrixSolverCL` (file `num/stokessolver.h`). The accuracy for the Oseen solver is increased by a factor 10 after each iteration of the fixed point method. The preconditioner for GCR is as in (3.35). This block preconditioner is implemented in the class `BlockPreCL` in `num/stokessolver.h`. To apply $\mathbf{Q}_A^{-1}$ to a vector $\mathbf{b}$, we use a Jacobi-preconditioned BICGSTAB solver. For $\mathbf{Q}_S^{-1}$ we use the preconditioner as in (3.36).

11.2 Implementation

In this section we describe how the problem and the methods outlined in the previous section are implemented in DROPS (`navstokes/drivcav.cpp`).

11.2.1 Input parameters

The parameters needed to solve the Navier-Stokes problem are the following:

- parameters for the fixed point iteration
  - `fp_tol`: tolerance.
  - `fp_maxiter`: maximal number of iteration.
  - `deco_red`: the reduction factor, which is used to set the new tolerance of the linear solver for the next fixed point iteration.
- parameters for the time integration
  - `theta`: parameter of the θ-scheme.
  - `num_timestep`: number of time step.
  - `time_begin` and `time_end`: begin and end point of the time interval.
- parameters for the mesh refinement:
  - `shell_width`: width of the domain to be further refined.
  - `c_level`: coarsest level.
  - `f_level`: finest level.

They will be read from the command line in the `main` function.

11.2.2 Structure of the program

We need the following libraries:

```
#include "geom/multigrid.h"
#include "out/output.h"
#include "out/ensightOut.h"
#include "geom/builder.h"
#include "stokes/stokes.h"
#include "num/nssolver.h"
#include "navstokes/navstokes.h"
#include "navstokes/integrTime.h"
#include <fstream>
#include <sstream>
#include <string>
```

The program consists of

- A class `StokesCoeffCL`, which contains the coefficients of the PDE and the right hand side function **g**, to be used for the template parameter of the problem class `NavierStokesP2P1CL`.
- The functions `uD` for the Dirichlet boundary conditions and `Null` for the initial condition of the velocity.

- The functions `MakeInitialTriangulation` and `ModifyGridStep` for generating the initial triangulation.
- The `main` function, which describes the Navier-Stokes problem and calls the function `Strategy` to solve it.
- The function `Strategy`: the most important part, in which the Navier-Stokes problem defined in the `main` function is solved.

In the next sections, we present the essential parts of functions `main` and `Strategy` in detail.

11.2.3 The function `main`

The `main` function first reads the input parameters into the corresponding variables.

The domain is partitioned uniformly into $n \times n \times n$ cubes, then each of them is dived into 6 tetrahedra, using an object of the `BrickBuilderCL` class.

```
DROPS::BrickBuilderCL brick(DROPS::std_basis<3>(0), DROPS::std_basis<3>(1),
                            DROPS::std_basis<3>(2), DROPS::std_basis<3>(3),n,n,n);
```

For each face of the cube, the type of boundary condition is set with `DirBC`, which indicates the inhomogeneous Dirichlet boundary condition and the boundary value for the velocity is contained in the array `bnd_fun`

```
const DROPS::BndCondT bc[6]= { DROPS::DirBC, DROPS::DirBC, DROPS::DirBC,
                               DROPS::DirBC, DROPS::DirBC, DROPS::DirBC};
const DROPS::StokesVelBndDataCL::bnd_val_fun bnd_fun[6]=
      {&uD, &uD, &uD, &uD, &uD, &uD);
```

We define an object `prob` of the class `NavierStokesP2P1CL` for the Navier-Stokes problem:

```
typedef DROPS::NavierStokesP2P1CL<StokesCoeffCL> NavierStokesCL;
NavierStokesCL prob(brick, StokesCoeffCL(),
                    DROPS::StokesBndDataCL(6, bc, bnd_fun));
```

The problem is solved by calling the function `Strategy`.

```
Strategy(prob, fp_tol, fp_maxiter, deco_red, oseen_maxiter,
    theta, num_timestep,  time_begin, time_end, shell_width, c_level, f_level);
```

One can now access the computed solutions via the member functions `GetVelSolution` and `GetPrSolution` of `prob`.

11.2.4 The function `Strategy`

The `Strategy` function has the following prototype

```
template<class Coeff>
void Strategy(DROPS::NavierStokesP2P1CL<Coeff>& NS, double fp_tol, int fp_maxiter,
              double deco_red, int oseen_maxiter, double theta,
              DROPS::Uint num_timestep, double time_begin, double time_end,
              double shell_width, int c_level, int f_level)
```

First, we refine the grid in the above mentioned subdomains further and create the triangulation `MG` with

```
MultiGridCL& MG= NS.GetMG();
MakeInitialTriangulation( MG, shell_width, c_level, f_level);
```

For the spatial discretization, we define the variables `A`, `B`, `M`, `N`, `b`, `c`, `cplM`, `cplN` as pointers to the corresponding members of the problem `NS`. After setting the velocity and pressure indices `vidx` and `pidx`, we create the numbering for the velocity and pressure unknowns with

```
NS.CreateNumberingVel( MG.GetLastLevel(), vidx);
NS.CreateNumberingPr( MG.GetLastLevel(), pidx);
```

The members `NumUnknowns` of `vidx` and `pidx` now contain the number of velocity and pressure unknowns. We set the dimensions of the matrices and vectors by assigning their row and column indices to `vidx` and/or `pidx` with the routine `SetIdx`.

The initial condition for $\mathbf{u}$ when $t = 0$ is set to `v` by

```
NS.InitVel(v, &Null);
```

Then we assemble the matrices and the right hand side vector with

```
NS.SetupInstatSystem( A, B, M);
NS.SetupNonlinear( N, v, cplN, 0., 0.);
NS.SetupInstatRhs( b, c, cplM, 0., b, 0.);
```

As we use the same vector `b` for the right hand side and the couplings with the matrix `A`, the sum of them is contained in `b`.
We also assemble the mass matrix:

```
NS.SetupPrMass( &M_pr);
```

The preconditoner $\mathbf{Q}_A$, named `Apc`, is definen by

```
typedef JACPcCL  APcPcT;
APcPcT Apcpc;
typedef BiCGStabSolverCL<APcPcT> ASolverT; // BiCGStab-based APcT
ASolverT Asolver( Apcpc, 300,1e-2, /*relative=*/ true);
typedef SolverAsPreCL<ASolverT> APcT;
APcT Apc( Asolver);
```

We create the preconditoner $\mathbf{Q}_S$ with the name `BBTpc`:

```
typedef  MinCommPreCL BBTPcT;
BBTPcT BBTpc(0, B->Data, M-> Data, M_pr.Data);
```

The constructor of the class `MinCommPreCL` requires the pointer `A` to the upper left matrix $\hat{\mathbf{A}}$ of the block matrix as its first argument. This matrix can only be obtained after we have an object of class `InstatNavStokesThetaSchemeCL`, which should be defined later because its template parameters in turn depend on `MinCommPreCL`. Therefore we initialize `A` with the null pointer and will set the actual pointer to it when the matrix $\hat{\mathbf{A}}$ is available with the function `SetupMatrixA`.

We construct the block preconditioner `oseenpc` for the Oseen problem

```
typedef BlockPreCL<APcT, BBTPcT, false> OseenPcT;
OseenPcT oseenpc( Apc, BBTpc);
```

The third template parameter of the class `BlockPreCL` indicates the type of the preconditioner. The value `true` denotes the diagonal block preconditioner and `false` stands for the upper triangular one.

With this preconditoner, we define the block preconditoned GCR solver `oseensolver`

```
typedef GCRSolverCL<OseenPcT> OseenBaseSolverT;
OseenBaseSolverT oseensolver0( oseenpc, 100, oseen_maxiter, 1e-4, false, &os);
typedef BlockMatrixSolverCL<OseenBaseSolverT> OseenSolverT;
OseenSolverT oseensolver( oseensolver0);
```

For the nonlinear solver `statsolver` we use the fixed point defect correction with step size control

```
typedef AdaptFixedPtDefectCorrCL<NavStokesCL, OseenSolverT> NSSolverT;
NSSolverT statsolver(NS, oseensolver, fp_maxiter, fp_tol, deco_red);
```

The θ-scheme `instatsolver` is applied for the time integration. After that we obtain the reference to the upper left matrix $\hat{\mathbf{A}}$ to set the pointer `A` of `BBTpc`

```
InstatNavStokesThetaSchemeCL<NavStokesCL, NSSolverT > instatsolver(NS, statsolver, theta);
BBTpc.SetMatrixA(&instatsolver.GetUpperLeftBlock());
```

Now we solve the instationary Navier-Stokes equations from `time_begin` to `time_end` with the time step `timestep`. In each time step, the new time is set via `SetTime` and the routine `DoStep` is called.

```
for (int timestep=1; timestep<=num_timestep; timestep++)
{
    double t= time_begin + timestep*dt;
    NS.SetTime( t);
    instatsolver.DoStep( *v, p->Data);
}
```

After compiling, we run the executable with the command of the form

```
./drivcav <fp_tol> <fp_maxiter> <deco_red> <oseen_maxiter> <theta> <num_timestep>
          <time_begin> <time_end> <shell_width> <c_level> <f_level>
```

In our example we take the following parameters:

```
./drivcav  1e-7 200 0.1 100 1 200 0 100 0.2 0 1
```

11.3 Results

The "steady state" of the flow, which is determined by taking a tolerance 10^{-7} in the fixed point iteration, is illustrated in Figures 11.1 and 11.2. In these figures the counter-rotating eddies in the lower right and left corner can be seen.

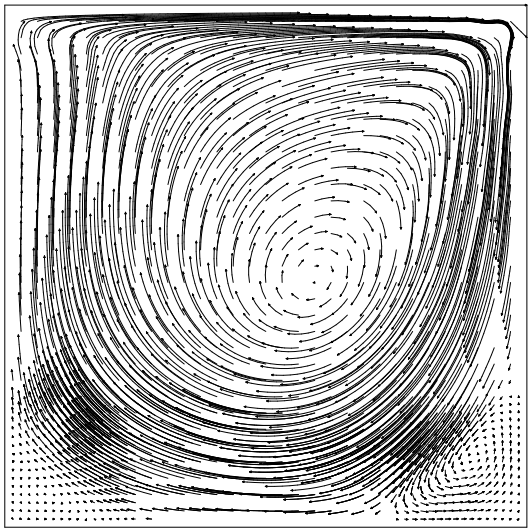

Figure 11.1: Driven Cavity: Velocity field at steady state.

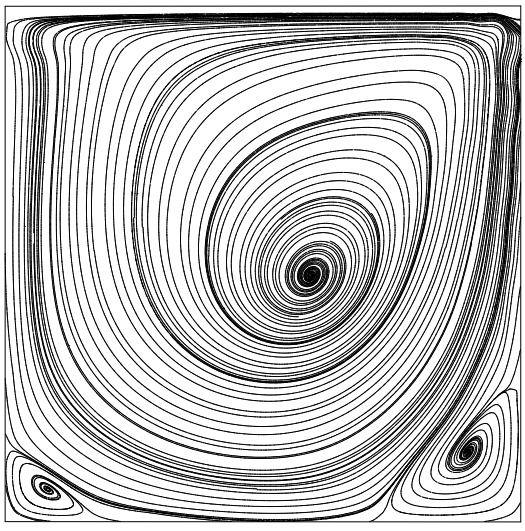

Figure 11.2: Driven Cavity: Streamlines at steady state.

Chapter 12

Navier-Stokes equations for a two-phase flow

12.1 Introduction

We consider a model of the form (1.4) that describes a "rising bubble" problem, namely an n-butanol droplet in water that rises due to gravity. The domain is given as $\Omega := [0, 0.008] \times [0, 0.04] \times [0, 0.008]$. At the initial time, the droplet (Ω_1) is a sphere of radius $R = 0.001$, centered at $(0.004, 0.002, 0.004)$. The material coefficients are given as

$$\begin{aligned} \mu_1 &= 0.003281 \\ \mu_2 &= 0.001388 \\ \rho_1 &= 845.442 \\ \rho_2 &= 986.506 \\ \tau &= 0.00163 \end{aligned} \tag{12.1}$$

Homogeneous Dirichlet boundary conditions for velocity are used. The initial velocity is $(0, 0, 0)^T$ and the external gravity force is $(0, -9.81, 0)^T$.
The following methods are used:

- P_2-P_1X finite element spaces for velocity and pressure. The discretization of the two-phase flow Navier-Stokes equations is done in the class `InstatNavierStokes2PhaseP2P1CL` with the routines `SetupSystem1`, `SetupSystem2` and `SetupNonlinear` (in the files `stokes/instatstokes2phase.h` and `stokes/instatnavstokes2phase.h`).

- The spatial discretization of the levelset equation is implemented in the classes `LevelsetP2CL` and `InterfacePatchCL` (file `levelset/levelset.h`). The discretization is obtained using a quadratic finite element space with streamline diffusion stabilization. The matrices and vectors are assembled using the routines `SetupSystem`.

- The Laplace-Beltrami method for the discretization of the curvature localized force term, cf. section 4.2.3, implemented in the function `SF_ImprovedLaplBeltrami`, which is called by the member function `AccumulateBndIntegral`.

- The discrete approximation of the interface, cf. section 4.2.2, is implemented in the class `InterfacePatchCL`

- A mass conservation correction, implemented in the routine `AdjustVolume`
- The Fast Marching Method for the reparametrization is implemented in the class `FastMarchCL` (file `levelset/fastmarch.h`). Its member function `Reparam` is called by the function `ReparamFastMarching` of `LevelsetP2CL`.
- The time integration is of implicit Euler type, cf. sections 4.3.3 and is implemented in the class `LinThetaScheme2PhaseCL` (file `levelset/coupling.h`).
- A preconditoned GCR method is used to solve the Oseen problem, cf. section 11.2.

12.2 Implementation

The rising bubble problem is implemented in `levelset/risingBubbleAdapt.cpp`.

12.2.1 Input parameters

The input parameters are given in a parameter file (namely `levelset/rising_bubble_butanol_water.param`), a text file (with the extension ".param") which contains assignments of the form

parameter_name = value.

These parameters are obtained easily using an object of the class `ParamMesszelleCL` (file `levelset/params.h`) and can be accessed through its members. (Note that the names of the parameters in the parameter file and the corresponding class members are not identical).

12.2.2 Structure of the program

The following libraries are needed:

```
#include "geom/multigrid.h"
#include "geom/builder.h"
#include "navstokes/instatnavstokes2phase.h"
#include "stokes/integrTime.h"
#include "num/stokessolver.h"
#include "out/output.h"
#include "out/ensightOut.h"
#include "levelset/coupling.h"
#include "levelset/params.h"
#include "levelset/adaptriang.h"
#include "num/bndData.h"
#include <fstream>
```

Similar to the example in chapter 11, this program consists of

- the coefficient class `ZeroFlowCL` as the template parameter for the problem class, `InstatNavierStokes2PhaseP2P1CL`,
- the function `Null` for the initial and Dirichlet boundary conditions of the velocity,
- the function `DistanceFct` to initialize the levelset function,
- the function `sigmaf` which defines the surface tension. If the surface tension is variable one needs an additional function for its gradient.
- the functions `Strategy` and `main`.

12.2.3 The function `main`

In the function `main` we read the input parameters from the the parameter file into a global object `C` of the class `ParamMesszelleCL`

```
std::ifstream param( argv[1]);
param >> C;
param.close();
```

Now we define the triangulation `mg` as an object of the `BrickBuilderCL`

```
std::string mesh( C.meshfile), delim("x@");
size_t idx;
while ((idx= mesh.find_first_of( delim)) != std::string::npos )
    mesh[idx]= ' ';
std::istringstream brick_info( mesh);
brick_info >> dx >> dy >> dz >> nx >> ny >> nz;
if (!brick_info || dx!=dz)
{
    std::cerr << "error while reading geometry information: " << mesh << "\n";
    return 1;
}
DROPS::Point3DCL orig, px, py, pz;
px[0]= dx; py[1]= dy; pz[2]= dz;
DROPS::BrickBuilderCL builder( orig, px, py, pz, nx, ny, nz);
DROPS::MultiGridCL mg( builder);
```

which will be adaptively refined in a zone of width `C.ref_width` up to the level `C.ref_flevel` around the interface through an object `adap` of the class `AdapTriangCL`. The routine `MakeInitialTriang` creates the initial triangulation at the time $t = 0$ and uses the distance function `DistanceFct`.

```
DROPS::AdapTriangCL adap( mg, C.ref_width, 0, C.ref_flevel);
adap.MakeInitialTriang(DistanceFct);
```

We assign a boundary condition to each boundary segment

```
DROPS::BndCondT bc[6];
DROPS::StokesVelBndDataCL::bnd_val_fun bnd_fun[6];
for (DROPS::BndIdxT i=0; i<num_bnd; ++i)
{
    bc[i]= DROPS::WallBC;
    bnd_fun[i]= &Null;
}
```

and define an object `prob` of the problem class `InstatNavierStokes2PhaseP2P1CL`

```
typedef ZeroFlowCL CoeffT;
typedef DROPS::InstatNavierStokes2PhaseP2P1CL<CoeffT> MyStokesCL;
MyStokesCL prob(mg, ZeroFlowCL(C), DROPS::StokesBndDataCL(
                num_bnd, bc, bnd_fun), DROPS::P1X_FE, C.XFEMStab);
```

The function `Strategy` is now called to solve the problem

```
Strategy( prob, adap);
```

12.2.4 The function `Strategy`

The `Strategy` function has the prototype

```
template<class Coeff>
void Strategy( InstatNavierStokes2PhaseP2P1CL<Coeff>& Stokes, AdapTriangCL& adap)
```

The triangulation from `Stokes` is obtained using the routine `GetMG`

```
MultiGridCL& MG= Stokes.GetMG();
```

An object `lset` of `LevelsetP2CL` is created to describe the levelset equation:

```
sigma= Stokes.GetCoeff().SurfTens;
LevelsetP2CL lset( MG, &sigmaf, /*grad sigma*/ 0, C.lset_theta, C.lset_SD, -1,
                   C.lset_iter, C.lset_tol, C.CurvDiff);
```

The third parameter in the above constructor of `lset` is set as the null pointer, because the surface tension is constant. Otherwise, one should use a pointer to the function which describes the gradient of the surface tension.

Then we create the numberings for the levelset, velocity and pressure using the member functions `CreateNumbering...` of `lset` and `Stokes` with the corresponding index description pointers `lidx`, `vidx` and `pidx`

```
IdxDescCL* lidx= &lset.idx;
IdxDescCL* vidx= &Stokes.vel_idx;
IdxDescCL* pidx= &Stokes.pr_idx;
lset.CreateNumbering( MG.GetLastLevel(), lidx);
Stokes.CreateNumberingVel( MG.GetLastLevel(), vidx);
Stokes.CreateNumberingPr(  MG.GetLastLevel(), pidx, 0, &lset);
```

and set the indices to their member matrix- and vector description objects via the member function `SetIdx`

```
lset.Phi.SetIdx( lidx);
Stokes.b.SetIdx( vidx);
Stokes.v.SetIdx( vidx);
cplN.SetIdx( vidx);
Stokes.c.SetIdx( pidx);
Stokes.p.SetIdx( pidx);
Stokes.A.SetIdx(vidx, vidx);
Stokes.B.SetIdx(pidx, vidx);
Stokes.prM.SetIdx( pidx, pidx);
Stokes.M.SetIdx(vidx, vidx);
Stokes.N.SetIdx(vidx, vidx);
```

The levelset function and velocity are initialized with the functions `DistanceFct` and `Null` respectively

```
lset.Init( DistanceFct);
Stokes.InitVel( &Stokes.v, Null);
```

To export the solution at each time step to the ensight ouput files for visualization, we use an object of the class `EnsightP2SolOutCL`

```
EnsightP2SolOutCL ensight( MG, lidx);
const string filename= C.EnsDir + "/" + C.EnsCase;
const string datgeo= filename+".geo",
             datpr = filename+".pr" ,
             datvec= filename+".vel",
             datscl= filename+".scl";
ensight.CaseBegin( string(C.EnsCase+".case").c_str(), C.num_steps+1);
ensight.DescribeGeom( "Messzelle", datgeo, true);
ensight.DescribeScalar( "Levelset", datscl, true);
ensight.DescribeScalar( "Pressure", datpr,  true);
ensight.DescribeVector( "Velocity", datvec, true);
```

The iterative nonlinear and linear and solvers (including the corresponding preconditioners) for the Navier-Stokes problem are similar to those in the one-phase flow example in the previous chapter:

```
typedef JACPcCL  APcPcT;
APcPcT Apcpc;

typedef GMResSolverCL<APcPcT> ASolverT;
ASolverT Asolver( Apcpc, 500, /*restart*/ 100, 1e-2, /*relative=*/ true);
typedef SolverAsPreCL<ASolverT> APcT;
APcT Apc( Asolver);

typedef  MinCommPreCL BBTPcT;
BBTPcT BBTpc(0, Stokes.B.Data, Stokes.M.Data, Stokes.prM.Data);

typedef BlockPreCL<APcT, BBTPcT, false> OseenPcT;
OseenPcT oseenpc( Apc, BBTpc);

typedef GCRSolverCL<OseenPcT> OseenBaseSolverT;
OseenBaseSolverT oseensolver0( oseenpc, /*truncate*/ C.outer_iter, C.outer_iter,
                               C.outer_tol, /*relative*/ false);
typedef BlockMatrixSolverCL<OseenBaseSolverT> OseenSolverT;
OseenSolverT oseensolver( oseensolver0);

typedef AdaptFixedPtDefectCorrCL<StokesProblemT, OseenSolverT> NSSolverT;
NSSolverT nssolver( Stokes, oseensolver, C.ns_iter, C.ns_tol, C.ns_red);
```

An object of the class `LinThetaScheme2PhaseCL` is created for time integration:

```
typedef LinThetaScheme2PhaseCL<StokesProblemT, NSSolverT> CouplingT;
CouplingT cpl( Stokes, lset, nssolver, C.nonlinear, true, C.cpl_stab);
```

The routine `SetMatrixA` is used to set the actual matrix $\hat{\mathbf{A}}$ to the `BBTPcT` preconditioner.

```
BBTpc.SetMatrixA( &cpl.GetUpperLeftBlock());
```

We use the routine `SetTimeStep` to set the time and the time integration procedure is as follows:

```
cpl.SetTimeStep( C.dt);
for (int step= 1; step<=C.num_steps; ++step)
{
    cpl.DoStep( C.cpl_iter);
    time+=C.dt;

    if (C.VolCorr)
    {
        double dphi= lset.AdjustVolume( Vol, 1e-9);
        lset.Phi.Data+= dphi;
    }

    if (C.RepFreq && step%C.RepFreq==0)
    {
        lset.ReparamFastMarching( C.RepMethod);
        adap.UpdateTriang( Stokes, lset);
        if (adap.WasModified() )
        {
            cpl.Update();
        }

        if (C.VolCorr)
        {
            double dphi= lset.AdjustVolume( Vol, 1e-9);
            lset.Phi.Data+= dphi;
        }
    }
    ensight.putGeom( datgeo, step*C.dt);
    ensight.putScalar( datpr, Stokes.GetPrSolution(), step*C.dt);
    ensight.putVector( datvec, Stokes.GetVelSolution(), step*C.dt);
    ensight.putScalar( datscl, lset.GetSolution(), step*C.dt);
    ensight.Commit();
}
```

In each time step, the member function `DoStep` of `cpl` is used to solve the levelset and the Navier-Stokes equations. After that, we compute the amount of volume difference in Ω_1 with the routine `AdjustVolume` and add the correction term to the levelset function. The reparametrization is performed after each `C.RepFreq` steps by the member function `ReparamFastMarching` of `levelset`. The triangulation is then modified and the solutions are interpolated based on the new re-initialized levelset function with the function `UpdateTriang` of `adap`. If the numberings and indices of the vectors are changed, the mass matrix for the pressure should also be updated. At the end, we export the geometry and the solutions to the ensight files with the member functions `putGeom`, `putScalar` and `putVector` of `ensight`.

12.3 Results

The domain is partitioned into $4 \times 20 \times 4$ cubes and then each of them is subdivided into six tetrahedra. A strip with width $4.5 \cdot 10^{-4}$ that contains the interface is refined three times more, resulting in a mesh size of 0.25 mm near the interface. The initial grid is showed in Figure 12.3.

We consider the time interval [0, 0.75] with the time step Δt is $5 \cdot 10^{-4}$. The nonlinear system for the velocity and pressure is solved with a tolerance $5 \cdot 10^{-10}$ and the tolerance parameter for the levelset equation is 10^{-10}.

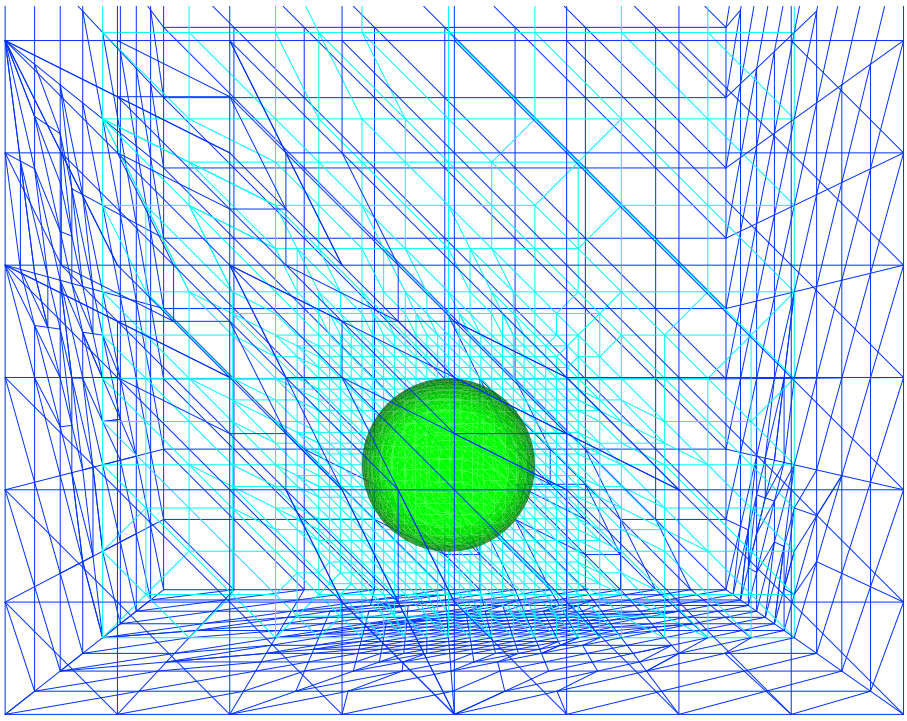

Figure 12.1: Rising droplet: Initial state.

In Figure 12.2 position and velocities for different times are shown.

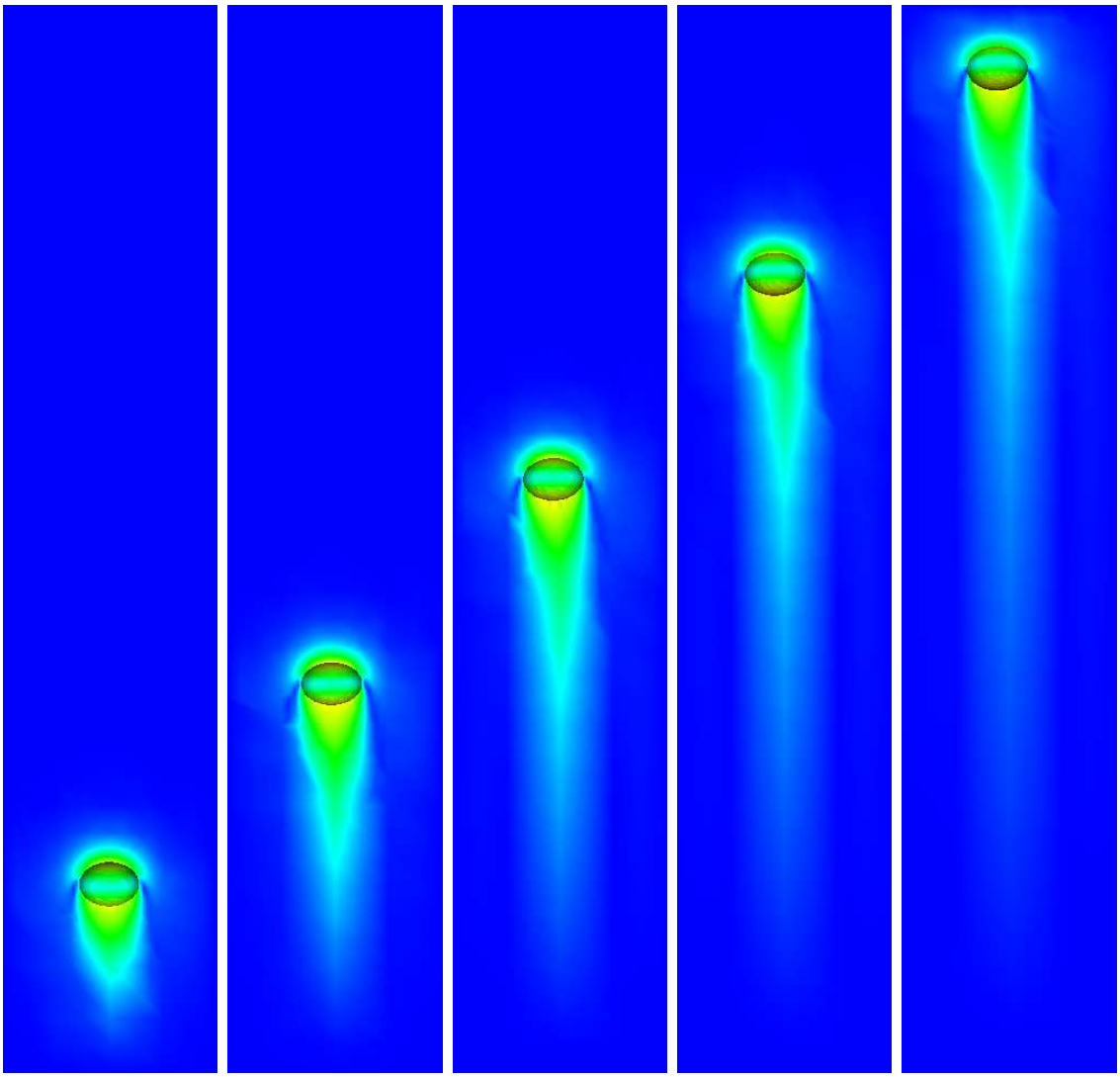

Figure 12.2: Dynamics of the rising droplet at $t = 0.15,\ 0.3,\ 0.45,\ 0.6,\ 0.75$, color coding indicates velocity magnitude

A zoom-in that illustrates the velocity field close to the droplet is given in Figure 12.3.

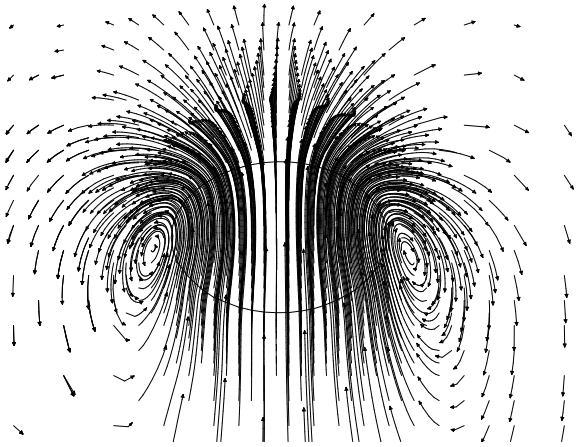

Figure 12.3: Velocity vectors at time $t = 0.6$

Figure 12.4 shows the size of the y-component of velocity at the droplet's barycenter, for t from 0.2 to 0.6. These results were used for validation. It turns out that this rising velocity is in good agreement with results from measurements that are available in the literature.

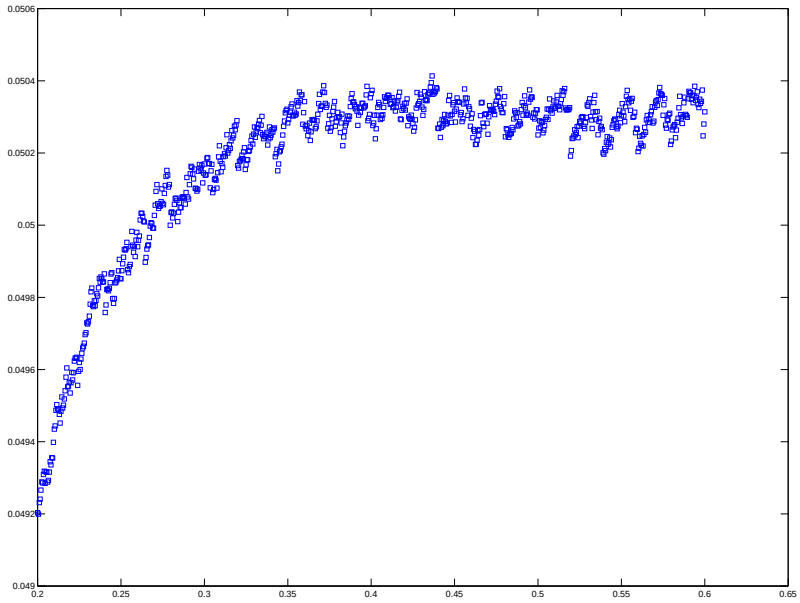

Figure 12.4: Rising velocity of the droplet.

Chapter 13

A two-phase flow problem with mass transport

13.1 Introduction

We consider the two-phase flow problem described in chapter 12 but now combined with an additional the mass transport equation. Thus the model is as in (1.5). The surface tension coefficient τ is assumed to be constant (independent of the concentration of the surfactant). The diffusion coefficients and Henry's constant are

$$\begin{aligned} D_1 &= 2.3 \cdot 10^{-6} \\ D_2 &= 5.8 \cdot 10^{-6} \\ C_H &= 0.75 \end{aligned} \tag{13.1}$$

which are much larger than physically realistic ones. We use these large values to have a time scale for the transport process that is comparable the one of the flow dynamics. Hence one can better observe the change in the concentration profiles while the droplet rises, cf. figure 13.1. A homogeneous Neumann boundary condition is imposed for the mass transport problem. The initial values of the concentrations inside and outside the droplet are 10 and 0.1, respectively. Thus at $t = 0$ the Henry interface condition is not satisfied.

Since the surface tension coefficient is assumed to be independent of the concentration of the third component there is a coupling only in *one* direction between the two-phase flow problem and the transport equation for the third component. Thus per time step we first solve the two-phase Navier-Stokes plus level set equations and then we use the given velocity and interface to solve the transport equation. For the two-phase flow problem we use the same methods as in chapter 12. For the transport equation we use the following methods, which are implemented in `TransportP1CL` class (file `poisson/transport2phase.h`):

- $P1$ finite element space for the concentration c and the transformed concentration $\tilde{c}$, cf. chapter 5. The matrices and vectors are assembled in the member functions `SetupInstat-System`.
- the θ-scheme for the time integration, cf. section 5.3, in the function `DoStep`.
- the transformation from c to $\tilde{c}$ and vice versa, with the functions `c2ct` and `ct2c`.

13.2 Implementation

The program is almost identical to that in the previous chapter, with the following additional components for the mass transport part:

- The header file for `TransportP1CL` class

```
#include "poisson/transport2phase.h"
```

- The functions `Initialcneg`, `Initialcpos` to describe the initial values of the concentration in Ω_1 and Ω_2, respectively. These values are read from the parameter file in the `main` function.

- The boundary conditions for the concentration

```
typedef DROPS::BndDataCL<> cBndDataCL;
typedef cBndDataCL::bnd_val_fun  c_bnd_val_fun;

const DROPS::BndCondT c_bc[6]= { DROPS::OutflowBC, DROPS::OutflowBC, DROPS::OutflowBC,
                                 DROPS::OutflowBC, DROPS::OutflowBC, DROPS::OutflowBC};
const c_bnd_val_fun c_bfun[6]= {0, 0, 0, 0, 0, 0};
```

- In the function `main`, we assign the parameters of the transport problem, which are read from the parameter file, to the global variables needed to construct the `TransportP1CL` class

```
ini_c[0]= C.cF;
ini_c[1]= C.cD;
D[0]= C.diffusityF;
D[1]= C.diffusityD;
H= C.equilibrium_jump;
```

- In the function `Strategy`, we define an object `c` of the `TransportP1CL` class with

```
cBndDataCL Bnd_c( 6, c_bc, c_bfun);
TransportP1CL c( MG, Bnd_c, Bnd_v, /*theta*/ 1., D, H, &Stokes.v, lset,
    /*t*/ 0., C.dt, C.transp_iter, C.transp_tol);
IdxDescCL* cidx= &c.idx;
c.CreateNumbering( MG.GetLastLevel(), cidx);
c.ct.SetIdx( cidx);
c.Init( &Initialcneg, &Initialcpos);
c.Update();
c.SetTimeStep( C.dt);
```

  The solutions of the transport problem can be exported to the ensight output files

```
const string datc = filename+".c" ,
             datct = filename+".ct";
ensight.DescribeScalar( "Concentration", datc, true);
ensight.DescribeScalar( "TransConc", datct, true);
ensight.putScalar( datc,  c.GetSolution(), 0);
ensight.putScalar( datct, c.GetSolution( c.ct), 0);
```

  In each time step, after solving the two-phase flow part, we solve the transport problem with the function `DoStep` of `c`. If the grid is modified, the concentration should be updated with the member functions `UpdateTriang` of `adap` (using the address of `c` as the third parameter) and `Update` of `c`.

```
for (int step= 1; step<=C.num_steps; ++step)
{
    std::cerr << "Schritt " << step << "\t t = "<< step*C.dt<<":\n";
    cpl.DoStep( C.cpl_iter);
```

```
    c.DoStep( step*C.dt);
    lset.GetInfo( maxGradPhi, Volume, bary_drop, min_drop, max_drop);
    infofile << Stokes.t << '\t' << maxGradPhi << '\t' << Volume << '\t' << bary_drop
     << '\t' << min_drop << '\t' << max_drop << std::endl;
    std::cerr << "rel. Volume: " << lset.GetVolume()/Vol << std::endl;
    if (C.VolCorr)
    {
        double dphi= lset.AdjustVolume( Vol, 1e-9);
        std::cerr << "volume correction is " << dphi << std::endl;
        lset.Phi.Data+= dphi;
        std::cerr << "new rel. Volume: " << lset.GetVolume()/Vol << std::endl;
    }

    if (C.RepFreq && step%C.RepFreq==0) // reparam levelset function
    {
        lset.ReparamFastMarching( C.RepMethod);

        adap.UpdateTriang( Stokes, lset, &c);
        if (adap.WasModified() )
        {
            cpl.Update();
            c.Update();
        }

        std::cerr << "rel. Volume: " << lset.GetVolume()/Vol << std::endl;

        if (C.VolCorr)
        {
            double dphi= lset.AdjustVolume( Vol, 1e-9);
            std::cerr << "volume correction is " << dphi << std::endl;
            lset.Phi.Data+= dphi;
            std::cerr << "new rel. Volume: " << lset.GetVolume()/Vol << std::endl;
        }
    }
    ensight.putGeom( datgeo, step*C.dt);
    ensight.putScalar( datpr, Stokes.GetPrSolution(), step*C.dt);
    ensight.putVector( datvec, Stokes.GetVelSolution(), step*C.dt);
    ensight.putScalar( datscl, lset.GetSolution(), step*C.dt);
    ensight.putScalar( datc, c.GetSolution(), step*C.dt);
    ensight.putScalar( datct, c.GetSolution( c.ct), step*C.dt);
    ensight.Commit();
}
```

13.3 Results

In each time step the linear system for the new concentration is solved with a tolerance 10^{-10}.

The concentration at several points in time are shown in Figure 13.1.

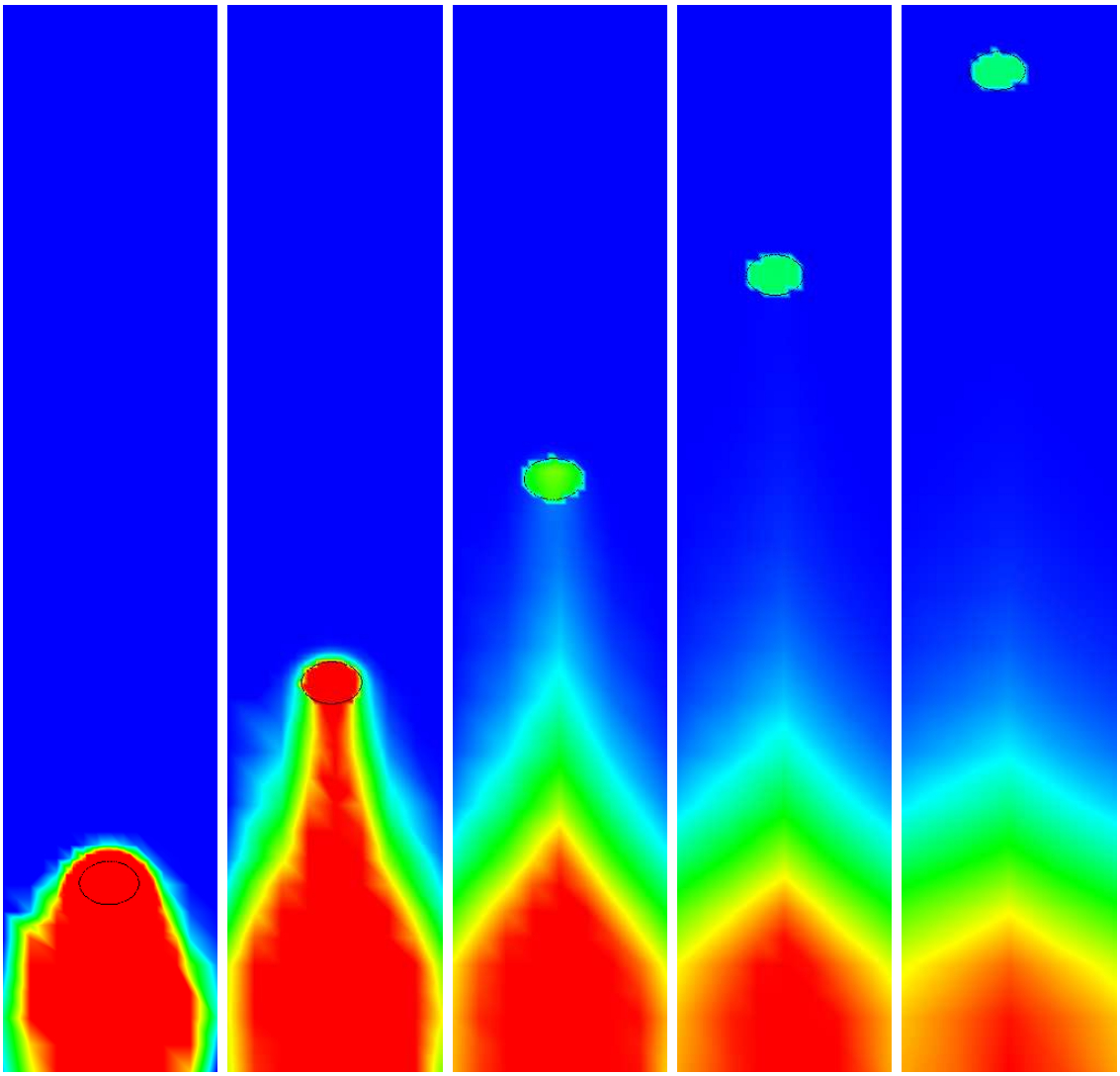

Figure 13.1: Dynamics of the mass transport. Droplet position and concentration profiles at $t = 0.15$, 0.3, 0.45, 0.6, 0.75, color coding indicates concentration value

Chapter 14

Appendix

14.1 Parameter file

```
#============================================================
#    DROPS parameter file for
#    simulation of two-phase flow:
#    rising droplet
#============================================================



# time stepping
Time {
  NumSteps      =       1300
  StepSize      =       5e-4
  ThetaLevelset =       1       # backward Euler
  ThetaStokes   =       1       # backward Euler
}

# flow solver
Stokes {
  InnerIter     =       1000
  OuterIter     =       200
  InnerTol      =       1e-12
  OuterTol      =       1e-10
  StokesMethod  =       -1      # no effect
}

NavStokes {
  Nonlinear     =       1
  Scheme        =       1       # time integration: 0=FS op.splitting, 1=theta-scheme
  Tol           =       5e-10
  Iter          =       20
  Reduction     =       0.01
}

# levelset solver
Levelset {
  Tol           =       1e-10
  Iter          =       10000
  SD            =       0.1
  CurvDiff      =       -1
  VolCorrection =       1
}

Coupling {
  Tol           =       1e-10   # not used a.t.m.
  Iter          =       -1      # -1 = till convergence
  Stab          =       1.      # Laplace-Beltrami-Stabilization
}
```

```
# re-initialization of levelset function
Reparam {
  Freq          =       3     # 0 = no reparametrization
  Method        =       1     # 0/1 = fast marching without/with modification of zero
}

# adaptive refinement
AdaptRef {
  Freq          =       1
  FinestLevel   =       3
  Width         =       0.45e-3
}

# material data, all units are SI
Mat {
  DensDrop      =       845.442    # n-Butanol / Wasser
  ViscDrop      =       3.281e-3
  DensFluid     =       986.506
  ViscFluid     =       1.388e-3
  SmoothZone    =       1e-4
  SurfTension   =       1.63e-3
}

# mass-transport-parameters, all units are SI
MassTransp {
  ConcDrop      =       5
  ConcFluid     =       0.1
  DiffusityDrop =       2.3e-6
  DiffusityFluid=       5.76e-6
  EquilibJump   =       0.75     # cFluid= EquilibJump*cDrop in the equilibrium
  Tol           =       1e-10     # relative
  Iter          =       500
}

# experimental conditions
Exp {
  RadDrop       =       1e-3  1e-3  1e-3
  PosDrop       =       4e-3    2e-3    4e-3
  Gravity       =       0       -9.81   0
  FlowDir       =       1       # flow in y-direction
  InflowVel     =       0
  RadInlet      =       4e-3    # 3.5e-3 for old meas. cell
}

# miscellaneous

InitialCond     =       0       # 0=zero, 1/2=flow with/without drop, 3=read from file
InitialFile     =       initial/brick_adap0
MeshFile        =       8e-3x40e-3x8e-3@4x20x4
EnsightCase     =       butanol_diam2mm
EnsightDir      =       ensight
XFEMStab        =       1.0
```

Bibliography

[1] S. Gross, A. Reusken, *Parallel multilevel tetrahedral grid refinement*, SIAM J. Sci. Comp. 26, No.4, pp. 1261–1288, 2005.

[2] S. Gross, V. Reichelt, A. Reusken, *A Finite Element based Level Set Method for Two-phase Incompressible Flows*, Comp. Vis. Sci. 9, No. 4, pp. 239–257, 2006.

[3] S. Gross, A. Reusken, *Finite element discretization error analysis of a surface tension force in two-phase incompressible flows*, SIAM J. Numer. Anal. 45, No. 4, pp. 1679–1700, 2007.

[4] S. Gross, A. Reusken, *An extended pressure finite element space for two-phase incompressible flows with surface tension*, J. Comp. Phys. 224, pp. 40–58, 2007.

[5] M. Larin, A. Reusken, *A comparative study of efficient iterative solvers for generalized Stokes equations*, Numer. Lin. Alg. Appl. 15, pp. 13–34, 2008.

[6] M. A. Olshanskii, A. Reusken, *Analysis of a Stokes interface problem*, Numer. Math. 103, pp. 129–149, 2006

[7] M. A. Olshanskii, J. Peters, A. Reusken, *Uniform preconditioners for a parameter dependent saddle point problem with application to generalized Stokes interface equations*, Numer. Math. 105, No. 252, pp. 159–191, 2006.

[8] M. A. Olshanskii, A. Reusken, *A Stokes interface problem: stability, finite element analysis and a robust solver*, In: P. Neittaanmäki, et al. (Eds.), Proceedings of European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS 2004.

[9] M. A. Olshanskii, A. Reusken, J. Grande, *An Eulerian Finite Element method for elliptic equations on moving surfaces*, IGPM Preprint (2008). Submitted to SIAM J. Numer. Anal.

[10] J. Peters, V. Reichelt, A. Reusken, *Fast iterative solvers for discrete Stokes equations*, SIAM J. Sci. Comput. 27, No. 2, pp. 646–666, 2005.

[11] A. Reusken, *Multilevel techniques for two-phase incompressible flows*, Proceedings of the 8th European Multigrid Conference (EMG 2005), 2005.

[12] A. Reusken and V. Reichelt, *Mit Gittern auf der Spur von Tropfen*, TerraTech, Zeitschrift für Altlasten und Bodenschutz 6,pp. 35–37, 2001.

[13] A. Reusken, *Analysis of an extended pressure finite element space for two-phase incompressible flows*, Accepted for publication in Comp. Vis. Sci.

[14] C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt, *Experiences with the OpenMP Parallelization of DROPS, a Navier-Stokes Solver written in C++*, Proceedings of the first international workshop on OpenMP (IWOMP 2005), 2005.

[15] C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt, *Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP*, Proceedings of the international conference on parallel computing (ParCo 2005), 2005.

[16] DROPS webpage:, `http://www.igpm.rwth-aachen.de/DROPS/`

[17] N. Anderson, Å. Björck, *A new high Order Method of Regula Falsi Type for Computing the Root of an Equation*, BIT 13, pp. 253–264, 1973.

[18] E. Bänsch, *Numerical methods for the instationary Navier-Stokes equations with a free capillary surface*, Habilitation thesis, Albert-Ludwigs-University Freiburg, 1998.

[19] E. Bänsch, *Finite element discretization of the Navier-Stokes equations with a free capillary surface*, Numer. Math. 88, pp. 203–235, 2001.

[20] R. E. Bank, A. H. Sherman, A. Weiser, *Refinement algorithms and data structures for regular local mesh refinement*, in: Scientific computing (R. Stepleman, ed.), North-Holland, Amsterdam, pp. 3–17, 1983.

[21] R. E. Bank, B. D. Welfert, H. Yserentant, *A class of iterative methods for solving saddle point problems*, Numer. Math. 56, pp. 645–666, 1990.

[22] P. Bastian, *Parallele adaptive Mehrgitterverfahren*, Teubner, Stuttgart, 1996.

[23] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, C. Wieners, *UG - A flexible software toolbox for solving partial differential equations*, Computing and Visualization in Science 1, pp. 27–40, 1997.

[24] P. Bastian, *Load balancing for adaptive multigrid methods*, SIAM J. Sci. Comput. 19, pp. 1303–1321, 1998.

[25] P. Bastian, K. Birken, K. Johannsen, S. Lang, V. Reichenberger, G. Wittum, C. Wrobel, *A parallel software-platform for solving problems of partial differential equations using unstructured grids and adaptive multigrid methods*, In: High performance computing in science and engineering (E. Krause and W. Jäger, eds.), pp. 326–339, Springer, Berlin, 1999.

[26] M. Behr, *Free-surface flow simulations in the presence of inclined walls*, Comput. Methods Appl. Mech. Engrg. 191, pp. 5467–5483, 2002.

[27] J. Bey, *Tetrahedral grid refinement*, Computing 55, pp. 355–378, 1995.

[28] J. Bey, *Finite-Volumen- und Mehrgitterverfahren für elliptische Randwertprobleme*, Advances in Numerical Methods, Teubner, Stuttgart, 1998.

[29] J. Bey, *Simplicial grid refinement: on Freudenthal's algorithm and the optimal number of congruence classes*, Numer. Math. 85, pp. 1–29, 2000.

[30] D. Bothe, M. Koebe, H.-J. Warnecke, *VOF-simulations of the rise behavior of single airbubbles with oxygen transfer to the ambient liquid*, In: F.-P. Schindler (ed.): *Transport Phenomena with Moving Boundaries*, pp. 134–146. VDI-Verlag, Düsseldorf, 2004.

[31] D. Bothe, J. Prüss, G. Simonett, *Well-posedness of a two-phase flow with soluble surfactant*, In: M. Chipot, J. Escher (ed.): *Nonlinear Elliptic and Parabolic Problems*, pp. 37–61. Birkhäuser, 2005.

[32] J. U. Brackbill, D. B. Kothe, C. Zemach, *A continuum method for modeling surface tension*, J. Comput. Phys. 100, pp. 335–354, 1992.

[33] D. Braess, *Finite Elements: Theory, Fast Solvers and Applications in Solid Mechanics*, Cambridge University Press, Cambridge, 2007.

[34] J. H. Bramble, J. E. Pasciak, *Iterative techniques for time dependent Stokes problems*, Comput. Math. Appl. 33, No. 1-2, pp. 13–30, 1997.

[35] M. O. Bristeau, R. Glowinski, J. Periaux, *Numerical methods for the Navier-Stokes equations. Application to the simulation of compressible and incompressible flows*, Computer Physics Report 6, pp. 73–188, 1987.

[36] Cahouet, J., Chabard, J.P.: Some fast 3D finite element solvers for the generalized Stokes problem. Int. J. Numer. Methods Fluids, **8**, 1988, 869–895.

[37] B. Chamberlain *Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations*, TR-98-10-03, 1998.

[38] Y. C. Chang, T. Y. Hou, B. Merriman, S. Osher, *A level set formulation of Eulerian interface capturing methods for incompressible fluid flows*, J. Comput. Phys. 124, pp. 449–464, 1996.

[39] K. Deckelnick, G. Dziuk, *Mean curvature flow and related topics*, In: Frontiers in Numerical Analysis, Durham 2002 (J. F. Blowey, A. W. Craig, T. Shardlow, eds.), pp. 63–108, Springer, 2003.

[40] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, U.V. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 10 pp.-, 2006

[41] G. Dziuk, *An algorithm for evolutionary surfaces*, Numer. Math. 58, pp. 603–611, 1991.

[42] H. Elman, V. E. Howle, J. Shadid, R. Shuttleworth, R. Tuminaro, *Block preconditioners based on approximate commutators*, SIAM J. Sci. Comp. 27, pp. 1651–1055, 2005.

[43] Ensight, CEI Inc., 3D visualization tool, `http://www.ensight.com/`.

[44] GAMBIT, ANSYS Inc., mesh generation tool, `http://www.fluent.com/`

[45] S. Ganesan, G. Matthies, L. Tobiska, *On spurious velocities in incompressible flow problems with interfaces*, Preprint, Department of Mathematics, University of Magdeburg, 2005.

[46] S. Ganesan, L. Tobiska, *Finite element simulation of a droplet impinging a horizontal surface*, Proceedings of ALGORITMY 2005, pp. 1–11, 2005.

[47] Geomview, 3D viewing tool, `http://www.geomview.org/`

[48] I. Ginzburg, G. Wittum, *Two-phase flows on interface refined grids modeled with VOF, staggered finite volumes, and spline interpolants*, J. Comput. Phys. 166, pp. 302–335, 2001.

[49] J. Glimm, J.W. Grove, X.L. Li, K.-M. Shyue, Y. Zeng, Q. Zhang, *Three-dimensional front tracking*, SIAM J. Sci. Comp. 19, pp. 703–727, 1998.

[50] J. Glimm, M.J. Graham, J. Grove, X.L. Li, T.M. Smith, D. Tan, F. Tangerman, Q. Zhang, *Front tracking in two and three dimensions*, Comput. Math. Appl 35, pp. 1–11, 1998.

[51] Greenbaum, A., *Iterative Methods for Solving Linear Systems*, Frontiers in Applied Mathematics, Vol. 17, SIAM, Philadelphia, 1997.

[52] S. Gross, *Parallelisierung eines adaptiven Verfahrens zur numerischen Lösung partieller Differentialgleichungen*, Diploma thesis (in german), 2002.

[53] W. Hackbusch, *Multi-grid Methods and Applications*, Springer, Berlin, Heidelberg, 1985.

[54] W. Hackbusch, *Iterative Solution of Large Sparse Systems of Equations*, Springer, Berlin, Heidelberg, 1994.

[55] A. Hansbo, P. Hansbo, *An unfitted finite element method, based on Nitsche's method, for elliptic interface problems*, Comput. Methods Appl. Mech. Engrg. 191, pp. 5537–5552, 2002.

[56] C.W. Hirt, B.D. Nichols, *Volume of Fluid (VOF) Method for the Dynamics of Free Boundaries*, J. Comp. Phys. 39, pp. 201–225, 1981.

[57] S. Hysing, *A new implicit surface tension implementation for interfacial flows*, Int. J. Numer. Meth. Fluids, ..?...., 2005.

[58] A.J. James, J. Lowengrub, *A surfactant-conserving volume of fluid method for interfacial flows with insoluble surfactant.* J. Comp. Phys. 201(2), pp. 685–722, 2004.

[59] George Karypis, Vipin Kumar, *A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering*, Journal of Parallel and Distributed Computing 48(1), pp. 71–95, 1998.

[60] KASKADE, A toolbox for adaptive multilevel codes,
`http://www.zib.de/Scisoft/kaskade2/`

[61] R. Kimmel, J. A. Sethian, *Computing geodesic paths on manifolds*, Proc. Natl. Acad. Sci. 95, pp. 8431–8435, 1998.

[62] P. Kloucek, F. Rys, *Stability of the fractional step θ-scheme for the nonstationary Navier-Stokes equations*, SIAM J. Numer. Anal. 31, pp. 1312–1335, 1994.

[63] S. Lang, *Parallele numerische Simulation instationärer Probleme mit adaptiven Methoden auf unstrukturierten Gittern*, Ph.D. thesis, University of Stuttgart, 2001.

[64] S. Lang, *UG - A parallel software tool for unstructured adaptive multigrids*, In: Parallel Computing: Fundamentals & Applications, Proceedings of the International conference ParCo'99 (E. H. D'Hollander, J. R. Joubert, F. J. Peters, H. Sips, eds.), Imperial College Press, Delft, 1999.

[65] Message Passing Interface Forum, MPI *A Message-Passing Interface Standard*, International Journal of Supercomputer Applications, pp. 165–414, 1994.

[66] OpenMP, *a specification for shared memory parallelization*, `http://www.openmp.org`.

[67] S. Osher, J. A. Sethian, *Fronts propagating with curvature dependent speed: algorithms based on Hamilton-Jacobi formulations*, J. Comp. Phys. 79, pp. 12–49, 1988.

[68] S. Osher, R. P. Fedkiw, *Level set methods: An overview and some recent results*, J. Comput. Phys. 169, pp. 463–502, 2001.

[69] Paige, C.C., Saunders, M.A.: Solution of sparse indefinite systems of linear equations. SIAM J. Numer. Anal., **11**, 1974, 197–209.

[70] ParaView, parallel visualization application, `http://www.paraview.org/`

[71] S. B. Pillapakkam, P. Singh, *A level-set method for computing solutions to viscoelastic two-phase flow*, J. Comput. Phys. 174, pp. 552–578, 2001.

[72] ParMETIS, *parallel graph partitioning package*, `http://glaros.dtc.umn.edu/gkhome/metis/parmeti`

[73] Powell, C., Silvester, D.: Black-box preconditioning for mixed formulation of self-adjoint elliptic PDEs. In: *Challenges in Scientific Computing - CISC 2002*, ed. E. Bänsch. Lecture Notes in Computational Science and Engineering, Vol. 35, 268–285, 2003.

[74] R. Rannacher, *Finite Element Methods for the Incompressible Navier-Stokes Equations*, in: Fundamental directions in mathematical fluid mechanics (G.-P. Galdi et al., eds.), pp. 191–293, Birkhäuser, Basel, 2000.

[75] R. Rannacher, *On the numerical solution of the incompressible Navier-Stokes equations*, ZAMM 73, pp. 203–216, 1993 (V. C. Boffy and H. Neunzert, eds.), pp. 34–53, Teubner, Stuttgart, 1998.

[76] H.-G. Roos, M. Stynes, L. Tobiska, *Numerical methods for singularly perturbed differential equations: convection diffusion and flow problems*, Springer ser. in comp. math. Vol 24, Springer, Berlin, Heidelberg, 1996.

[77] M. Rudman, *Volume-Tracking Methods for Interfacial Flow Calculations*, Int. J. Num. Meth. Fluids 24, pp. 671–691, 1997.

[78] Rusten, T. and Winther, R.: A preconditioned iterative method for saddlepoint problems. SIAM J. Matrix Anal. Appl., **13**, 1992, 887–904.

[79] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996.

[80] William J. Schroeder, Lisa S. Avila, William Hoffman, *Visualizing with VTK: A Tutorial*, IEEE Computer Graphics and Applications, 20(5), 2000,

[81] J. A. Sethian, *A fast marching level set method for monotonically advancing fronts*, Proc. Natl. Acad. Sci. 93, 1591, 1996.

[82] J. A. Sethian, *Theory, algorithms, and applications of level set methods for propagating interfaces*, In: Acta Numerica, Vol. 5, pp. 309–395, 1996.

[83] J. A. Sethian, *Level set methods and fast marching methods*, Cambridge University Press, 1999.

[84] SFB 540, *"Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems"*, `http://www.sfb540.rwth-aachen.de/`

[85] Silvester, D. Wathen, A.: Fast iterative solution of stabilised stokes systems. Part II: using general block preconditioners. SIAM J. Numer. Anal., **31**, 1994, 1352–1367.

[86] MG, A parallel multilevel platform for unstructured grids, `http://rcswww.urz.tu-dresden.de/~jstiller/projects/mg/`

[87] M. Sussman, P. Smereka, S. Osher, *A level set approach for computing solutions to incompressible two-phase flow*, J. Comp. Phys. 114, pp. 146–159, 1994.

[88] M. Sussman, A. S. Almgren, J. B. Bell, Ph. Colella, L. H. Howell, M. L. Welcome, *An adaptive level set approach for incompressible two-phase flows*, J. Comp. Phys. 148, pp. 81–124, 1999.

[89] Tecplot, Tecplot Inc., 2D/3D visualization tool, `http://www.tecplot.com/`

[90] A.A. Johnson, T.E. Tezduyar, *Mesh update strategies in parallel finite element computations of flow problems with moving boundaries and interfaces*, Comput. Methods Appl. Mech. Engrg. 119, pp. 73–94, 1994.

[91] A.-K. Tornberg, *Interface tracking methods with application to multiphase flows*, Doctoral Thesis, Royal Institute of Technology, Department of Numerical Analysis and Computing Science, Stockholm, 2000

[92] A.-K. Tornberg, B. Engquist, *A finite element based level-set method for multiphase flow applications*, Comput. Visual. Sci. 3, pp. 93–101, 2000.

[93] G. Tryggvason, B. Bunner, A. Esmaeeli, D. Juric, N. Al-Rawahi, W. Tauber, J. Han, S. Nas, Y.J. Jan, *A front-tracking method for the computations of multiphase flow*, J. Comp. Phys. 169, pp. 708–759, 2001.

[94] S. Turek, *Efficient solvers for incompressible flow problems: An algorithmic approach in view of computational aspects*, LNCSE Vol 6, Springer, Berlin, Heidelberg, 1999.

[95] UG, `http://cox.iwr.uni-heidelberg.de/~ug/`

[96] S.O. Unverdi, G. Tryggvason, *A front-tracking method for viscous, incompressible multi-fluid flows*, J. Comp. Phys. 100, pp. 25–37, 1992.

[97] S. Zhang, *Successive subdivisions of tetrahedra and multigrid methods on tetrahedral meshes*, Houston J. Math. 21, pp. 541–556, 1995.

[98] H. Zhou, C. Cristini, J. Lowengrub, C.W. Macosko, *Numerical simulation of deformable drops with soluble surfactant: Pair interactions and coalescence in shear flow*, Preprint 1898, Institute of Applied Mathematics, University of Minnesota, 2002.