# The DROPS package for numerical simulations of incompressible flows using parallel adaptive multigrid techniques

Sven Groß, Jörg Peters, Volker Reichelt, Arnold Reusken *

## 1 Introduction

In this paper we present an overview of a software package called DROPS, which has recently been developed at the IGPM (Institut für Geometrie und Praktische Mathematik) at the Aachen University of Technology. This development is still continuing and our aim is to build an efficient software tool for the numerical simulation of incompressible flows. In the field of incompressible CFD already quite a few packages exist. The state of the art, however, is such that for complicated three-dimensional fluid dynamics (e. g. turbulent flows, multiphase reacting flows, flows with free boundaries) a black-box solver is not yet available. The DROPS package is developed in an interdisciplinary project (SFB 540 "Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems", cf. [36]) where complicated flow phenomena are investigated. The modeling of several complex physical phenomena in this project (e. g., mass transfer between liquid drops and a surrounding fluid or the fluid dynamics and heat transport in a laminar falling film) requires a flexible efficient and robust CFD package. Our aim is to provide such a tool. From the scientific computing point of view it is of interest to develop a CFD code in which several modern numerical techniques which have recently been introduced in the literature are implemented. Examples of such techniques are error estimation methods for Navier-Stokes equations ([18, 33]), fast and robust iterative solvers for discretized Navier-Stokes equations with large Reynolds numbers ([15, 16]) and level set methods for two-phase flows ([37, 44]).

In this paper we describe the main components of the DROPS package and show results of a few first applications. At the end of the paper an outlook concerning the further development of the code in the near future is given.

We start with a brief description of the problem class that we consider. As landmarks for testing certain components in the code we use the Poisson equation, the convection-diffusion equation, the stationary Stokes problem and the stationary and time-dependent Navier-Stokes equations. We only consider *spatially three-dimensional* problems, i. e. we assume $\Omega \subset \mathbb{R}^3$ to be a polygonal Lipschitz domain. As a first test problem we consider the *Poisson equation* with homogeneous boundary conditions: given $f \in C(\overline{\Omega})$ determine $u = u(x)$ such that

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \,. \end{aligned} \tag{1}$$

The Poisson equation corresponds to a symmetric elliptic operator. A first test case in which (strong) nonsymmetry arises is the *convection-diffusion problem*. In strong formulation this

*Institut für Geometrie und Praktische Mathematik, RWTH Aachen, D-52056 Aachen, Germany

problem is as follows: given sufficiently smooth functions $b = (b_1, b_2, b_3)^T$, $a_0$, $f$ with $a_0 \geq 0$ and a scalar $\nu > 0$, determine $u = u(x)$ such that

$$\begin{aligned} -\nu\Delta u + (b \cdot \nabla)u + a_0 u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \,. \end{aligned} \tag{2}$$

We introduce the *stationary Stokes problem*: given sufficiently smooth functions $f = (f_1, f_2, f_3)^T$ and $a_0 \geq 0$, determine functions $u = (u_1(x), u_2(x), u_3(x))^T$ and $p = p(x)$ such that

$$\begin{aligned} -\Delta u + a_0 u + \nabla p &= f \quad \text{in } \Omega \\ \operatorname{div} u &= 0 \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \,. \end{aligned} \tag{3}$$

The instationary *Navier-Stokes problem* reads in strong formulation: given sufficiently smooth vector functions $f$, $u_0$ and a scalar $\nu > 0$, determine functions $u = (u_1(x,t), u_2(x,t), u_3(x,t))^T$ and $p = p(x,t)$ ($x \in \overline{\Omega}$, $t \in [0,T]$) such that

$$\begin{aligned} \frac{\partial u}{\partial t} - \nu\Delta u + (u \cdot \nabla)u + \nabla p &= f \quad \text{in } \Omega \times [0,T] \\ \operatorname{div} u &= 0 \quad \text{in } \Omega \times [0,T] \\ u &= 0 \quad \text{on } \partial\Omega \times [0,T] \\ u(\cdot, 0) &= u_0 \quad \text{on } \Omega \,. \end{aligned} \tag{4}$$

Below we will also use the weak formulations of the problems in (1)–(4).

The instationary Navier-Stokes equations are discretized in time by an implicit integration method which is explained in §3.3. In each time step one then has to solve an elliptic (continuous in space) boundary value problem. The nonlinearity $N(u)u = (u \cdot \nabla)u$ in the Navier-Stokes equations is often treated by a fixed point technique (cf. §4.3) in which $N(u)$ is replaced by $\tilde{u} \cdot \nabla$, where $\tilde{u}$ is a known approximation of the solution $u$. This implicit time integration and linearization of the instationary Navier-Stokes equations results in so called *Oseen equations* (or Stokes equations with convection). Such an Oseen problem is of the following form: given sufficiently smooth vector functions $f$ and $b$ and a scalar $a_0 \geq 0$, determine functions $u = (u_1(x), u_2(x), u_3(x))^T$ and $p = p(x)$ such that

$$\begin{aligned} -\Delta u + (b \cdot \nabla)u + a_0 u + \nabla p &= f \quad \text{in } \Omega \\ \operatorname{div} u &= 0 \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \,. \end{aligned} \tag{5}$$

For notional convenience we scaled the velocity $u$ such that $\nu = 1$. The linearization of a *stationary* Navier-Stokes problem results in an Oseen problem as in (5) with $a_0 = 0$.

For the numerical solution of the stationary elliptic boundary value problems in (1), (2), (3), (5) we apply an adaptive strategy as sketched in Figure 1. This diagram shows the following main building blocks of the solution method: grid generation and grid refinement, discretization method, iterative solvers for the discrete problem and error estimation techniques. We briefly comment on our choices for these components.

*Grid generation and grid refinement.* We only use tetrahedral grids. These grids are constructed in such a way that they are consistent (no hanging nodes) and that the hierarchy of triangulations is stable. The main ideas are taken from [5, 6]. A detailed discussion is given in §2.

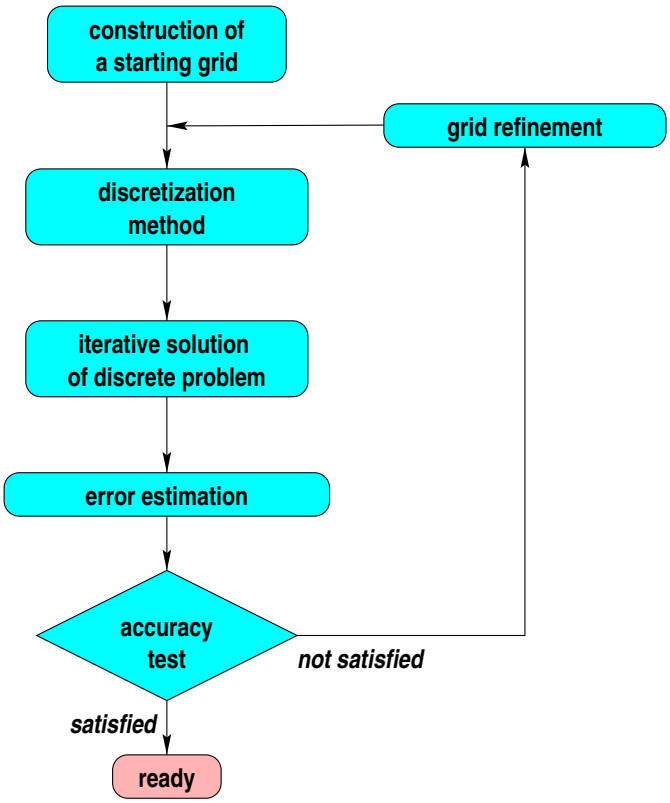


Figure 1: Adaptive solution strategy for stationary problems

*Discretization method.* We use finite element methods for the discretization of the elliptic boundary value problems. Up to now we only implemented low order conforming finite elements. Further explanation is given in §3. The discrete time integration is discussed in §3.3.

*Iterative solution methods.* For the scalar Poisson and convection-diffusion problems we use a multigrid solver. The (Navier-)Stokes equations are treated by a Schur complement (inexact Uzawa) technique. Details on these methods are presented in §4.

*Error estimation methods.* For the error estimation we use techniques from [39]. Up to now we implemented error estimators for the Poisson equation and for the stationary Stokes problem. We refer to §5 for further information.

We realize that for each of these choices there are one or more good alternatives which for certain problems might even be more appropriate. However, in view of the problem class that we want to treat and of the aims formulated in the interdisciplinary research project we believe that the approach sketched above results in a good compromise between simplicity, flexibility, efficiency and robustness.

Apart from the numerical building blocks that are outlined above there are other aspects which are of main importance for the performance of a CFD code. Here we mention the use of *suitable data structures* and the *implementation on parallel architectures*. One important decision we made related to data structures is to decouple the grid generation and finite element discretization (using a grid based data handling) as much as possible from the iterative solution methods (which use a sparse matrix format). Our code is programmed in C++ and uses several attractive facilities offered by this programming language. A discussion of certain important implementation issues is given in §4.5.

In many cases the numerical complexity of three dimensional flow simulations is extremely high. Hence in view of both memory requirements and computing times the implementation on a parallel machine can be very important. A good parallelization potential is an important objective in the development of the DROPS package. A first MPI based parallel version of DROPS has been implemented on a PC-cluster. A further explanation of parallelization issues is given in §6.

## 2 Grid generation

The first step in discretizing PDE's with finite elements is to provide suitable grids. We opted for *tetrahedral grids*, because they can handle complex geometries better than hexahedral grids and are much easier to deal with than hybrid grids consisting of combinations of, for example, tetrahedra, pyramids, prisms and hexahedra. In the remainder, a triangulation stands for a tetrahedral grid.

For the quality of the discretization two properties of the underlying triangulation are essential: consistency and stability. Consistency allows for conforming discretizations without taking special care of hanging nodes etc. In addition certain error estimators assume (at least in theory) consistent grids [39]. Stability means that in a hierarchy of triangulations all angles of the tetrahedra are uniformly bounded away from 0 and $\pi$. Note that stability is not a property of a single (non-degenerate) grid but of a sequence of (refined) grids which will be constructed in an adaptive discretization method and used in a multigrid solver. Error bounds (and therefore the accuracy of the solution) as well as error estimators and the LBB-stability for the discretized Stokes problem depend on the stability of the underlying triangulations.

The construction of a hierarchy of grids can be divided into two parts. In a first step a consistent initial grid must be constructed. This is done by DROPS only for fairly simple geometries. For more complex settings we plan to use standard (CAD) software. Appropriate modules that parse standard file formats and translate them into DROPS's internal representation will then be implemented. The second part is a refinement/unrefinement[1] algorithm that serves two purposes. Firstly, by using an appropriate adaptively refined grid a discretization error smaller than some given tolerance can be obtained without introducing too many unknowns. Secondly, it generates a hierarchy of grids that can be used for multigrid solvers. The refinement algorithm is a crucial component in the DROPS package. The algorithm we implemented is based on the one described in [4, 5] but contains several improvements.

To maintain consistency and stability the refinement algorithm uses two kinds of refinement rules.

For uniform refinement the "regular" or "red" rules are applied which subdivide a tetrahedron into 8 children such that all its edges are cut in half. This is accomplished by first taking out the four corners of the tetrahedron and then cutting the remaining octahedron twice which yields four additional tetrahedra (see Figure 2). The children belong to (up to) three congruency classes: The outer four tetrahedra belong to the same class as their father and the inner four belong to two different classes (in general). Note that the two transversal cuts through the octahedron share one diagonal which in principle offers three possible subdivisions of the octahedron since there are three diagonals. All of them lead to consistent triangulations of the enclosing tetrahedron. If one carefully picks the appropriate diagonals when subdividing the children and their children etc. using the "red" rule, it can be shown that all tetrahedra in the hierarchy of triangulations *belong to at most three different congruency classes* (see [6]). Using this strategy we are able to perform uniform refinement in a stable (due to the finite number of congruency classes) and consistent fashion.

[1] for convenience we will call it simply "refinement algorithm"

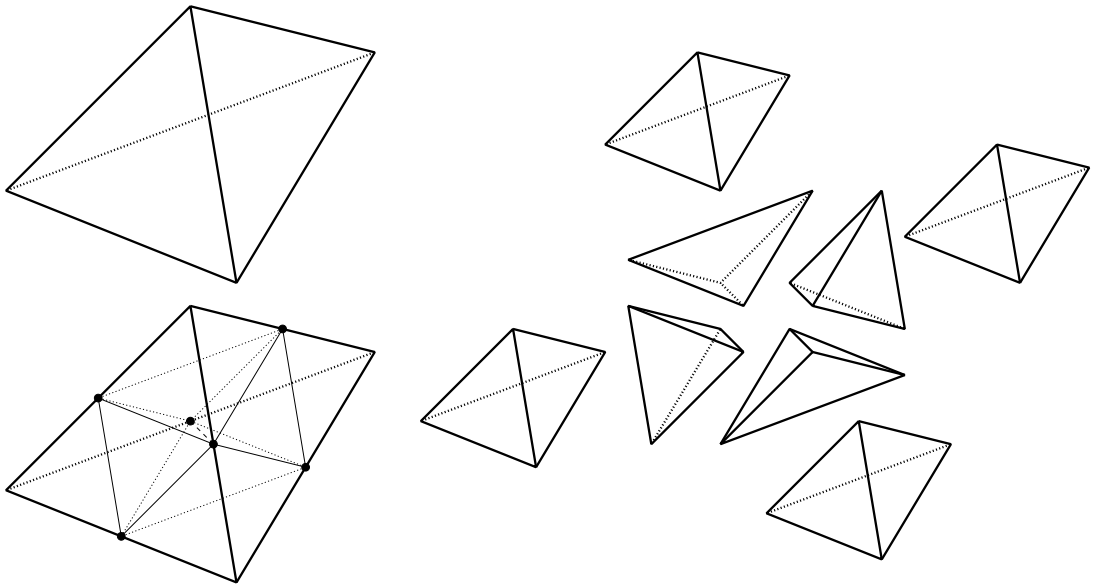

Figure 2: Regular refinement of a tetrahedron

For non-uniform refinement, different levels of refinement have to be combined in a stable and consistent way. This is done by the "irregular" or "green" rules that generate tetrahedra which are refined on some faces/edges but remain unrefined on others (see examples in Figure 3). The resulting children can connect tetrahedra that differ by one level of refinement. If we do not allow "green" children (i. e. children that were generated by green rules) to be further refined, we still have a finite number of possible congruency classes (due to the finite number of green rules). Hence, stability is preserved. The non-trivial task of the refinement algorithm is to apply the rules in such a fashion that green children remain unrefined and consistency is guaranteed.

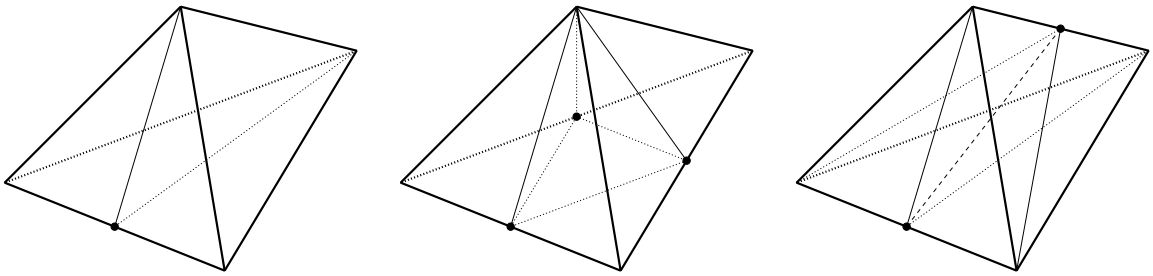

Figure 3: Examples of irregular refinement

For the algorithm the tetrahedra are organized in so called levels. Level $\mathcal{L}_0$ contains the tetrahedra of the initial grid, level $\mathcal{L}_1$ all the children, $\mathcal{L}_2$ all the grandchildren etc. up to some final level $\mathcal{L}_{\overline{\ell}}$. The triangulation $\mathcal{G}_\ell$ (for $\ell = 0, \ldots, \overline{\ell}$) then consists of all tetrahedra from level $\mathcal{L}_\ell$ and the tetrahedra from the levels below that do not have any children. The triangulations $\mathcal{G}_0, \ldots, \mathcal{G}_{\overline{\ell}}$ form a set of nested grids that are suitable for the application of multigrid methods. The levels, however, are more natural for the refinement algorithm.

In addition to the tetrahedra the edges play an important role in the algorithm because of the following consideration. If one subdivides a tetrahedron or a face, one has to divide some of its neighbours, too, to maintain consistency. However, if one refines an edge, no neighbouring edge is affected — edges can be subdivided independently. This is exploited by the algorithm.

Let us assume, that an error estimator has marked some of the tetrahedra of the finest

triangulation $\mathcal{G}_{\overline{\ell}}$ for refinement and some others for coarsening.[2] The structure of the refinement algorithm — which basically consists of two loops — is as follows:

- *A loop from the finest level $\mathcal{L}_{\overline{\ell}}$ to the coarsest $\mathcal{L}_0$.* It does not actually add or remove tetrahedra, but modifies the marks on the edges, that determine whether the edge should be refined or not. Its inner loop iterates over all tetrahedra on the current level. All 6 edges of the current tetrahedron will be refined, if one of the following three conditions holds:

  **1.** The current tetrahedron is not green and marked for refinement by the error estimator.

  **2.** The current tetrahedron is refined irregularly and one of its children is marked for refinement (so that the green rule will be replaced by a red rule).

  **3.** If a child of a neighbour shall be refined (to ensure that the levels of refinement differ by at most 1).

  If all children of a tetrahedron are marked for removement, the edges of the tetrahedron will be unrefined (if that is permitted by the neighbours).

- *A loop from the coarsest level $\mathcal{L}_0$ to the finest $\mathcal{L}_{\overline{\ell}}$.* In this loop the construction and destruction of tetrahedra actually takes place: If the marks on the edges do not coincide with the actual refinement pattern of a tetrahedron, the children will be deleted and new children (if required) will be generated.

This strategy is illustrated for the 1D case in Figure 4. In Figure 4a) we have four grid cells on level 0. The inner ones are refined regularly (this is denoted by the fat line on top), their children are on level 1. The outer two grid cells are refined irregularly, their green children (which connect two levels of refinement) are on level 1, too. In b) one cell is marked for refinement. In c) we start the top-down loop on level 1: The mark on the grid cell is translated into a refinement pattern (thus the fat line on top). In d) we go to level 0. Because of rule **3.** from above the leftmost cell has to be refined, since its right neighbour has a child that will be refined. In e) we start the bottom-up loop on level 0: The leftmost cell has a regular refinement pattern (the fat line), but its child is a green one. Therefore the green child is deleted and replaced by two regular children. On level 1 in f) we have to add the missing children, thus generating level 2.

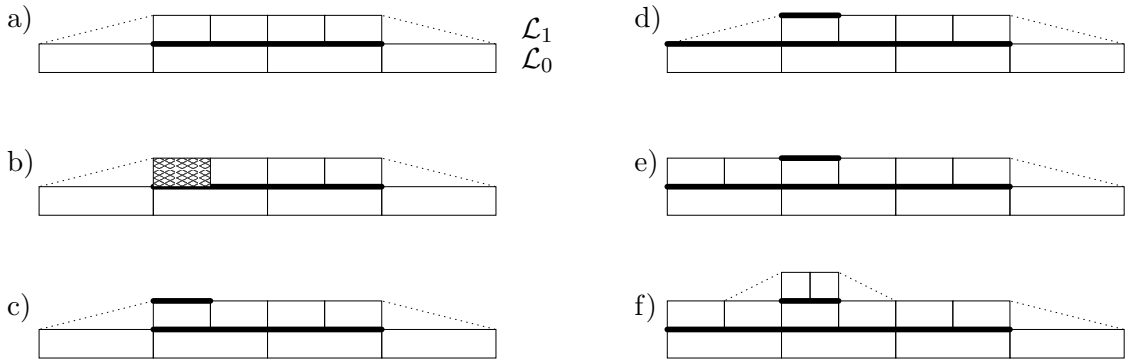


Figure 4: The refinement algorithm – 1D example

There is, however, a dangerous pitfall. The refinement rule for each new tetrahedron must be determined from the refinement patterns on its edges. For some patterns there are several possible matching refinement rules. Consider the situation in Figure 5: If two of the three edges of a face are marked for refinement, there are two ways to cut the face into smaller triangles accordingly.

[2] Note that because no tetrahedron in $\mathcal{G}_{\overline{\ell}}$ has children only tetrahedra without children will be marked.

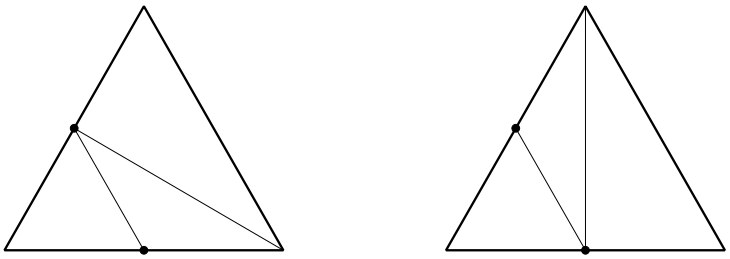

Figure 5: Non-unique refinement

If two tetrahedra share such a face one has to choose their refinement rules so that both cut this face in the same way to maintain consistency. One might think that just prescribing the pattern on all those faces would solve the problem, but this is only part of the truth, because it is possible to have a pattern on all four faces of a tetrahedron in such a way that no acceptable refinement rule can match all the patterns on the faces. Since this situation would lead to an inconsistent triangulation it has to be avoided!

In [4, 5] this problem does not arise, because only a limited set of green rules is used: In complicated cases green refinement is replaced by red refinement. The disadvantage is that the neighbours have to be adjusted to match the new rule. This in turn can affect their neighbours and so on. Since such a domino effect hinders parallelization we favor a different solution.

Our approach to deal with this problem is as follows. We use an arbitrary global numbering of the vertices of the tetrahedra in level $\mathcal{L}_0$ to determine an ordering of the vertices in each tetrahedron of this level. Our set of refinement rules not only tells how to cut a tetrahedron into pieces, but it also induces an ordering of the vertices in each of the children. This results in an ordering of the vertices of each tetrahedron in each level. If a situation as in Figure 5 occurs we choose the pattern that connects the vertex with the lowest number and the midpoint of the opposite edge. Since the ordering is consistent between neighbouring tetrahedra, the choice of the patterns will be, too. By using this strategy to determine the patterns on the faces we can always find a rule to match the patterns and we never get the "forbidden" situation mentioned above. In fact there is precisely one rule for every pattern on the edges, so that we need a set of $2^6 = 64$ refinement rules. In addition there are two positive side-effects. One can determine the appropriate refinement rule without considering the neighbour — this reduces the communication in a parallel algorithm. Furthermore, when applying the red rule one automatically chooses the appropriate diagonal for subdivision of the inner octahedra.

**Remark 1** In the refinement algorithm any operation on a specific tetrahedron etc. is determined by/affects only some small neighbourhood of this tetrahedron. This locality property is very favorable for parallelization,

The locality property of the refinement algorithm also yields an interesting result related to adaptivity. A refinement mark for some tetrahedron will affect only a small part of the triangulation. Hence, only a small number of new tetrahedra/unknowns will be generated. In combination with an appropriate local error estimator this allows for a fine-tuned error control.

**Remark 2** We use vertices, edges, faces and tetrahedra as data structures which allows a very "geometric" implementation of the refinement algorithm described above. This also enables us to store geometry-related data at the appropriate places, which facilitates the implementation of finite element discretizations.

**Remark 3** The refinement rules are stored as data sets for each rule that are parsed whenever new tetrahedra have to be generated. Each set contains the number of the children, the order and

location (relative to the father) of their vertices, edges and faces. To determine the appropriate rule we interpret the given refinement pattern on the edges as 6-bit binary number which we use as index into the data base.

# 3 Discretization

## 3.1 Finite element discretization of scalar stationary problems

We consider the weak formulation of the *Poisson equation* with homogeneous boundary conditions (1): given $f \in L_2(\Omega)$,

$$\text{find} \quad u \in H_0^1(\Omega) \quad \text{such that} \quad a(u,v) = (f,v) \quad \text{for all} \quad v \in H_0^1(\Omega), \tag{6}$$

where $a(u,v) := \int_\Omega \nabla u \cdot \nabla v$ and $(\cdot,\cdot)$ is the scalar product on $L_2(\Omega)$. The finite element approach is a special case of a Galerkin method where we replace the infinite dimensional space $V := H_0^1(\Omega)$ by a space $V_h \subset V$ of finite dimension:

$$\text{Find} \quad u_h \in V_h \quad \text{such that} \quad a(u_h,v_h) = (f,v_h) \quad \text{for all} \quad v_h \in V_h. \tag{7}$$

Given a basis $\Phi := \{\varphi_1, \ldots, \varphi_n\}$ of $V_h$ and using the decomposition $u_h = \sum_{j=1}^n \alpha_j \varphi_j$ with $\alpha_j \in \mathbb{R}$, we get the following linear system of equations:

$$\text{Find} \quad \alpha_1, \ldots, \alpha_n \quad \text{such that} \quad \sum_{j=1}^n a(\varphi_j,\varphi_i)\,\alpha_j = (f,\varphi_i) \quad \text{for all} \quad i = 1, \ldots, n.$$

Using the notation $\mathbf{A} := (a(\varphi_j,\varphi_i))_{i,j=1,\ldots,n}$, $\mathbf{b} := ((f,\varphi_1), \ldots, (f,\varphi_n))^T$, and $\mathbf{x} := (\alpha_1, \ldots, \alpha_n)^T$ we can reformulate the problem as

$$\text{Find} \quad \mathbf{x} \in \mathbb{R}^n \quad \text{such that} \quad \mathbf{A}\mathbf{x} = \mathbf{b}\ . \tag{8}$$

The bilinear form $a$ is symmetric and elliptic, hence the stiffness matrix $\mathbf{A}$ is symmetric positive definite. This property can be exploited by iterative solvers.

Let $u$ and $u_h$ be the solutions of (6) and (7), respectively. For $v \in V$ we introduce the energy norm $\|v\|_1 := a(v,v)^{1/2}$. An important theoretical result is Cea's Lemma (see [30]), which for the Poisson equation yields the discretization error bound

$$\|u_h - u\|_1 \le \inf_{v_h \in V_h} \|v_h - u\|_1\ . \tag{9}$$

This means that the discretization error is directly related to approximation properties of the space $V_h$.

Hence, two obvious criteria for the choice of $V_h$ and its basis $\Phi$ are obtained: $u$ should be approximated by functions from $V_h$ "very accurately" and $\Phi$ should be chosen such that the matrix $\mathbf{A}$ has "nice" properties.

We chose one of the most popular methods to achieve these goals, namely *finite elements*. Given a consistent triangulation $\mathcal{G} = \{T\}$ of $\Omega$ the space $V_h$ consists of all functions $v_h$ on $\Omega$, such that $v_h|_T$ is a polynomial for any given tetrahedron $T \in \mathcal{G}$. The choices for the triangulation (e. g. tetrahedral, hexahedral), the polynomials (e. g. linear, quadratic) and the smoothness of the functions (e. g. continuous, differentiable) determine the type of the finite elements.

On the tetrahedral triangulations used in DROPS we implemented the following two types of elements: $P_1$-element (linear polynomials) and $P_2$-element (quadratic polynomials). One can interpret the polynomials as interpolating polynomials with nodes as shown in Figure 6. The

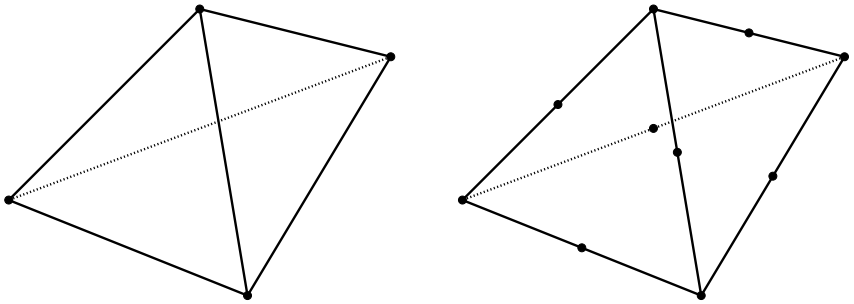

Figure 6: $P_1$-element (4 nodes) and $P_2$-element (10 nodes)

location of the nodes automatically guarantees that the polynomials on different tetrahedra are continuous across edges and faces.

Based on the nodes we can easily construct the so called *nodal basis* for $V_h$. It consists of all functions in $V_h$ that have the value 1 at one node and zero at all the others. These functions are linearly independent and span $V_h$, so that they indeed form a basis for $V_h$. In addition it is easy to decompose any given function in $V_h$ with respect to this basis. A further important property is that the nodal functions have local support. Hence, the matrix $\mathbf{A}$ is sparse and we only have to integrate over few tetrahedra to compute $a(\varphi_j, \varphi_i)$ and $(f, \varphi_i)$.

Using Cea's Lemma and approximation properties of polynomials one can for example show the following error bound (see [30]) for $P_1$-elements on a uniform grid of step size $h$:

$$\|u_h - u\|_{L_2} \leq ch^2. \tag{10}$$

However, for many problems uniform refinement is not optimal. Actually, Cea's Lemma already suggests the use of adaptive grids. It would be a waste of computational time to put many nodes into those parts of $\Omega$ where $u$ could be approximated almost just as good with much fewer nodes. Note that in an adaptive discretization method with strong local refinements an error bound as in (10) makes no sense because there is no reasonable global mesh size parameter $h$.

Assembling the stiffness matrix $\mathbf{A}$ is done by iterating over all tetrahedra of a given triangulation $\mathcal{G}$ and computing their share of the integrals. This can of course be done in parallel in a straightforward manner. To compute the entries of $\mathbf{A}$ we just have to integrate polynomials, so that we can provide the exact values. To compute the entries $(f, \varphi_i)$ of the right hand side $\mathbf{b}$ we have to use quadrature formulas, since in general $f$ is not a piecewise polynomial. In case of $P_1$-elements, for example, DROPS uses a quadrature formula that evaluates the integrand in the four nodes and the barycenter of the tetrahedron $T$. This guarantees that the local error is of order $\mathcal{O}(\mathrm{diam}(T)^2)$.

The treatment of certain boundary conditions different from homogeneous Dirichlet conditions is also implemented in DROPS. In the case of nonhomogeneous Dirichlet conditions we simply take into account the known values at nodes that lie on the boundary. In the case of Neumann boundary conditions the values at the boundary nodes are treated as additional unknowns.

We briefly discuss the finite element discretization method for the convection-diffusion equation (2). The weak formulation of this problem is as in (7) with the bilinear form $a$ replaced by

$$\tilde{a}(u, v) := \nu\, a(u, v) + \int_\Omega b \cdot \nabla u\, v + \int_\Omega a_0\, u\, v$$

We use the same finite element spaces as for the Poisson equation. This is reasonable only if the problem is *not* convection-dominated, i. e. if $\max_{i=1,2,3} \|b_i\|_{\infty,\Omega} \lesssim \nu$ holds. Up to now we

*restricted ourselves to diffusion dominated problems*. In the near future we plan to implement stabilization techniques like streamline diffusion finite elements (SDFEM) which then allow a proper finite element discretization of convection dominated problems (cf. §8).

For the convection-diffusion problem the assembling of the stiffness matrix is very similar to the assembling of the Poisson stiffness matrix $\mathbf{A}$. One only has to compute a few additional integrals. Note that for the convection-diffusion problem the stiffness matrix is nonsymmetric. For the diffusion dominated case, however, we can solve the resulting discrete problem using the same iterative solvers as for the discrete Poisson equation.

**Remark 4** For iterative solvers like Krylov subspace methods we only need the matrix of the linear system (8) resulting from the discretization on the finest grid $\mathcal{G}_{\overline{\ell}}$. For multigrid solvers, however, we also need the hierarchy of grids. In addition we have to provide prolongations and restrictions. The prolongations are obtained from the natural embedding of a coarse finite element space in the next finer one. The prolongation is constructed by projecting the coarse grid piecewise polynomial nodal functions to the next finer grid and decomposing them into the respective fine grid basis functions. The restrictions are merely the transposed prolongations, so we get them for free (and can save the computer memory).

### 3.2 Discretization of stationary systems

We consider the stationary Stokes equations. Using the notation $V := H_0^1(\Omega)^3$, $Q := \{\, q \in L_2(\Omega) \mid \int_\Omega q = 0 \,\}$ the weak formulation of the Stokes problem (3) reads: Given $f \in L_2(\Omega)^3$ and a sufficiently smooth function $a_0 : \Omega \to [0, \infty)$,

$$\text{find } u \in V,\ p \in Q \text{ such that } \quad \begin{cases} a(u,v) + (a_0 u, v) + b(v,p) = (f,v) & \text{for all } v \in V \\ b(u,q) = 0 & \text{for all } q \in Q, \end{cases} \tag{11}$$

where $a(u,v) := (\nabla u, \nabla v) = \sum_{i,j=1}^3 (\frac{\partial u_i}{\partial x_j}, \frac{\partial v_i}{\partial x_j})$ for $u, v \in V$ and $b(u,q) := -(\operatorname{div} u, q)$ for $u \in V$, $q \in Q$.

For the discretization we again use the Galerkin approach with finite element spaces $V_h \subset V$ and $Q_h \subset Q$. The discrete problem is obtained by replacing $V$ by $V_h$ and $Q$ by $Q_h$ in (11). For the discrete problem to be well-posed (for $h \downarrow 0$) the pair of spaces $V_h$, $Q_h$ must satisfy the famous *inf-sup* (or LBB) condition (see [19]):

$$\inf_{q_h \in Q_h} \sup_{v_h \in V_h} \frac{b(v_h, q_h)}{\|v_h\|_1 \cdot \|q_h\|_{L_2}} \geq \beta \tag{12}$$

where $\beta$ is some positive constant independent of $h$. There is extensive literature on finite element spaces that satisfy this condition. A pair that is known to work (see [19]) and used in DROPS is $P_1$-elements for the pressure unknowns ($Q_h$) and $P_2$-elements for the velocity unknowns ($V_h$).

Using the nodal bases of the spaces $V_h$ and $Q_h$ the discrete problem can be formulated as a linear system of the form

$$\begin{pmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ 0 \end{pmatrix} . \tag{13}$$

The matrix of this system is symmetric but strongly indefinite. The matrix $\mathbf{A}$ is symmetric positive definite. Special iterative solvers for this type of system are discussed in §4.2.

For the representation of the stiffness matrix in (13) we store the matrices $\mathbf{A}$ and $\mathbf{B}$ separately. This simplifies the implementation of iterative solvers that exploit the special block structure of the stiffness matrix in (13).

The Oseen equations (5) can be written in weak formulation and a finite element discretization method as discussed above can be applied. The discrete problem is of the same form as in (13), but now the matrix $\mathbf{A}$ is nonsymmetric due to the convection term.

### 3.3 Time discretization for instationary problems

We treat instationary problems by first discretizing with respect to time. For each time step the resulting problem can be seen as a stationary equation for which the discretization methods discussed in §3.2 can be used.

The time-dependent Navier-Stokes equations (4) can be represented as follows (where for simplicity we omit the initial/boundary conditions):

$$\begin{aligned} \tfrac{\partial u}{\partial t} &= f - F(u)u - \nabla p \\ \operatorname{div} u &= 0 \end{aligned}$$

with $F(u) := -\nu\Delta + N(u)$. Replacing $\frac{\partial u}{\partial t}$ by a forward finite difference with time step $\tau$ we obtain the following explicit Euler scheme:

$$\begin{aligned} \tfrac{u^{i+1}-u^i}{\tau} &= f^i - F(u^i)u^i - \nabla p^i \\ \operatorname{div} u^{i+1} &= 0. \end{aligned}$$

Here $u^i$ denotes the known velocity field at time $t_i$ and $u^{i+1}$ the unknown field at time $t_{i+1} = t_i + \tau$. This scheme is only first order accurate, i. e. the discretization error is $\mathcal{O}(\tau)$. Furthermore, the scheme is not $A$-stable. In many CFD applications the equations are very stiff and then the lack of $A$-stability causes strong limitations on the time step. This is considered to be major drawback of the explicit Euler scheme. A generalization of this Euler method is the so called $\theta$-scheme ($\theta \in [0,1]$):

$$\begin{aligned} \tfrac{u^{i+1}-u^i}{\tau} &= \theta[f^{i+1} - F(u^{i+1})u^{i+1} - \nabla p^{i+1}] + (1-\theta)[f^i - F(u^i)u^i - \nabla p^i] \\ \operatorname{div} u^{i+1} &= 0. \end{aligned}$$

The parameter $\theta$ controls the implicitness of the scheme. For $\theta = 0$ we obtain the explicit Euler method. Two other well-known examples are the implicit (or backward) Euler scheme ($\theta = 1$) and the Crank-Nicholson scheme ($\theta = \frac{1}{2}$). The first offers $A$-stability (even strong $A$-stability), but is still of order one. The second is of order two, but does not have the strong $A$-stability property, which leads to stability problems in certain situations.[3]

In DROPS we use a similar second order method which, however, is strongly $A$-stable. This so called fractional-step method can be found in [38]. Each time step is subdivided into three substeps with step sizes $\mu\tau$, $\mu'\tau$ and $\mu\tau$ where $\mu := 1 - \frac{\sqrt{2}}{2} \approx 0.293$ and $\mu' := 1 - 2\mu = \sqrt{2} - 1 \approx 0.414$:

$$\begin{aligned} \tfrac{u^{i+\mu}-u^i}{\mu\tau} &= f^i - \theta F(u^{i+\mu})u^{i+\mu} - (1-\theta)F(u^i)u^i - \nabla p^{i+\mu} \\ \operatorname{div} u^{i+\mu} &= 0 \\ \tfrac{u^{i+1-\mu}-u^{i+\mu}}{\mu'\tau} &= f^{i+1-\mu} - \theta F(u^{i+\mu})u^{i+\mu} - (1-\theta)F(u^{i+1-\mu})u^{i+1-\mu} - \nabla p^{i+1-\mu} \\ \operatorname{div} u^{i+1-\mu} &= 0 \\ \tfrac{u^{i+1}-u^{i+1-\mu}}{\mu\tau} &= f^{i+1-\mu} - \theta F(u^{i+1})u^{i+1} - (1-\theta)F(u^{i+1-\mu})u^{i+1-\mu} - \nabla p^{i+1} \\ \operatorname{div} u^{i+1} &= 0. \end{aligned}$$

[3] In spite of this we implemented the $\theta$-schemes to be able to perform numerical comparisons.

Because of the special choice of $\mu$ it is a second order scheme. The parameter $\theta$ is chosen such that $\theta\mu = (1-\theta)\mu'$ holds ($\theta = 2 - \sqrt{2} \approx 0.586$). The scheme can be shown to be strongly A-stable (cf. [38]).

For the implementation we have to collect the terms with the unknown velocities and pressures on the left hand side of the equations and the rest is treated as a source term on the right hand side ($\theta' := 1 - \theta$):

$$\begin{aligned}
\left[I + \theta\mu\tau F(u^{i+\mu})\right] u^{i+\mu} + \mu\tau\nabla p^{i+\mu} &= \left[I - \theta'\mu\tau F(u^i)\right] u^i + \mu\tau f^i \\
\operatorname{div} u^{i+\mu} &= 0 \\
\\
\left[I + \theta'\mu'\tau F(u^{i+1-\mu})\right] u^{i+1-\mu} + \mu'\tau\nabla p^{i+1-\mu} &= \left[I - \theta\mu'\tau F(u^{i+\mu})\right] u^{i+\mu} + \mu'\tau f^{i+1-\mu} \\
\operatorname{div} u^{i+1-\mu} &= 0 \\
\\
\left[I + \theta\mu\tau F(u^{i+1})\right] u^{i+1} + \mu\tau\nabla p^{i+1} &= \left[I - \theta'\mu\tau F(u^{i+1-\mu})\right] u^{i+1-\mu} + \mu\tau f^{i+1-\mu} \\
\operatorname{div} u^{i+1} &= 0
\end{aligned}$$

These equations can be written in weak formulation and then discretized w. r. t. space as before. The discretization of the identity operator $I$ then results in a mass matrix $\mathbf{M} := ((\varphi_j, \varphi_i)_{i,j=1,\ldots,n_v}) \in \mathbb{R}^{n_v \times n_v}$. In the implementation the boundary conditions (at time $t_i$ and $t_{i+1}$) must be treated carefully.

# 4 Iterative solvers

## 4.1 Solvers for the Poisson equation

In this section we discuss iterative solvers for the Poisson equation, as they are implemented in DROPS, namely the preconditioned conjugate gradient method (PCG) and the multigrid method (MG). The discretization of the Poisson equation leads to a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ as in (8), with a stiffness matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ that is symmetric positive definite. For this type of problems PCG methods are very popular iterative solvers. The presentation of the PCG algorithm is skipped, as it is well known and can be found in the literature (e. g. [22]). In each iteration of the PCG algorithm one matrix-vector multiplication with the matrix $\mathbf{A}$ has to be computed and a linear system of the form

$$\mathbf{W}\tilde{\mathbf{x}} = \mathbf{r} \tag{14}$$

must be solved, where $\mathbf{W}$ is the preconditioner. Here we use an SSOR technique for preconditioning. Let $\mathbf{A}$ be decomposed as $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$ where $\mathbf{D} = \operatorname{diag}(\mathbf{A})$ and $\mathbf{L}$, $\mathbf{U}$ are strictly lower and strictly upper triangular matrices, respectively. The SSOR preconditioner is given by

$$\mathbf{W} = \frac{1}{\omega(2-\omega)}(\mathbf{D} - \omega\mathbf{L})\mathbf{D}^{-1}(\mathbf{D} - \omega\mathbf{U}) \ . \tag{15}$$

The parameter $\omega \in (0, 2)$ is user-defined. For the case $\omega = 1$ one obtains the symmetric Gauss-Seidel preconditioner. The solution $\tilde{\mathbf{x}}$ of (14) is given by

$$\begin{aligned}
\tilde{x}_i^0 &:= \omega/a_{ii}\left(r_i - \textstyle\sum_{j<i} a_{ij}\tilde{x}_j^0\right), && i = 1, \ldots, n \\
\tilde{x}_i^1 &:= (2-\omega)\tilde{x}_i^0 - \omega/a_{ii}\textstyle\sum_{j>i} a_{ij}\tilde{x}_j^1, && i = n, n-1, \ldots, 1 \\
\tilde{\mathbf{x}} &:= (\tilde{x}_1^1, \tilde{x}_2^1, \ldots, \tilde{x}_n^1)^T \ .
\end{aligned}$$

For the *multigrid method*, a hierarchy of grids (triangulations)

$$\mathcal{G}_0 \subset \mathcal{G}_1 \subset \cdots \subset \mathcal{G}_{\overline{\ell}}$$

and corresponding matrices

$$\mathbf{A}_\ell \in \mathbb{R}^{n_\ell \times n_\ell}, \quad \ell = 0, \cdots, \overline{\ell},$$

as well as prolongations

$$\mathbf{P}_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \quad \ell = 1, \cdots, \overline{\ell},$$

are needed. The restrictions $\mathbf{R}_\ell \in \mathbb{R}^{n_{\ell-1} \times n_\ell}$, $\ell = 1, \cdots, \overline{\ell}$, are defined by $\mathbf{R}_\ell := \mathbf{P}_\ell^T$.

The idea of multigrid methods is to smooth the error (or defect) so that it can be represented accurately on a coarser grid by restriction, and then solve a coarse grid error equation. Optionally, a post-smoothing follows, as we do for the sake of symmetry. For the solution of the coarse grid problem, the same idea can be applied recursively. For an introduction to multigrid methods we refer to [22]. The multigrid algorithm has the following structure:

**Algorithm 1 (Multigrid method)**

*function* $\mathrm{MGM}_\ell(\mathbf{x}_\ell, \mathbf{b}_\ell)$
{
  *if* $l = 0$ *then*
    $\mathbf{x}_0 := \mathbf{A}_0^{-1}\mathbf{b}_0$; // solve coarse grid problem
  *else*
  {
    $\mathbf{x}_\ell := S_\ell^\nu(\mathbf{x}_\ell, \mathbf{b}_\ell)$; // presmoothing
    $\mathbf{d}_{\ell-1} := \mathbf{R}_\ell\,(\mathbf{b}_\ell - \mathbf{A}_\ell \mathbf{x}_\ell)$; // restriction of defect
    $\mathbf{e}_{\ell-1}^0 := 0$;
    *for* $i = 1$ *to* $\tau$ *do* // recursion
      $\mathbf{e}_{\ell-1}^i := \mathrm{MGM}_{\ell-1}(\mathbf{e}_{\ell-1}^{i-1}, \mathbf{d}_{\ell-1})$;
    $\mathbf{x}_\ell := \mathbf{x}_\ell + \mathbf{P}_\ell\,\mathbf{e}_{\ell-1}^\tau$; // add coarse grid correction
    $\mathbf{x}_\ell := S_\ell^\nu(\mathbf{x}_\ell, \mathbf{b}_\ell)$; // postsmoothing
  }
  *return* $\mathbf{x}_\ell$;
}

In our applications we use $\tau = 1$ (V-cycle) and $\nu \in \{1, 2, 3\}$ smoothing steps, where $S$ is one step of the symmetric Gauss-Seidel method.

An important difference between the PCG and MG method concerns the dependence of the rate of convergence on the mesh size parameter $h$. When $h$ is reduced the PCG method will typically need more iterations to obtain a certain error reduction. This behaviour is predicted by the following theoretical result (cf. [22]). Let $\mathbf{x}_0$ be the starting vector of the PCG algorithm and $\mathbf{x}_k$ the result after $k$ iterations. For the error in the energy norm $\|\mathbf{y}\|_{\mathbf{A}}^2 := \mathbf{y}^T\mathbf{A}\mathbf{y}$ the inequality

$$\|\mathbf{x}_k - \mathbf{x}\|_{\mathbf{A}} \leq 2\Big(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\Big)^k \|\mathbf{x}_0 - \mathbf{x}\|_{\mathbf{A}} \tag{16}$$

holds, with $\kappa = \mathrm{cond}(\mathbf{W}^{-1}\mathbf{A})$ the spectral condition number of the preconditioned matrix. In our applications we typically have $\kappa \to \infty$ for $h \downarrow 0$. For example for the Poisson equation discretized with $P_1$ finite elements on a uniform triangulation and using SSOR preconditioning one has

$$\mathrm{cond}(\mathbf{W}^{-1}\mathbf{A}) = \mathcal{O}(h^{-2}) \quad \text{for } h \downarrow 0\ . \tag{17}$$

Hence, for this case if the mesh size is halved one expects a doubling of the number of PCG iterations needed to obtain a given error reduction.

The multigrid method is a linear iterative method and thus the (asymptotic) convergence rate is determined by the spectral radius of the iteration matrix, which is denoted by $\rho_{\mathrm{MG}}$. In [21, 22] it is shown that for a large class of elliptic boundary value problems the multigrid method applied on a hierarchy of regularly refined grids satisfies the inequality

$$\rho_{\mathrm{MG}} \leq \alpha < 1,$$

with a constant $\alpha$ independent from $h$. In [9] similar theoretical results are shown to hold for an adaptive setting in which multigrid is applied on a hierarchy of locally refined grids (cf. also [43]). These results give a theoretical explanation for the fact that in many practical applications the multigrid convergence rate does not deteriorate when the grid is (locally) refined. This can also be seen in the numerical examples in §7.

Of course, for the efficiency of an iterative solver apart from the rate of convergence the arithmetic work per iteration has to be taken into account, too. For the PCG method with SSOR preconditioning the work per iteration is comparable to 2–3 sparse matrix-vector multiplications $\mathbf{A} * \mathbf{x}$. We briefly discuss the arithmetic work needed in one multigrid V-cycle iteration. As a convenient unit of arithmetic work we use MV, which stands for the work needed in one $\mathbf{A} * \mathbf{x}$ computation on the finest level $\overline{\ell}$. Let $n_\ell$ be the number of unknowns in the discrete problem $\mathbf{A}_\ell \mathbf{x}_\ell = \mathbf{b}_\ell$ (in a scalar problem with linear finite elements this is approximately the same as the number of vertices in the triangulation $\mathcal{G}_\ell$). We introduce $\theta_\ell := \frac{n_{\ell-1}}{n_\ell}$ $(1 \leq \ell \leq \overline{\ell})$, which measures the reduction of the dimension of the discrete problem when going from level $\ell$ to $\ell - 1$. In case of global uniform refinement one has $\theta_\ell \approx \frac{1}{8}$. In an adaptive setting with local refinement one has $\frac{1}{8} \leq \theta_\ell < 1$. The value of $\theta_\ell$ is strongly influenced by the marking strategy that is used in the refinement algorithm (cf. §5.3). The arithmetic work in the multigrid V-cycle on level $\overline{\ell}$ is denoted by $W_{\mathrm{MGM}_{\overline{\ell}}}$. For the costs of one multigrid V-cycle one obtains (cf. [22])

$$W_{\mathrm{MGM}_{\overline{\ell}}} \approx \big(1 + \theta_{\overline{\ell}} + \theta_{\overline{\ell}}\theta_{\overline{\ell}-1} + \theta_{\overline{\ell}}\theta_{\overline{\ell}-1}\theta_{\overline{\ell}-2} + \ldots + \theta_{\overline{\ell}}\theta_{\overline{\ell}-1}\ldots\theta_1\big) 2(\nu+1)\mathrm{MV} + W_0 \ .$$

Here $\nu$ is the number of smoothing iterations used in the multigrid algorithm and $W_0$ the work needed for solving the problem on the coarsest grid, $\mathbf{A}_0\mathbf{x}_0 = \mathbf{b}_0$. This work can be considered to be negligible compared to one MV. If we assume $\theta_\ell \leq \theta < 1$ for all $\ell$ and neglect $W_0$ we obtain

$$W_{\mathrm{MGM}_{\overline{\ell}}} \approx \frac{2}{1-\theta}(\nu+1)\mathrm{MV} \ .$$

Hence the arithmetic costs for the multigrid V-cycle are proportional to a sparse matrix-vector multiplication, $\mathbf{A} * \mathbf{x}$, on the finest grid: $W_{\mathrm{MGM}_{\overline{\ell}}} \approx C\,\mathrm{MV}$. The constant $C$ depends on the number of smoothing iterations used and on the "rate of refinement" (parameter $\theta$). In particular, if from one level to the next finer one only relatively few new vertices (unknowns) are added the arithmetic costs per iteration can be quite high. Hence, we use a marking strategy which tries to avoid this.

**Remark 5** The finite element discretization of the convection-diffusion problem (2) results in a linear system with a nonsymmetric stiffness matrix. Up to now we only considered diffusion dominated problems. For such problems the stiffness matrix is a perturbation of a symmetric positive definite matrix and the PCG method described above still works quite well. For problems with stronger convection the CG method has to be replaced by other Krylov subspace methods (cf. §8). Standard multigrid methods can be applied to diffusion dominated convection-diffusion equation. For the convection-dominated case special tools like robust smoothers or matrix-dependent prolongations and restrictions should be used (cf. [35]).

### 4.2 Solvers for the stationary Stokes and Oseen equation

The discretization of the stationary Stokes equation leads to the saddle point problem

$$\begin{aligned} \mathbf{A}\mathbf{x} + \mathbf{B}^T\mathbf{y} &= \mathbf{b}, \\ \mathbf{B}\mathbf{x} &= \mathbf{c}. \end{aligned}$$

with a symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n_v \times n_v}$ and $\mathbf{B} \in \mathbb{R}^{n_p \times n_v}$. In order to decouple the equations, we multiply the first one with $\mathbf{BA^{-1}}$ from the left, which results in

$$\mathbf{Sy} = \mathbf{BA}^{-1}\mathbf{b} - \mathbf{c} \qquad \text{with the Schur complement} \qquad \mathbf{S} := \mathbf{BA}^{-1}\mathbf{B}^T.$$

This leads to

**Algorithm 2 (Schur complement method)**

1. *Solve* $\mathbf{Az} = \mathbf{b}$.
2. *Solve* $\mathbf{Sy} = \mathbf{Bz} - \mathbf{c}$.
3. *Solve* $\mathbf{Ax} = \mathbf{b} - \mathbf{B}^T\mathbf{y}$.

For the systems with matrix $\mathbf{A}$ in steps 1 and 3 one can use the PCG method described above. The Schur complement matrix $\mathbf{S}$ is symmetric positive definite, too, and hence we can apply the CG method. Every multiplication with $\mathbf{S}$ in the CG algorithm involves solving a linear system with matrix $\mathbf{A}$ (plus multiplications with $\mathbf{B}$ and $\mathbf{B}^T$). The inner iteration for solving this $\mathbf{A}$-system has to be performed with high accuracy for the CG-solver to converge (we use $\text{res}_{\text{outer}} < 10^{-10}$, $\text{res}_{\text{inner}} < 10^{-14}$ for instance). Note that finding a preconditioner for the matrix $\mathbf{S}$ is a difficult task because this matrix is not explicitly available. The SSOR preconditioner, for example, cannot be applied. Fortunately, for many problems the condition number of $\mathbf{S}$ is moderate and remains bounded for $h \downarrow 0$ (cf. [10]). In general the efficiency of the CG-solver for the Schur complement problem in step 2 can be improved by preconditioning the system with the mass matrix $\mathbf{M} \in \mathbb{R}^{n_p \times n_p}$ for the pressure unknowns. For preconditioning we use one SSOR iteration (i. e., the matrix $\mathbf{W}$ in (15)) applied to the matrix $\mathbf{M}$.

A disadvantage of the Schur complement technique is that the inner $\mathbf{A}$-systems in step 2 must be solved with high accuracy. A more popular approach which avoids this is the so called inexact Uzawa method (cf. [10]).

**Algorithm 3 (Inexact Uzawa method)**
*Repeat until desired accuracy:*

$$\mathbf{x}_{i+1} := \mathbf{x}_i + \tilde{\mathbf{A}}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_i - \mathbf{B}^T\mathbf{y}_i), \tag{18}$$

$$\mathbf{y}_{i+1} := \mathbf{y}_i + \tilde{\mathbf{S}}^{-1}(\mathbf{B}\mathbf{x}_{i+1} - \mathbf{c}). \tag{19}$$

In this method one uses preconditioners $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{S}}$ of $\mathbf{A}$ and $\mathbf{S}$, respectively (this is why this Uzawa method is called *inexact*). In our applications the correction term in (18) is the result of several iterations of a Poisson solver (multigrid or PCG) with starting vector $\mathbf{0}$ applied to the system

$$\mathbf{Az} = \mathbf{b} - \mathbf{Ax}_i - \mathbf{B}^T\mathbf{y}_i \ . \tag{20}$$

For the correction term in (19) we apply a few iterations of an iterative solver (e. g. PCG) with starting vector $\mathbf{0}$ applied to

$$\mathbf{Mz} = \mathbf{Bx}_{i+1} - \mathbf{c} \ . \tag{21}$$

We briefly discuss an interesting theoretical result from [10] concerning the rate of convergence of the inexact Uzawa method. We assume symmetric positive definite preconditioners $\tilde{\mathbf{A}}$ of $\mathbf{A}$

and $\tilde{\mathbf{S}}$ of the Schur complement $\mathbf{S}$. We assume that $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{S}}$ are scaled such that $\tilde{\mathbf{A}} - \mathbf{A}$ and $\tilde{\mathbf{S}} - \mathbf{S}$ are positive semidefinite. Furthermore, let $\sigma_A$, $\sigma_S \in [0, 1)$ be such that

$$\begin{aligned} (1 - \sigma_A)(\tilde{\mathbf{A}}\mathbf{x}, \mathbf{x}) &\leq (\mathbf{A}\mathbf{x}, \mathbf{x}) \quad \text{for all } \mathbf{x} , \\ (1 - \sigma_S)(\tilde{\mathbf{S}}\mathbf{y}, \mathbf{y}) &\leq (\mathbf{S}\mathbf{y}, \mathbf{y}) \quad \text{for all } \mathbf{y} . \end{aligned}$$

We use the notation $(\cdot, \cdot)$ for the Euclidean scalar product on $\mathbb{R}^n$. Note that since $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{S}}$ are positive definite such $\sigma_A$ and $\sigma_S$ always exist. In [10] it is shown that for the error $\mathbf{e}_i := \begin{pmatrix} \mathbf{x} - \mathbf{x}_i \\ \mathbf{y} - \mathbf{y}_i \end{pmatrix}$ the inequality

$$[|\mathbf{e}_i|] \leq \rho^i [|\mathbf{e}_0|] \quad \text{for } \; i = 0, 1, 2, \ldots$$

holds, where $[|\cdot|]$ is some problem dependent norm and

$$\rho = \frac{\sigma_S(1 - \sigma_A) + \sqrt{\sigma_S^2(1 - \sigma_A)^2 + 4\sigma_A}}{2} \leq 1 - \frac{1}{2}(1 - \sigma_S)(1 - \sigma_A) .$$

From this result we conclude that *the rate of convergence of the inexact Uzawa method is high if one uses good preconditioners* $\tilde{\mathbf{A}}$ *and* $\tilde{\mathbf{S}}$.

**Remark 6** We briefly discuss a result from [45] related to the inexact Uzawa method. Let

$$\mathbf{K} = \begin{pmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & 0 \end{pmatrix}, \quad \tilde{\mathbf{K}} = \begin{pmatrix} \tilde{\mathbf{A}} & 0 \\ \mathbf{B} & \tilde{\mathbf{S}} \end{pmatrix}, \qquad \overline{\mathbf{x}} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}, \quad \overline{\mathbf{b}} = \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix}, \quad \overline{\mathbf{x}}_i = \begin{pmatrix} \mathbf{x}_i \\ \mathbf{y}_i \end{pmatrix} .$$

The inexact Uzawa iteration (18),(19) can be rewritten as

$$\overline{\mathbf{x}}_{i+1} = \overline{\mathbf{x}}_i - \tilde{\mathbf{K}}^{-1}\big(\mathbf{K}\overline{\mathbf{x}}_i - \overline{\mathbf{b}}\big) . \tag{22}$$

Hence the inexact Uzawa method can be seen as a *Richardson method* for solving $\tilde{\mathbf{K}}^{-1}\mathbf{K}\overline{\mathbf{x}} = \tilde{\mathbf{K}}^{-1}\overline{\mathbf{b}}$.

We assume that the preconditioners $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{S}}$ are symmetric positive definite and that $\mathbf{A} - \tilde{\mathbf{A}}$ is positive definite. Then it can be shown that the preconditioned matrix $\tilde{\mathbf{K}}^{-1}\mathbf{K}$ is symmetric positive definite with respect to the scalar product

$$\left\langle \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}, \begin{pmatrix} \tilde{\mathbf{x}} \\ \tilde{\mathbf{y}} \end{pmatrix} \right\rangle_* := (\,(\mathbf{A} - \tilde{\mathbf{A}})\mathbf{x}, \tilde{\mathbf{x}}\,) + (\tilde{\mathbf{S}}\mathbf{y}, \tilde{\mathbf{y}}).$$

From this it follows that we can apply a CG method to the preconditioned system $\tilde{\mathbf{K}}^{-1}\mathbf{K}\overline{\mathbf{x}} = \tilde{\mathbf{K}}^{-1}\overline{\mathbf{b}}$. This results in a method with a (significantly) higher rate of convergence than the Uzawa (= Richardson) method in (22).

We consider the Oseen problem resulting from linearization of the stationary Navier-Stokes equation, i. e., (5) with $a_0 = 0$. Finite element discretization of this problem results in a linear system of the form

$$\begin{aligned} (\mathbf{A} + \mathbf{C})\mathbf{x} + \mathbf{B}^T\mathbf{y} &= \mathbf{b}, \\ \mathbf{B}\mathbf{x} &= \mathbf{c}. \end{aligned} \tag{23}$$

The matrix $\mathbf{A}$ corresponds to the Laplace operator and is symmetric positive definite. The matrix $\mathbf{C}$ is nonsymmetric and results from the discretization of the convection term in the Oseen equation. The matrix $\mathbf{A} + \mathbf{C}$ has (after permutation) a simple block diagonal structure:

$\mathbf{A} + \mathbf{C} = \text{blockdiag}(\mathbf{A}_C, \mathbf{A}_C, \mathbf{A}_C)$. The diagonal blocks $\mathbf{A}_C$ are the discretization of a scalar convection-diffusion problem as in (2).

Since we only consider problems which are diffusion dominated (Navier-Stokes with small Reynolds number) the linear system in (23) can be considered to be of saddle point type and the Schur complement and inexact Uzawa methods can be used as solvers. The linear problems that arise in step 1 and 3 of the Schur complement algorithm and in (20) are now of convection-diffusion type.

### 4.3 Solvers for the stationary Navier-Stokes equation

The discretization of the stationary Navier-Stokes equation leads to a nonlinear system of the form

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

For linearization of this system we use a fixed point iteration technique. The resulting linear problems are as in (23) and can be treated by the iterative methods from §4.2. In the linearization method we use a standard step size control technique. The following algorithm is from [38]:

**Algorithm 4 (Fixed point defect correction method with step size control)**
*Set* $\omega_0 = 1$.
*Repeat until desired accuracy:*

1. *Calculate the defect vector*

$$\begin{pmatrix} \mathbf{resx} \\ \mathbf{resy} \end{pmatrix} := \begin{pmatrix} \mathbf{A}\mathbf{x}_i + \mathbf{N}(\mathbf{x}_i)\mathbf{x}_i + \mathbf{B}^T\mathbf{y}_i - \mathbf{b} \\ \mathbf{B}\mathbf{x}_i - \mathbf{c} \end{pmatrix}$$

2. *Solve the discrete Oseen problem*

$$\begin{aligned} [\mathbf{A} + \mathbf{N}(\mathbf{x}_i)]\,\mathbf{v} + \mathbf{B}^T\mathbf{q} &= \mathbf{resx} \\ \mathbf{B}\mathbf{v} &= \mathbf{resy} \end{aligned}$$

   *with accuracy* $\text{tol}_i$ *to obtain the correction* $\mathbf{v}$ *and* $\mathbf{q}$.

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

$$\omega_{i+1} := \frac{\left\langle \mathbf{K}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix},\quad \mathbf{K}\begin{pmatrix}\mathbf{x}_i\\ \mathbf{y}_i\end{pmatrix} - \begin{pmatrix}\mathbf{b}\\ \mathbf{c}\end{pmatrix} \right\rangle}{\left\langle \mathbf{K}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix},\quad \mathbf{K}\begin{pmatrix}\mathbf{v}\\ \mathbf{q}\end{pmatrix} \right\rangle} \qquad \textit{with} \qquad \mathbf{K} := \begin{pmatrix} \mathbf{A} + \mathbf{N}(\mathbf{x}_i - \omega_i\mathbf{v}) & \mathbf{B}^T \\ \mathbf{B} & 0 \end{pmatrix}$$

4. *Update* $\mathbf{x}_i, \mathbf{y}_i$*:*

$$\begin{pmatrix} \mathbf{x}_{i+1} \\ \mathbf{y}_{i+1} \end{pmatrix} := \begin{pmatrix} \mathbf{x}_i \\ \mathbf{y}_i \end{pmatrix} - \omega_{i+1} \begin{pmatrix} \mathbf{v} \\ \mathbf{q} \end{pmatrix}$$

Step 3 may be skipped, using $\omega_{i+1} := 1$ in step 4, resulting in a simpler linearization method (cf. [31]). The use of step size control, however, improves the robustness of the algorithm. By far most of the computational work is done in step 2. The accuracy $\text{tol}_i$ of the solver in step 2 is increased during the outer iteration.

### 4.4 Solvers for the time-dependent Navier-Stokes equation

In the instationary Navier-Stokes equation the fractional step method from §3.3 is used for time discretization. In each time step a quasi stationary Navier-Stokes problem arises. Discretization of these problems yields nonlinear systems of the form

$$\begin{aligned} \mathbf{A}\mathbf{x} + \mathbf{N}(\mathbf{x})\mathbf{x} + \mathbf{M}\mathbf{x} + \mathbf{B}^T\mathbf{y} &= \mathbf{b} \\ \mathbf{B}\mathbf{x} &= \mathbf{c}. \end{aligned} \tag{24}$$

These nonlinear systems can be treated by the method given in algorithm 4. Note that in this time dependent case the Oseen problems in step 2 of the algorithm are discrete counterparts of the Oseen problem (5) with $a_0 > 0$. The reaction term $a_0 u$ in this Oseen problem ($\mathbf{M}\mathbf{x}$ in (24)) comes from the identity operator $I$ on the left handside of the fractional step method. We summarize the solution process:

- *Outer loop*: time stepping with fractional step scheme
- *Second loop*: fixed point defect correction method for linearization
- *Third loop*: inexact Uzawa (or Schur complement) method to solve discrete Oseen problems
- *Inner loop*: iterative solver (PCG, MG) for the convection-diffusion problems

This is only a rough description of the algorithm we implemented since we completely left out the construction of the systems (24), i. e. the whole discretization part. For adaptivity we have to add another loop that contains error estimation and grid refinement.

### 4.5 Implementation issues

In this section we describe important data structures (matrices and vectors), the numerical solvers are based on. Furthermore the design of the solvers is briefly addressed.

The vector data structure is mainly a wrapper class around a `std::valarray<double>` object, a standard container from the C++ Standard Template Library (STL) especially designed for the purpose of numerical applications.

Since the matrices arising from the discretization are sparse, we use an appropriate matrix storage format, the CRS (compressed row storage) format, in which only nonzero entries are stored. It contains an array for the values of the nonzero entries and two auxiliary `integer` arrays that define the position of the entries within the matrix. Consider an $m \times n$ matrix with $l$ nonzero entries. The latter are stored successively row by row in the data array of length $l$. In addition their column indices are stored in the first index array which is, of course, also of length $l$. Because the data are stored row-wise, we do not have to provide a row number for each nonzero entry. Instead we only store where a new row begins (and the last one ends). This is done by means of a second index array, which is of length $m + 1$.

The memory requirement for a sparse $n \times n$ stiffness matrix is largely determined by the size of the value array ($l$ `doubles`), which is proportional to $n$. This results in an $\mathcal{O}(n)$ memory requirement. Also the arithmetic work for a matrix-vector product computation is roughly proportional to $n$.

Unfortunately, the nice computational and storage properties of the CRS format are not for free. A disadvantage of this format is that insertion of a non-zero element into the matrix is rather expensive — usually $\mathcal{O}(l)$ elements and column indices have to be moved. Since this is unacceptable when building the matrix in the discretization step, we designed a sparse matrix builder class with an intermediate storage format that offers write access in $\mathcal{O}(\log l)$ time for each element. Afterwards, the matrix is converted into the CRS format.

We conclude with a note on the design of the iterative solvers. Since the (Navier-)Stokes solvers and the time integration schemes require the solution of inner problems (e. g. of Poisson- or convection-diffusion type), the solvers have been encapsulated in template classes, where the template parameter controls the inner solver used. All solver classes offer a standardized interface such that it is easy to use other inner solvers. This technique allows a simple way to compare different solvers for a given problem or to use a library of solvers which contains different methods which are appropriate for special problem classes.

# 5 A posteriori error estimators and refinement strategies

For reliable and efficient numerical simulations the topic of error estimation is of major importance. *A priori estimates*, which are known for many problems, usually yield bounds for the discretization error that are very pessimistic and should not be used for error estimation. Moreover, these a priori results bound the global (i. e. over the whole domain) average of the error, for example, the $L_2(\Omega)$ norm of the error as in (10). Hence, these a priori results cannot be used for adaptivity, i. e., for local grid refinement in those parts of the domain, where the error is relatively large.

Error estimation techniques that use a computed discrete approximation of the continuous solution are called *a posteriori estimators*. These methods overcome the above-mentioned drawbacks of a priori techniques. In this section we discuss the main ideas related to the a posteriori estimators we use.

Let $\mathcal{G} = \{T\}$ be a consistent triangulation. Let $u$ be the solution of the continuous problem and $u_h$ be an approximation of $u$ which results from a (finite element) discretization on the triangulation $\mathcal{G}$. We are interested in local error estimates $\eta_T$ ($T \in \mathcal{G}$), that give a good indication of the size of the error on $T$, i. e., $\eta_T \approx \|u - u_h\|_{*,\omega_T}$. Here $\|\cdot\|_{*,\omega_T}$ with $\omega_T \supset T$ is some suitable norm (examples are given below), which measures the discretization error $u - u_h$ on a small subdomain $\omega_T$, that contains $T$, but is not much larger than $T$.

For local error estimators the following properties are important:

- *Locality.* This means that the error estimates $\eta_T$ give a reasonable indication of the size of the *local* error $(u - u_h)|_T$. An error estimator which has this property can be used as an indicator in a local refinement method.

- *Reliability.* For this property to hold an inequality of the form

$$\|u - u_h\|_{*,\Omega} \leq C \sqrt{\sum_{T \in \mathcal{G}} \eta_T^2}$$

  must be fulfilled with a moderate constant $C$ independent of $\operatorname{diam}(T)$ for all $T \in \mathcal{G}$. It is useful for global error control, i. e., for checking a tolerance bound of the form $\|u - u_h\|_{*,\Omega} \leq$ eps, where eps is a user-defined parameter.

- *Efficiency.* This stands for the existance of a bound of the form

$$\eta_T \leq C \|u - u_h\|_{*,\omega_T}$$

  for the error estimator. As above, $C$ should be a moderate constant that does not depend on $\operatorname{diam}(T)$. Thereby we can guarantee that if the actual local error $\|u - u_h\|_{*,\omega_T}$ is relatively small, the error estimator $\eta_T$ will also be small. Thus, a refinement strategy based on the relative size of $\eta_T$ will not refine in regions with relatively small errors.

- *Low computational complexity.* The arithmetic work for computing $\eta_T$ for all $T \in \mathcal{G}$ should not exceed the arithmetic work needed in the discretization method or the iterative solver.

Until now we only implemented *residual error estimators* (cf. [39]) in DROPS. These methods are based on the following very general result:

**Theorem 1** *Let $(X, \|\cdot\|_X)$, $(Y, \|\cdot\|_Y)$ be two Banach spaces and $L : X \to Y$ be a linear continuous bijective mapping. For $g \in Y$ let $u$ be the solution of $Lu = g$. Then the error bounds*

$$\|L\|_{X\to Y}^{-1}\|L\tilde{u} - g\|_Y \leq \|\tilde{u} - u\|_X \leq \|L^{-1}\|_{Y\to X}\|L\tilde{u} - g\|_Y$$

*hold for all $\tilde{u} \in X$.*

**Remark 7** The result of this theorem can be seen as a generalization of the following very elementary fact. Consider a linear system $\mathbf{Ax} = \mathbf{b}$ with a nonsingular matrix $\mathbf{A}$. Let $\tilde{\mathbf{x}}$ be an approximation of the solution $\mathbf{x}$ with *residual* $\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b} =: \tilde{\mathbf{b}} - \mathbf{b}$. Then the following holds:

$$\|\mathbf{A}\|^{-1}\|\tilde{\mathbf{b}} - \mathbf{b}\| \leq \|\tilde{\mathbf{x}} - \mathbf{x}\| \leq \|\mathbf{A}^{-1}\|\|\tilde{\mathbf{b}} - \mathbf{b}\| \ .$$

**Remark 8** The Poisson equation in weak formulation can be cast into the general setting of Theorem 1 if one takes

$$\begin{aligned} &X = H_0^1(\Omega), \quad \|u\|_X^2 = \int_\Omega \nabla u \nabla u, \quad Y = X' \quad \text{(dual space of } X\text{)}, \\ &L : X \to Y, \quad L(u)(v) := \int_\Omega \nabla u \nabla v, \quad g(v) := \int_\Omega fv \ . \end{aligned}$$

For this special case one obtains $\|L\|_{X\to Y}^{-1} = \|L^{-1}\|_{Y\to X} = 1$.

The main task in the development of residual error estimators for a class of partial differential equations (e. g. Poisson equation or the Stokes equation) is the construction of efficient approximation methods for the norm of the residual $L\tilde{u} - g$. How this can be done, can be found in [1] or [39], for example. In the following two sections we present some concrete methods for the Poisson and for the Stokes equation.

## 5.1 Residual error estimator for the Poisson equation

Here the general result of Theorem 1 is applied with $X = H_0^1(\Omega)$, $Y = X'$ (cf. Remark 8). In [39] it is shown how the estimation of the residual $\|Lu_h - g\|_Y$ (where $u_h$ is the solution of the discrete problem (7)) can be made computable by using jumps of the gradient of $u_h$. For simplicity, we only present a result for the case of linear finite elements.

For an arbitrary tetrahedron $T$ or face $F$ the diameter of the circumcircle of the simplex is denoted by $d_T$ and $d_F$, respectively. We introduce

$$\begin{aligned} \omega_T &:= T \cup \tilde{\omega}_T, \quad \tilde{\omega}_T := \bigcup_{T' \cap T \in \mathcal{F}_h} T', \\ \mathcal{F}_h &:= \{F \mid F \text{ is a face of some } T \in \mathcal{G}\} \ . \end{aligned}$$

Note that $\tilde{\omega}_T$ is the union of tetrahedra in $\mathcal{G}$, which have one common face with $T$. Also, for any function $f \in L_2(\Omega)$ let $f_T$ be the piecewise constant function given by

$$f_T(x) = \begin{cases} \frac{1}{|T|}\int_T f & \text{for} \quad x \in T \\ 0 & \text{otherwise} \ . \end{cases}$$

Every face $F \in \mathcal{F}$ is assigned a normal vector $n_F$ that shall point out of $\Omega$, if $F \in \partial\Omega$. Let $F \in \mathcal{F}$ be an interior face and $g$ be a scalar function that is continuous on each of the two tetrahedra that share $F$ as a common face. The jump of $g$ across $F$ is defined by

$$[g]_F(x) = \lim_{t \downarrow 0} g(x + tn_F) - \lim_{t \downarrow 0} g(x - tn_F) \qquad (x \in F).$$

Using these definitions we can now introduce a *residual error estimator* for the Poisson equation:

$$\eta_{R,T} := \sqrt{d_T^2 \|f_T\|_{L_2(T)}^2 + \frac{1}{2} \sum_{F \in \mathcal{F}_h \cap \mathcal{F}(T)} d_F \| \left[n_F \cdot \operatorname{grad} u_h\right]_F \|_{L_2(F)}^2}. \tag{25}$$

Here, $\mathcal{F}(T)$ denotes the set of all faces of $T$, which lie in the interior of $\Omega$. For the computation of $\eta_{R,T}$ only quantities from $T$ and from the neighbouring tetrahedra $T' \subseteq \tilde{\omega}_T$ are needed. Note that for linear finite elements $\operatorname{grad} u_h$ is a piecewise constant function. Hence, the computation of $\eta_{R,T}$ is quite simple and the arithmetic costs are acceptable.

In the following theorem from [39] two important results for this residual error estimator are stated.

**Theorem 2** *Let $u$ be the solution of the Poisson equation (6) and $u_h$ the solution of the discrete problem (7) with linear finite elements. There are constants $c, C > 0$ that depend only on the smallest ratio of incircle- to circumcircle-radius of any $T \in \mathcal{G}$ such that the following estimates hold:*

$$\|u - u_h\|_{H_0^1(\Omega)} \leq C \sqrt{\sum_{T \in \mathcal{G}} \eta_{R,T}^2 + \sum_{T \in \mathcal{G}} d_T^2 \|f - f_T\|_{L_2(T)}^2} \tag{26}$$

$$\eta_{R,T} \leq c \sqrt{\|u - u_h\|_{H^1(\omega_T)}^2 + \sum_{T' \in \omega_T} \|f - f_{T'}\|_{L_2(T')}^2} \tag{27}$$

The terms with $\|f - f_T\|$ on the right hand side in equations (26) and (27) describe the data approximation error. Without exact integration of $f$ in $\eta_{R,T}$ these cannot be avoided, but for $d_T \downarrow 0$ they are in general small compared to the other terms on the right hand side. From the inequalities in the preceding theorem it follows that the error estimator $\eta_{R,T}$ is *reliable and efficient*. Results of numerical experiments using $\eta_{R,T}$ are given in §7.

DROPS also offers a version of this estimator, that can deal with Neumann boundary values. For this case, integrals over the faces lying on the Neumann boundary must be calculated in $\eta_{R,T}$, and the estimates in theorem 2 then contain a term, which takes the data approximation error on the Neumann boundary into account.

In [39] additional methods are described, which allow an estimation of the discretization error in the $L_2(\Omega)$ norm. One such technique is implemented in DROPS, too.

### 5.2 Residual estimator for the Stokes equations

In this section we briefly describe a residual error estimator for the $P_2$-$P_1$ finite element discretization of the Stokes equation. For an a posteriori estimator that has been applied to the Mini-element discretization in DROPS, we refer to [40].

Let $V := H_0^1(\Omega)^3$ and $Q := \{\, q \in L_2(\Omega) \mid \int_\Omega q = 0 \,\}$ be as in §3.2. The Stokes problem in weak formulation (11) can be analyzed in the setting of theorem 1 if one takes

$$\begin{aligned}
&X = V \times Q, \quad Y = X' \quad \text{(dual space of } X\text{)},\\
&\|(u,p)^T\|_{\mathrm{X}} = \sqrt{\|u\|_{H^1(\Omega)}^2 + \|p\|_{L_2(\Omega)}^2},\\
&L(u,p;v,q) := a(u,v) + (a_0 u, v) + b(v,p) + b(u,q), \quad g(v,q) := (f,v).
\end{aligned}$$

The definitions of $d_T$, $d_F$, $\omega_T$, $\mathcal{F}_h$, $\mathcal{F}(T)$ and $n_F$ are the same as in section 5.1; $f_T$ and $[f]_F$ can be defined componentwise for functions taking values in $\mathbb{R}^3$.

With this notation an error estimator for the discrete Stokes problem formulated in section 3.2 can be defined as follows (where for simplicity we assume $a_0 = 0$):

$$\begin{aligned} \eta_{R,T} \quad = \quad & \Big\{ d_T^2 \| - \Delta u_h + \operatorname{grad} p_h - f_T \|_{L_2(T)}^2 \\ & + \sum_{F \in \mathcal{F}_h \cap \mathcal{F}(T)} d_F \| \left[ \nu \, n_F \operatorname{grad} u_h - n_F \, p_h \right]_F \|_{L_2(F)}^2 \\ & + \| \operatorname{div} u_h \|_{L_2(T)}^2 \Big\}^{\frac{1}{2}} \end{aligned} \tag{28}$$

As for the Poisson equation the arithmetic costs of this error estimator are acceptable. The following result concerning reliability and efficiency can be shown to hold:

**Theorem 3** *Let $(u,p)^T$ be the solution of the Stokes problem (11) and $(u_h, p_h)^T$ be the discrete solution that is obtained by a $P_2$-$P_1$ finite element discretization. There are constants $c, c_2, C, C_2 > 0$ that depend only on the smallest ratio of incircle- to circumcircle- radius of any $T \in \mathcal{G}$ such that the following inequalities hold:*

$$\| (u,p)^T - (u_h, p_h)^T \|_{\mathrm{X}} \leq C \sqrt{\sum_{T \in \mathcal{G}} \eta_{R,T}^2} + C_2 \sqrt{\sum_{T \in \mathcal{G}} d_T^2 \| f - f_T \|_{L_2(T)}^2} \tag{29}$$

$$\eta_{R,T} \leq c \sqrt{\| u - u_h \|_{H^1(\omega_T)}^2 + \| p - p_h \|_{L_2(\omega_T)}^2} + c_2 \sqrt{\sum_{T' \in \omega_T} \| f - f_{T'} \|_{L_2(T')}^2} \tag{30}$$

As in Theorem 2 the bounds contain a data approximation error that is in general negligible on fine triangulations.

### 5.3 Marking Strategies

An a posteriori error estimator yields a positive real number for each tetrehedron $T \in \mathcal{G}$: $\{ (\eta_T, T) \}_{T \in \mathcal{G}}$. The number $\eta_T$ is an estimate for the size of the discretization error on (a small neighborhood of) $T$. Based on this set of local error estimates one needs a method for deciding, which tetrahedra should be marked for refinement or removement. Such a method is called a *marking strategy*. For DROPS three different algorithms have been implemented so far.

Presently, we only use grid refinement and do not consider grid coarsening. Hence, it suffices to have a marking scheme, which does not take coarsening into account. In a setting with time dependent problems one must have a marking strategy that is capable of coarsening, too.

We describe the three different marking strategies that are implemented in DROPS.

#### Threshold technique

In this method the user chooses a threshold value $w_{high}$ and calling the routine *EstimateError* (currently only implemented for Poisson equation) all unrefined tetrahedra $T$ fulfilling $\eta_T > w_{high}$ are marked for refinement. Note that by defining a similar threshold $w_{low}$ this method can easily be extended to take coarsening into account, too.

A disadvantage of this simple technique is that it is often not clear how one should choose an appropriate threshold value $w_{high}$.

**Error equilibration**

To improve the above technique, a second marking strategy is implemented, which tries to distribute the error equally over the domain $\Omega$. In an initialization step the global error estimate $E := (\sum_{T\in\mathcal{G}} \eta_T^2)^{\frac{1}{2}}$ is computed and based on a user specified factor $0 < red \leq 1$ a target value $err_{\text{tar}} = red\, E\, |\Omega|^{-\frac{1}{2}}$ is computed. Now, every tetrahedron, for which $\eta_T\, |T|^{-\frac{1}{2}} > err_{\text{tar}}$ holds, is marked for refinement. Note that if $\eta_T\, |T|^{-\frac{1}{2}} \approx err_{\text{tar}}$ holds for all $T$ we obtain $(\sum_{T\in\mathcal{G}} \eta_T^2)^{\frac{1}{2}} \approx red\, E$. Unfortunately, this method has the tendency to generate rather uniform refinements.

**Largest error first strategy**

This marking strategy is used in [14]. As in the previous method, the global error estimate $E := (\sum_{T\in\mathcal{G}} \eta_T^2)^{\frac{1}{2}}$ is computed. We assume a user defined parameter value $err_{\text{rel}} \in (0, 1)$. The local estimates $\eta_T$ are sorted by decreasing value and the correspondingly sorted tetrahedra are denoted by $T_1, T_2, \ldots, T_m$. Then, the tetrahedra with the largest error estimates $T_1, \ldots, T_k$ are marked such that

$$\sqrt{\sum_{j\leq k} \eta_{T_j}^2} \approx err_{\text{rel}}\, E$$

holds. This procedure is applied in each iteration of the adaptive solution strategy from figure 1. In this way, the regions, in which refinement takes place, are responsible for a relatively large amount of the global error. Coarsening could be performed via an additional threshold for marking those tetrahedra with the smallest errors for removement. We use this method as default marking strategy in DROPS.

# 6 Parallelization

For numerical simulations of instationary spatially three dimensional flows one often has to solve problems with very high numerical complexity. Hence, parallelization is an important issue — it increases the amount of available memory and can significantly reduce computation time. As a consequence, DROPS was already designed with parallelization in mind.

This does not mean that the whole code is parallelized already, but this should be possible without major changes. By now, the data structures can be managed in parallel so that a grid that was built on one processor can be distributed among the others by means of a graph partitioner (ParMetis [29]). We can discretize the Poisson equation on this distributed grid and solve the linear system using a parallel CG solver.

The parallel implementation of the refinement algorithm is about to be finished so that each part of the grid can be refined on the processor where it is stored. This eliminates the restriction that the whole grid has to fit into the memory of a single processor. The next step is the parallelization of the multigrid algorithm and the expansion to the parallel numerical treatment of the Stokes/Navier-Stokes equations.

The parallelization is based on the library DDD [7, 8], that was originally designed for the parallelization of the software package UG [3]. DDD can use several different parallel interfaces — we opted for MPI (message passing interface) which is the quasi-standard for distributed memory machines. The functionality of DDD is sketched in §6.3.

We start with a brief discussion of a few basic notions related to parallelization.

A distributed memory machine basically is a number of processors, each with its own memory, that are connected via a network. Data transfer between processors (so called inter-processor

communication) is carried out by calling library routines — MPI is such a library. Communication is very time-consuming compared to floating point operations so that it should be minimized. In addition messages should be bundled to save startup time caused by the network's latency.

The quality of a parallel code is measured by its scalability:

**Definition 1 (Speedup, efficiency, scalability)**

- *The speedup $S(P)$ is the time profit of a parallel algorithm running on $P$ processors compared to the serial version for a fixed problem size:*

$$S(P) = \frac{T(1)}{T(P)}.$$

  *$T(j)$ denotes the computation time of the algorithm on $j$ processors.*

- *The (parallel) efficiency $E(P)$ is defined as*

$$E(P) = \frac{S(P)}{P}.$$

- *An algorithm is called scalable, if $E(P)$ is close to 1, assuming a sufficiently large problem size.*

PDE solvers typically spend most of their time solving linear systems, which mainly reduces to performing matrix-vector-multiplications in the case of iterative solvers. Hence, parallelizing the matrix-vector-multiplication is a crucial step to achieve scalability.

## 6.1 Data distribution

The parallel version of the matrix-vector-multiplication requires parallel (i. e. distributed) storage of the numerical data, i. e. every processor stores a certain part of the matrices and vectors, such that the product can be computed with minimal communication.

In order to avoid communication during the discretization step the parts of the matrices and vectors are stored on the same processor as the correponding part of the grid. Hence, the distribution of the numerical data is determined by the distribution of the geometrical data and the *load-balancing problem is transformed into a graph-partitioning problem.*

We chose a distribution of the tetrahedra without overlap (a tetrahedron is stored on one processor only), whereas faces, edges and vertices are stored with overlap (e. g. if a tetrahedron has a neighbour on a different processor the connecting face is stored on both processors). This is because for the discretization as well as for the refinement algorithm it is desirable that every tetrahedron has (fast) access to its faces, edges and vertices. The consistency of the distributed geometry data storage is guaranteed by DDD (cf. §6.3).

As a consequence, the numerical data are stored with overlap, too, since most finite element types have unknowns on faces, edges or vertices.

## 6.2 Load balancing

For optimal scalability it is important to distribute the workload equally among the processors. An unbalanced distribution not only increases the computation time but also wastes computer memory thus decreasing the maximal problem size that can be handled on a specific machine. Even though the initial grid may be distributed equally, load unbalance arises due to the adaptive refinement of the grid.

In case of load unbalance, an improved, more balanced, configuration has to be computed and the corresponding load migration has to be performed (the latter is done by DDD). This

kind of load balancing is called dynamic (in contrast to static load balancing, which is performed only once at the beginning).

The load balancing algorithm does not only have to yield for a uniform data distribution (which would be trivial), it also has to minimize the overlap (to minimize communication). In order to describe this mathematically, we define for a triangulation $\mathcal{G}$ the *dual graph* $G(\mathcal{G}) = (\mathcal{G}, F)$ with

$$(T_1, T_2) \in F \subset \mathcal{G} \times \mathcal{G} \quad :\Leftrightarrow \quad \text{the tetrahedra } T_1 \text{ and } T_2 \text{ share a common face.}$$

Note that this graph is undirected. In order to express the workload for each tetrahedron (i. e. vertex of the dual graph) and the amount of communication across each face (i. e. edge of the dual graph), a *weighted graph* $G = (\mathcal{G}, F, \alpha, \beta)$ with weight functions $\alpha : \mathcal{G} \to \mathbb{R}_+$ and $\beta : F \to \mathbb{R}_+$ is used. The weight of a subset $\tilde{\mathcal{G}} \subset \mathcal{G}$ is defined as

$$\alpha(\tilde{\mathcal{G}}) := \sum_{T \in \tilde{\mathcal{G}}} \alpha(T),$$

representing the cumulative workload for the corresponding tetrahedra. For a given partitioning $m : \mathcal{G} \to \{1, \ldots, P\}$ the set

$$\mathrm{cut}(m) := \{(T_1, T_2) \in F : m(T_1) \neq m(T_2)\}$$

corresponds to the faces, where the inter-processor communication occurs. The load balancing problem can now be written as a graph partitioning problem, where the parameter $C$ controls the level of imbalance:

**Definition 2 (Generalized graph partitioning problem)** *Given $P \in \mathbb{N}$, $C \in \mathbb{R}$, $C \geq 1$, and an undirected weighted graph $G = (\mathcal{G}, F, \alpha, \beta)$, find a mapping $m : \mathcal{G} \to \{1, \ldots, P\}$ such that*

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

*subject to*

$$\alpha(\mathcal{G}_p) \leq C \frac{\alpha(\mathcal{G})}{P} \qquad \forall\, 1 \leq p \leq P$$

*with $\mathcal{G}_p := m^{-1}(p) = \{T \in \mathcal{G} : m(T) = p\}$.*

As the graph partitioning problem is NP complete, an optimal solution of the load balancing problem is out of reach. However, there exist several heuristic approaches that produce reasonably good results. Besides methods from discrete optimization like *simulated annealing* (SA), geometry based algorithms like *recursive coordinate bisection* (RCB) or *recursive inertial bisection* and graph based algorithms like the *recursive spectral bisection* (RSB) or the *Kernighan-Lin algorithm* (KL) are widely used. A new approach is the application of the multilevel idea, but now in the field of graph partitioning. In this approach the initial graph is coarsened several times resulting in a hierarchy of graphs, where only the coarsest graph is partitioned by means of one of the foregoing methods. An overview of several graph partitioning algorithms can be found in [13].

As load balancing has to be performed when the mesh is already distributed among the processors, we need *parallel* graph partitioning method. Therefore, we use the parallel graph partitioning library *ParMetis* [29]. It offers both graph-oriented and geometry-oriented algorithms for partitioning, also including a (graph-)multilevel version of the KL-algorithm.

### 6.3 DDD — dynamic distributed data

The DDD library ([7, 8]) was originally designed for the software package UG to parallelize the serial code. DDD performs all communication via its portable parallel interface layer (PPIF) and therefore is independent from the underlying parallel programming model. Hence, DDD can be used on almost all platforms that offer a message passing model (not necessarily MPI). DDD includes a wide range of functionality and can be easily adapted to a special application by means of user-defined handler functions.

DDD offers an abstract interface for the parallel management of distributed data. The data are assumed to be organized as a graph $(V, E)$, where $V$ consists of the distributed objects and an edge $(o_1, o_2) \in E \subset V \times V$ means, that object $o_1$ references object $o_2$, in our case via a pointer from $o_1$ to $o_2$. Because of the distributed memory model there is no global address space, only different local address spaces. Due to this fact, the global data graph divides into several local graphs. As references across processors are not admissible the coupling of the local graphs to the distributed graph has to be established by means of the distributed objects. The coupling is established and held consistent automatically by four program modules of DDD: the management module, the transfer module, the interface module and the identify module.

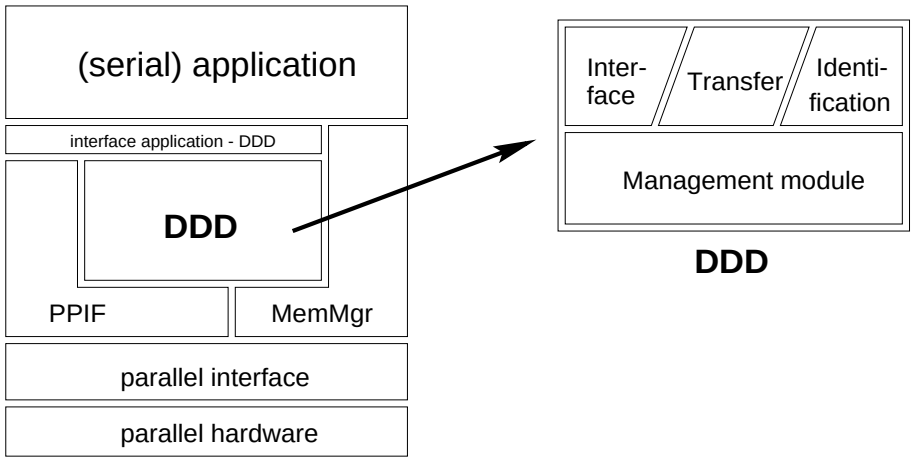


Figure 7: The structure of DDD (according to [7])

**Management module.** All data types that are assigned for distributed storage have to be registered with DDD as a DDD data type: Every DDD object contains a `DDD_HEADER` record, that stores the global ID, the DDD type, the two properties "priority" and "attribute", and some internal data about the coupling lists. The registration is done at runtime by telling DDD the location of the `DDD_HEADER` and all other members inside the object as well as specifying their validity (local or global): Global data has the same value for all copies of a distributed object, whereas the local data may differ on each processor. Once a DDD type is registered, distributed objects of this type can be constructed and transferred from one processor to another.

**Transfer module.** DDD object copies can be deleted or copied to other processors by means of the transfer functions `DDD_XferDeleteObj()` and `DDD_XferCopyObj()`. All calls to transfer functions are enclosed by `DDD_XferBegin()` and `DDD_XferEnd()`, such that all communication between two processors can be bundled. The behaviour of the objects during the transfer can be controlled by several standard or user-defined handler functions.

**Interface module.** Interfaces are sets of distributed objects of a certain type on the processor boundaries, for which synchronizing communication can be performed, e. g. for the accumulation of distributed vectors (cf. §6.4). The interface is defined by calling `DDD_IFDefine()`, the communication is determined by gather/scatter handlers, that also have to be defined by

the user. Every interface is adjusted automatically whenever topological changes of the data graph (e. g. by the transfer module) occur.

**Identify module.** The identify module is used, when objects on different processors are combined to one distributed object. This identification process utilizes identification tuples, that may consist of integers, strings or distributed objects. Identification is done between enclosing calls to `DDD_IdentifyBegin()` and `DDD_IdentifyEnd()`, thus allowing bundled communication as in the transfer process.

## 6.4 Parallelization of matrix and vector operations

This section describes the parallelization of the matrix-vector product and the inner product, the main operations of a CG iteration and at the same time the only operations that have to be parallelized to obtain a parallel CG algorithm.

There exist two types of parallel vectors: accumulated and distributed vectors. They differ in the way the data is stored in the overlapping parts. Accumulated vectors store the correct value in each copy of an entry, whereas distributed vectors only store some share of the value, the sum of all local shares yields the correct value.

The definition of the both vector types below is from [20]. In order to define them properly, we have to introduce coincidence matrices, first:

**Definition 3 (Coincidence matrix)** *Let $n$ be the total number of unknowns and $n_p$ the number of unknowns on processor $p$. The coincidence matrix $I_p \in \mathbb{R}^{n_p \times n}$ is defined as*

$$(I_p)_{i,j} := \begin{cases} 1 & \textit{unknown with global number } j \textit{ exists on process } p \\ & \textit{with local number } i \\ 0 & \textit{else} \end{cases}$$

**Definition 4 (Accumulated vector)** *$\overline{\mathbf{x}} \in \mathbb{R}^n$ is stored accumulated on processor $p$ as $\overline{\mathbf{x}}_p$, iff*

$$\overline{\mathbf{x}}_p = I_p \overline{\mathbf{x}}.$$

**Definition 5 (Distributed vector, distributed matrix)** *$\mathbf{x} \in \mathbb{R}^n$ is stored distributed on processor $p$ as $\mathbf{x}_p$, iff*

$$\mathbf{x} = \sum_{p=1}^{P} I_p^T \mathbf{x}_p,$$

*i. e. the overlap unknowns store only a fraction of the value, the sum of the values corresponding to an overlap unknown yields the global value. The same holds for a distributed matrix $\mathbf{M}$, whose part $\mathbf{M}_p$ is stored distributed on processor $p$:*

$$\mathbf{M} = \sum_{p=1}^{P} I_p^T \mathbf{M}_p I_p.$$

Note that if $\mathbf{M}$ is a stiffness matrix, then $\mathbf{M}_p$ is equal to the stiffness matrix corresponding to the subtriangulation $\mathcal{G}_p$ of processor $p$. This facilitates parallel discretization, since no communication is needed when assembling the stiffness matrix using standard finite elements.

Clearly, basic vector operations like addition and multiplication with a scalar can be done without communication if the parallel vectors are of the same type. The conversion from a distributed to an accumulated vector requires communication:

$$\overline{\mathbf{x}}_p = I_p \sum_{i=1}^{P} I_i^T \mathbf{x}_i.$$

In the opposite direction, the conversion is not unique. One way is to store the value in one unknown, whereas all copies store the value 0. Another way would be to distribute the value uniformly.

The easiest way to compute the inner product of two vectors in parallel is to use vectors of different type, in which case all computations can be performed locally. Only the last step, the accumulation of the partial inner products requires communication:

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

Thus the inner products are computed locally and then the local results are summed up via `MPI_Reduce()` or `MPI_Allreduce()`. If the vectors are of the same type, a type conversion (which requires additional communication) has to be done first.

Now let us consider the parallel matrix-vector product:

$$\mathbf{M} \cdot \overline{\mathbf{x}} = \sum_{p=1}^{P} I_p^T \mathbf{M}_p I_p \cdot \overline{\mathbf{x}} = \sum_{p=1}^{P} I_p^T \underbrace{\mathbf{M}_p \overline{\mathbf{x}}_p}_{:=\mathbf{y}_p} = \mathbf{y}$$

As one can see, the product of a distributed matrix and an accumulated vector yields a distributed vector. The products are computed locally, so no communication has to be done. Only the product of a distributed matrix with a distributed vector requires prior type conversions, i. e. communication.

The accumulation of distributed vectors for the purpose of type conversion is performed in DROPS using DDD interfaces. According to the finite element type, i. e. where the unknowns are placed, interfaces for vertices, edges or faces are required. The implementation in principle only requires the definition of a gather/scatter handler, in particular no parallel programming has to be done.

The pair of gather/scatter handler functions determines the action of the interface communication. The gather handler on the sending processor collects the appropriate values of the overlap unknowns and stores them in buffers, which are sent to other processors across the interface during the communication. Then on the receiving processor, the scatter handler adds the buffer values to the corresponding values of the vector.

### 6.5 A first parallel numerical experiment

We tested the parallel preliminary version of DROPS for a simple model problem. We consider the Poisson equation on the unit cube $\Omega = [0,1]^3$ with homogeneous Dirichlet boundary condition on the face $\Sigma_1 = [0,1]^2 \times \{0\}$ and homogeneous Neumann boundary conditions on the other faces:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= 0 && \text{on } \Sigma_1 \\ \frac{\partial u}{\partial n} &= 0 && \text{on } \partial\Omega \backslash \Sigma_1 \ . \end{aligned}$$

The triangulation of the unit cube consists of $21 \times 21 \times 21$ resp. $32 \times 32 \times 32$ subcubes, each subdivided into 6 tetrahedra. Table 1 and 2 show the times in seconds used for the numerical solution of this problem on a small PC cluster for different numbers of processors. $T_{run}$ is the time used for the discretization method and the iterative solver. $T_{CG}$ is the time for solving (CG solver with stopping criterion res $< 10^{-10}$, initial vector $x_0 := 0$).

| $P$ | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| $m_P$ | 10164 | 5569 | 2935 | 1571 |
| $T_{run}(P)$ | 9.7 | 5.5 | 3.0 | 2.9 |
| $T_{CG}(P)$ | 5.0 | 3.2 | 1.9 | 2.3 |
| $E_{run}(P)$ | — | 88.9% | 81.8% | 42.3% |
| $E_{CG}(P)$ | — | 78.0% | 68.1% | 27.3% |

Table 1: runtime for $n = 10164$ unknowns (176 iterations)

| $P$ | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| $m_P$ | 34848 | 18138 | 9682 | 4988 |
| $T_{run}(P)$ | 43.2 | 21.8 | 11.7 | 8.6 |
| $T_{CG}(P)$ | 26.8 | 13.7 | 7.8 | 6.6 |
| $E_{run}(P)$ | — | 99.0% | 92.1% | 62.5% |
| $E_{CG}(P)$ | — | 97.7% | 86.0% | 50.4% |

Table 2: runtime for $n = 34848$ unknowns (248 iterations)

The efficiency is of course better for the second example, as it has a significantly larger problem size. For this example we have scalability up to $P = 4$ processors.

The poor efficiency for $P = 8$ processors can be explained by the small local problem size, because then the communication costs are rather big compared to the computational costs. This leads to results like $T_{CG}(4) < T_{CG}(8)$ in Table 1, where the communication costs seem to exceed the computational costs.

The loss of efficiency with increasing $P$ is not surprising if one considers the increase of the communication cost. The inner processors have an interface size

$$s_p \approx 6n_p^{2/3}.$$

Since the communication costs are proportional to $s_p$ and the arithmetic costs are proportional to $n_p$, the quotient of communication costs to arithmetic costs is proportional to $n_p^{-\frac{1}{3}}$. This quotient increases with growing $P$ and thus the efficiency $E(P)$ can be expected to decrease. Only if the problem size is big enough compared to the number of processors $P$, the question for scalability is reasonable.

Unfortunately the extension of the problem size by the factor $P$ for a run on $P$ processors is not possible yet because of the preliminary parallel version's restrictions: The grid is generated on a single processor, from where it is distributed onto the other processors. This makeshift is only used for the reason that the parallelization of the refinement algorithm has not been finished yet. Not only the local memory size but also the huge time expense for the transfer of $\frac{P-1}{P}$ % of the tetrahedra limit the experiment to fairly low problem sizes $n$. The future parallel version of DROPS (with parallel refinement algorithm) will have to demonstrate its scalability for $P \geq 8$ by means of proportionally bigger problems, which will be possible soon.

# 7 Numerical results

In this section we present results of a few numerical experiments with the DROPS package. These experiments are conducted under SuSe-Linux 7.2 on one processor of a dual Pentium-III computer with 600 MHz and 1 GB RAM. DROPS is compiled with a recent snapshot of GCC-3.1 (-W -Wall -pedantic -O2 -funroll-loops -march=pentium3 -fomit-frame-pointer -finline-limit=2000). The system is in multi-user-mode, but calm apart from the usual daemons. Each

experiment is conducted twice and only the arithmetic mean of times in both runs is quoted in subsequent tables. All times are given in seconds if not stated otherwise.

## 7.1 Poisson equation

As a first test problem we take the Poisson equation on the unit cube $\Omega = [-1, 1]^3$ with Dirichlet boundary conditions. The right hand side $f$ and the boundary conditions are taken such that the continuous solution is given by

$$u(x) = \frac{1}{1 + e^{-60(r(x)-0.3)}}, \qquad \text{with} \qquad r(x) = \sqrt{x_1^2 + x_2^2 + x_3^2}.$$

$u$ is a slowly varying function with a very rapid change in a small neighbourhood of a sphere with radius 0.3 centered at the origin. For the initial triangulation we first partition $\Omega$ regularly into $4 \times 4 \times 4$ cubes and then each of these cubes is subdivided into six tetrahedra. One regular refinement step is performed yielding 3072 tetrahedra on the starting level 0. We use linear finite elements for discretization.

### 7.1.1 Poisson equation on a uniform triangulation

Here, we consider a case with only global regular refinement, where we compare a SSOR-preconditioned CG iterative solver with a multigrid method. For the later, we use a V-cycle with two applications ($\nu = 2$) of a symmetric Gauß-Seidel smoother for pre- and post-smoothing. Table 3 contains statistics on the geometric data-structures. For each loop as in Figure 1 the triangulation level and the number of unknowns and tetrahedra are listed. Also, the total (i. e., on the present and all coarser levels) numbers of tetrahedra ($T_{\text{tot}}$), faces ($F_{\text{tot}}$), edges ($E_{\text{tot}}$) and vertices ($V_{\text{tot}}$) are given.

| Loop | Level | #Unk. | $\#T_{\text{tri}}$ | $\#T_{\text{tot}}$ | $\#F_{\text{tot}}$ | $\#E_{\text{tot}}$ | $\#V_{\text{tot}}$ | $h_{\max}$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 3375 | 24576 | 28032 | 58080 | 35812 | 4913 | 0.2165 |
| 1 | 2 | 29791 | 196608 | 224640 | 457440 | 274500 | 35937 | 0.1083 |
| 2 | 3 | 250047 | 1572864 | 1797504 | 3627744 | 2146564 | 274625 | 0.05413 |

Table 3: Uniform triangulations

The number of simplices is roughly proportional to $h_{max}^{-3}$, as is expected for tetrahedral triangulations. This behaviour leads to a very strong increase in memory requirements if the grid is refined. The storage needed for the multigrid solver on the finest triangulation (level 3) is circa 690 MB, so only 70 percent of the main memory is utilized, but the next finer uniform triangulation would lead to a memory footprint of several Gigabytes, which exceeds the capacity of our computer by far.

Note that the use of higher-order finite elements (for example, $P_2$) would probably yield a higher attainable accuracy given the storage capacity.

In Tables 4 and 5 some characteristics of the multigrid and PCG solver are presented. The total time needed for the solution process, the time for setting up the discrete equation and the amount of time spent in the iterative solver are listed in the first columns. The fifth column contains the number of iterations needed to obtain a discrete residual less than $10^{-8}$. In the sixth column the average reduction of the residual per iteration is shown. In the last column the $L_2(\Omega)$ norm of the discretization error is given.

These results are consistent with the properties of the multigrid and PCG method discussed in §4.1. The rate of convergence of the multigrid method is high and independent of $h_{max}$,

| Loop | $t_{\text{cum}}$ | $t_{\text{disc}}$ | $t_{\text{sol}}$ | # it. | Av. $\text{Red}_{\text{res}}$ | $\|u - u_h\|_{L_2}$ |
|---|---|---|---|---|---|---|
| 0 | 0.91 | 0.81 | 0.10 | 10 | 0.121 | 0.369 |
| 1 | 8.78 | 7.20 | 1.58 | 11 | 0.157 | 0.0342 |
| 2 | 73.4 | 59.9 | 13.5 | 11 | 0.159 | 0.0117 |

Table 4: Multigrid-solver on uniform triangulations

| Loop | $t_{\text{cum}}$ | $t_{\text{disc}}$ | $t_{\text{sol}}$ | # it. | Av. $\text{Red}_{\text{res}}$ | $\|u - u_h\|_{L_2}$ |
|---|---|---|---|---|---|---|
| 0 | 0.85 | 0.75 | 0.10 | 25 | 0.438 | 0.369 |
| 1 | 9.3 | 6.5 | 3.00 | 47 | 0.644 | 0.0342 |
| 2 | 103.5 | 54.8 | 48.7 | 87 | 0.794 | 0.0117 |

Table 5: PCG-solver on uniform triangulations

whereas for the CG method we observe a $h^{-1}$ effect in the number of iterations. Due to the prolongation matrices and linear systems on coarser grids the discretization phase of the multigrid algorithm takes longer than that of the PCG algorithm, which leads to a (slightly) higher efficiency of the PCG algorithm for small problems. With increasing problem size the multigrid method becomes (much) faster than the PCG method.

In Figure 8 a plot of the discrete solution $u_h(x_1, x_2, 0)$ on the level 3 triangulation is shown.

### 7.1.2 Poisson equation with adaptive refinement

In this experiment for the Poisson equation the residual error estimator from section 5.1 is used in combination with the largest error first marking strategy. This introduces a new parameter, namely $err_{\text{rel}}$ of §5.3. We use the multigrid method as iterative solver.

In Table 6 the dependence of the total solution time, the number of unknowns and the number of tetrahedra on $err_{\text{rel}}$ is shown. Column two contains the number of iterations of the adaptive algorithm depicted in Figure 1 before the stopping criterion is met. The adaptive solution process is stopped as soon as the discretization error $\|u - u_h\|_{L_2}$ is less than 0.0117. This tolerance is chosen in order to compare the results with the results in Table 4 (uniform refinement).

| $err_{\text{rel}}$ | Loop it. | $t_{\text{cum}}$ | $\|u - u_h\|_{L_2}$ | # Unk. | $\#T_{\text{tot}}$ |
|---|---|---|---|---|---|
| 0.2 | 57 | 75.0 | 0.0116 | 9554 | 71165 |
| 0.3 | 29 | 40.1 | 0.0115 | 10246 | 76131 |
| 0.4 | 20 | 51.1 | 0.0078 | 23581 | 175105 |
| 0.6 | 10 | 25.5 | 0.0112 | 19937 | 147258 |
| 0.8 | 7 | 28.0 | 0.0070 | 31530 | 227609 |

Table 6: Results for adaptive refinement

Note that the adaptive algorithm is faster than the uniform one in all cases except for $err_{\text{rel}} = 0.2$. For small values of $err_{\text{rel}}$ the adaptive algorithm is rather slow, but due to the relatively small number of tetrahedra that are refined in each loop iteration the stopping criterion can be met very precisely. With $err_{\text{rel}} = 0.2$ the number of unknowns and of tetrahedra needed is approximately a factor 25 smaller as compared to uniform refinement.

In the following experiment we take $err_{\text{rel}} = 0.8$ and a residual tolerance of $10^{-8}$ for the multigrid iterative solver.

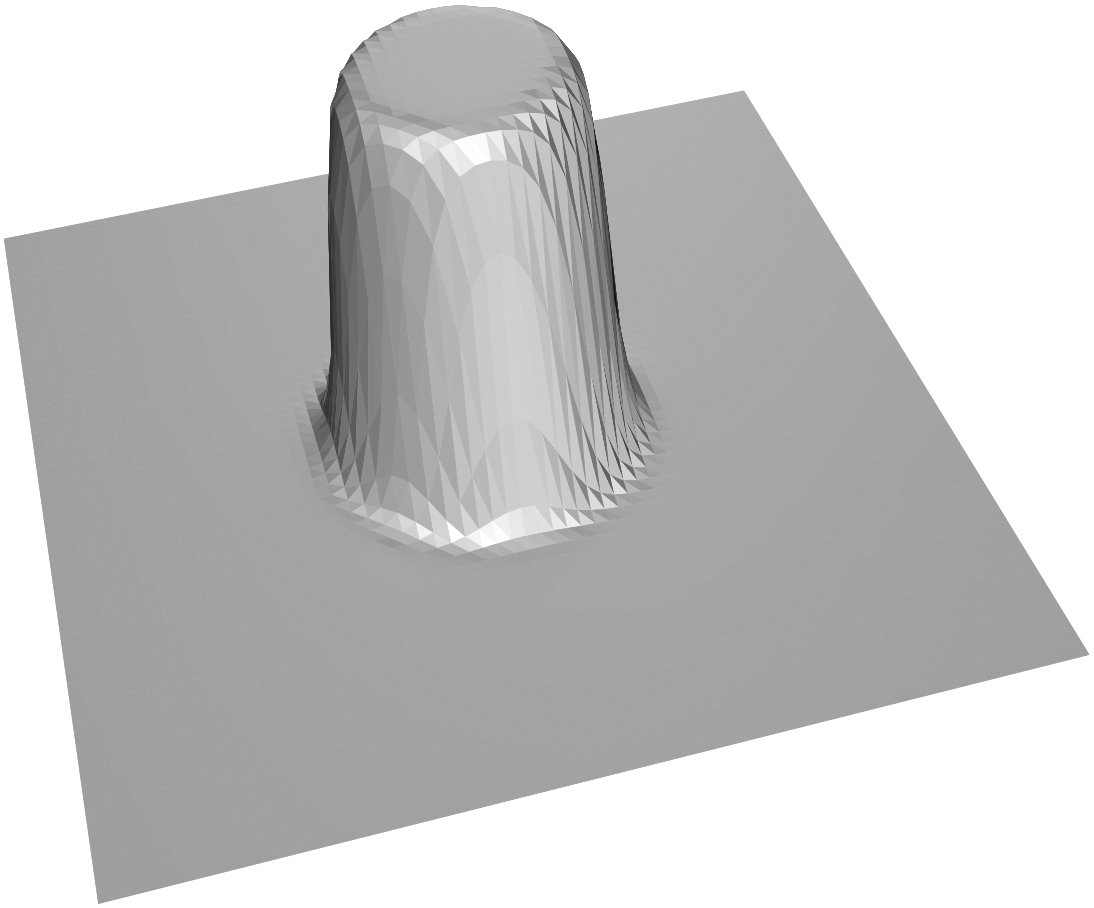

Figure 8: Solution on uniform triangulation with $1.6\,10^6$ tetrahedra; $x_3 = 0$ plane

In Table 7 results for the adaptive refinement method are given. Between two iterations of the algorithm in Figure 1 the number of unknowns is more than doubled, so that one can expect the multigrid solver to work with optimal complexity. Indeed, the time spent in the solver scales linearly with the number of unknowns as indicated by Tables 7 and 8. One can observe that the average reduction of the residual is better than for the uniform case.

Regarding memory efficiency, the adaptive algorithm is clearly superior to the one with uniform refinement. Although $err_{\rm rel}$ is rather high, the former only needs 31530 unknowns and about 2.3 $10^5$ tetrahedra to reduce the $L_2(\Omega)$-error to below 0.1166. The method with uniform refinement achieves this with 250047 unknowns and about 1.6 $10^6$ tetrahedra.

These savings come at the price of the error estimator though. Table 8 shows that the error estimator is slightly cheaper than the discretization method, which is the most time consuming component in this implementation. The grid-manager and solver are notably faster than the other two components. The 7 steps in this experiment take 28 seconds to complete, which is 45 seconds less than the uniform algorithm needs.

Figure 10, which is based on Table 7 illustrates the quality of the global error estimate. One can see that the ratio of estimated to true global error lies around 30. On coarse triangulations the effect of the perturbation terms in Theorem 2 and the use of quadrature formulas for calculating the 'exact' error becomes visible.

Finally, a test that is tuned to use $1.6\,10^6$ tetrahedra as in the finest uniform triangulation is performed. The result is shown in Figure 11. Here the $L_2(\Omega)$-error is 0.00200, a factor 6 less than for the uniform triangulation.

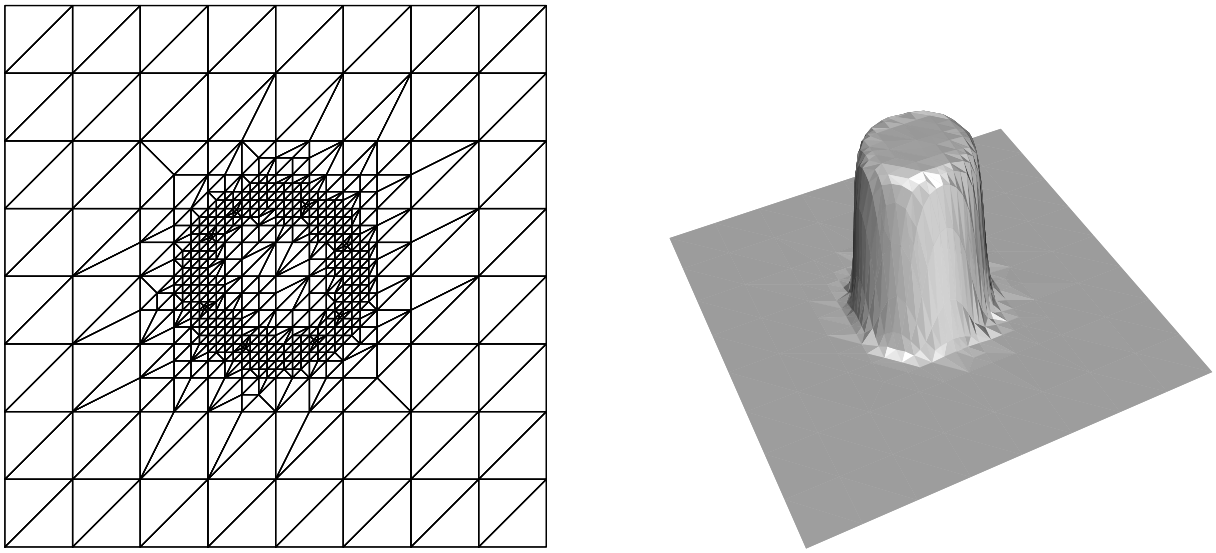

Figure 9: Adaptive triangulation for the Poisson equation, $err_{\mathrm{rel}} = 0.2$; 71000 tetrahedra; $x_3 = 0$ plane

| Loop | # Unk. | $\#T_{\mathrm{tot}}$ | Av. $\mathrm{Red}_{\mathrm{res}}$ | $\|u - u_h\|_{L_2}$. |
|---|---|---|---|---|
| 0 | 343 | 3456 | 1e-9 | 0.268 |
| 1 | 426 | 4100 | 0.01411 | 0.425 |
| 2 | 633 | 5664 | 0.01498 | 0.218 |
| 3 | 1345 | 10866 | 0.06519 | 0.0651 |
| 4 | 3840 | 29181 | 0.06952 | 0.0290 |
| 5 | 7807 | 56749 | 0.1154 | 0.0136 |
| 6 | 31530 | 227609 | 0.1163 | 0.00698 |

Table 7: Details of adaptive refinement for $err_{\mathrm{rel}} = 0.8$

## 7.2 Stokes equation

In the following, we consider the stationary Stokes equation (3) with $a_0 = 0$. The weak formulation is given in (11). The problem is discretized using $P_2$-$P_1$ finite elements. For the iterative solution of the discrete problem we apply the inexact Uzawa method described in §4.2. We first perform a numerical experiment with uniform refinement to test the reliability of the error estimator. Then we show results of a driven cavity experiment, in which adaptivity is used.

### 7.2.1 Stokes equation with regular refinement

To illustrate the reliability of the residual error estimator for the stationary Stokes problem, we solve a model Stokes problem as in (3) with $a_0 = 0$ on the domain $\Omega = [0, \pi/4]^3$. We take

$$\begin{aligned} f(x,y,z) &= \begin{pmatrix} 0 \\ 0 \\ 3 \cdot \cos(x)\sin(y)\cos(z) \end{pmatrix} \\ u(x,y,z) &= \frac{1}{3}\begin{pmatrix} \sin(x)\sin(y)\sin(z) \\ -\cos(x)\cos(y)\sin(z) \\ 2 \cdot \cos(x)\sin(y)\cos(z) \end{pmatrix} \\ p(x,y,z) &= \cos(x)\sin(y)\sin(z) + C \end{aligned}$$

| Loop | Level | $t_{\text{cum}}$ | $t_{\text{ref}}$ | $t_{\text{disc}}$ | $t_{\text{sol}}$ | $t_{\text{est}}$ |
|---|---|---|---|---|---|---|
| 0 | 1 | 0.20 | 0.01 | 0.08 | 0.01 | 0.10 |
| 1 | 2 | 0.27 | 0.01 | 0.13 | 0.01 | 0.12 |
| 2 | 3 | 0.44 | 0.02 | 0.23 | 0.02 | 0.17 |
| 3 | 3 | 0.84 | 0.07 | 0.40 | 0.05 | 0.32 |
| 4 | 4 | 2.39 | 0.19 | 1.16 | 0.19 | 0.85 |
| 5 | 4 | 4.74 | 0.43 | 2.15 | 0.45 | 1.71 |
| 6 | 5 | 19.2 | 1.56 | 8.62 | 2.11 | 6.87 |

Table 8: Time statistics for the adaptive refinement; $err_{\text{rel}} = 0.8$

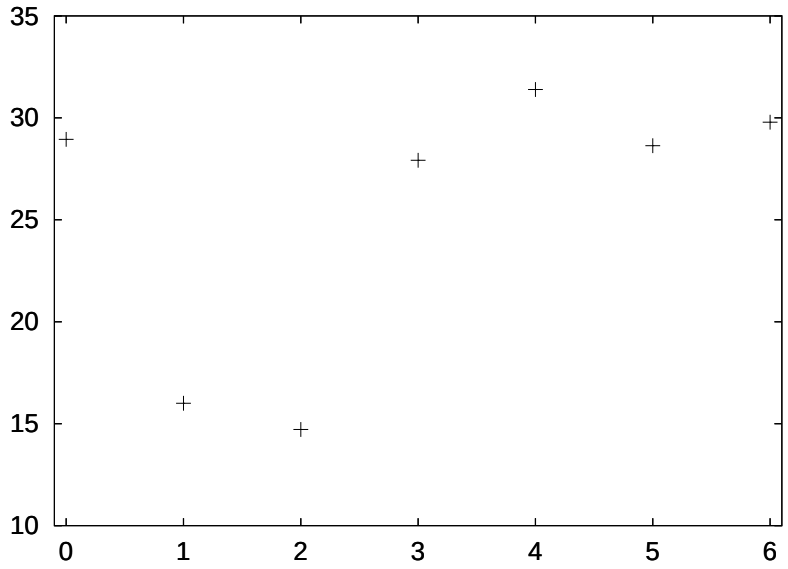


Figure 10: Ratio of estimated to true $L_2(\Omega)$-error for each loop iteration.

with a constant $C$ such that the integral of $p$ over $\Omega$ vanishes. The Dirichlet boundary conditions are obtained by restricting this solution $u$ to the boundary.

Initially, the domain is partitioned into 162 tetrahedra and refined regularly for each subsequent computation. Table 9 contains the level ($\overline{\ell}$), the total number of tetrahedra ($T_{\text{tot}}$), the estimated global error in the norm $\|\cdot\|_X$ introduced in §5.2 and the discretization error in this norm. The last column shows the ratio of the estimated error to the true error.

As theory for $P_2$-$P_1$ finite elements predicts ([30]), the true error scales like $h^{-2}$ for regular refinement. The results in the last column demonstrate the reliability of the residual error estimator.

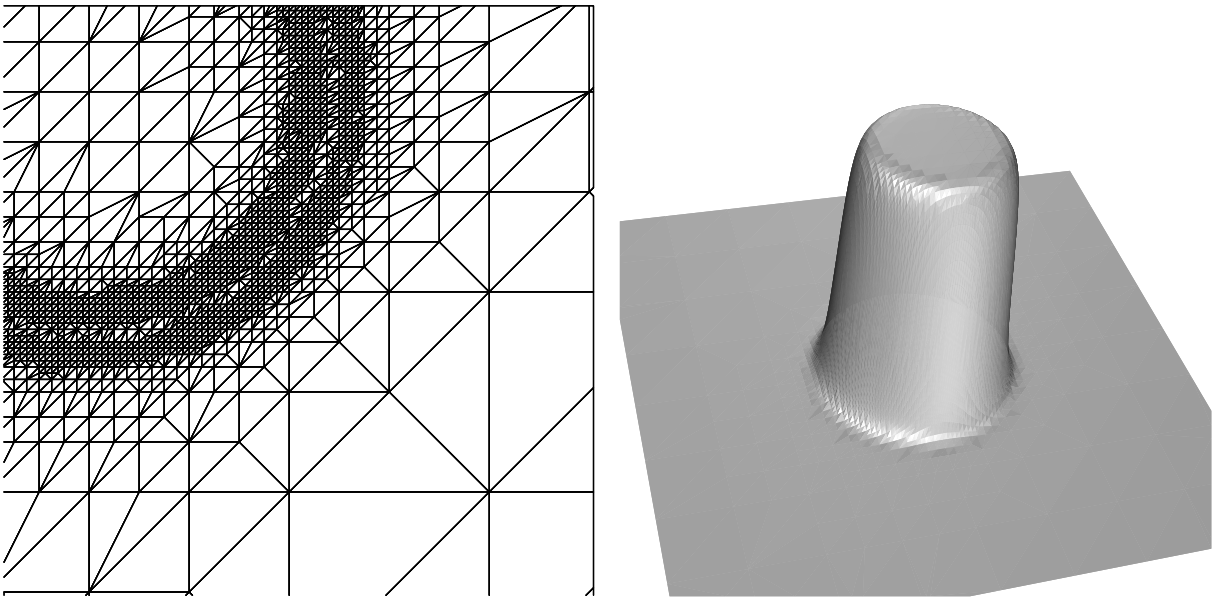

Figure 11: Grid and solution for adaptive refinement with $1.6 \cdot 10^6$ tetrahedra; $x_3 = 0$ plane

| $\bar{\ell}$ | $\#T_{\text{tot}}$ | $\|\cdot\|_{\text{X}}$-error estimate | $\|\cdot\|_{\text{X}}$-error | ratio |
|---|---|---|---|---|
| 0 | 162 | 0.0410 | 0.00482 | 8.51 |
| 1 | 1458 | 0.00995 | 0.00116 | 8.58 |
| 2 | 11826 | 0.00248 | 2.89e-4 | 8.58 |
| 3 | 94770 | 6.19e-4 | 7.22e-5 | 8.57 |

Table 9: Global error estimation for the Stokes problem

### 7.3 Driven cavity problem with adaptive refinement

We consider the popular *"driven cavity"* test problem. Let $\Omega = [0,1]^3$ be the unit cube and $\Sigma_0 = [0,1]^2 \times \{1\}$ be one of its faces. The continuous problem is given by

$$\begin{aligned} -\Delta u + \nabla p &= f \quad \text{in } \Omega \\ \operatorname{div} u &= 0 \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \setminus \Sigma_0 \\ u &= \phi \quad \text{on } \Sigma_0 \ , \end{aligned} \tag{31}$$

where

$$\phi(x,y,z) = \begin{cases} (1,0,0)^T & \text{for } 0.1 \le x, y \le 0.9 \\ \frac{0.5 - \max(|x-0.5|, |y-0.5|)}{0.1} (1,0,0)^T & \text{else.} \end{cases}$$

Instead of $u = (1,0,0)^T$ on the whole face $\Sigma_0$ the smoothed boundary values $u = \phi$ are chosen to obtain proper Dirichlet boundary conditions. Otherwise, in the presence of an adaptive error estimation technique, one would discover that the driven cavity problem with the former boundary condition is ill-posed, as jumping boundary conditions lead to infinite pressure in the corners of the in- and outflow edge of the domain.

The initial triangulation $\mathcal{G}_0$ consists of a $2 \times 2 \times 2$ grid of sub-cubes that are triangulated by inserting a diagonal. To control the adaptive refinement the default (largest error first) marking strategy of §5.3 is employed with $err_{\text{rel}} = 0.8$ and the residual error estimator of §5.2.

In the inexact Uzawa method for the preconditioners $\tilde{\mathbf{A}}^{-1}$ and $\tilde{\mathbf{S}}^{-1}$ we use $k_{\text{iter}}$ iterations of an SSOR-preconditioned CG solver applied to the systems (20), (21). In the near future the PCG solver will be replaced by the multigrid solver. At the moment this is not possible because the prolongation for $P_2$ elements is not implemented yet.

The Uzawa iteration is stopped, if $\|\mathbf{r}\|_2 < 10^{-8}$ holds, where $\mathbf{r} = \begin{pmatrix} \mathbf{A}\mathbf{x} + \mathbf{B}^T\mathbf{y} - \mathbf{b} \\ \mathbf{B}\mathbf{x} - \mathbf{c} \end{pmatrix}$ is the residual vector. The number of Uzawa iterations needed for this stopping criterion depends on $k_{\text{iter}}$. Table 10 shows the number of Uzawa iterations (# iter) and the computation times for different values of $k_{\text{iter}}$. The results are given for the triangulation $\mathcal{G}_6$ in loop iteration 7 ($n_v = 33129$ velocity unknowns, $n_p = 2637$ pressure unknowns).

| $k_{\text{iter}}$ | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| # iter | DIV | 231 | 158 | 156 |
| $t_{\text{sol}}$ | — | 106 | 89 | 106 |

Table 10: Inexact Uzawa method: inner-outer iteration

For $k_{\text{iter}} = 2$ the method does not converge because of poor preconditioning. For $k_{\text{iter}} = 4$ an optimal computation time is achieved. We now set $k_{\text{iter}} = 4$ and apply adaptive refinement for nine iterations of the algorithm in Figure 1. Results are shown in Table 11.

| Loop | Level | $t_{disc}$ | $t_{sol}$ | $t_{est}$ | # Uzawa It. | $n_v$ | $n_p$ | $\|\cdot\|_X$ error-est. |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0.01 | 0.16 | 0.01 | 463 | 81 | 27 | 51.77 |
| 1 | 1 | 0.01 | 0.46 | 0.03 | 360 | 243 | 48 | 48.57 |
| 2 | 2 | 0.04 | 1.04 | 0.06 | 348 | 546 | 86 | 32.84 |
| 3 | 3 | 0.11 | 3.55 | 0.14 | 314 | 1335 | 168 | 25.11 |
| 4 | 3 | 0.19 | 6.57 | 0.22 | 271 | 2247 | 246 | 17.09 |
| 5 | 4 | 0.50 | 17.9 | 0.53 | 228 | 5430 | 535 | 10.11 |
| 6 | 5 | 1.60 | 52.0 | 1.49 | 193 | 16335 | 1338 | 6.131 |
| 7 | 6 | 3.35 | 89.5 | 3.02 | 158 | 33129 | 2637 | 3.782 |
| 8 | 7 | 8.14 | 229 | 7.03 | 162 | 78903 | 5823 | 2.546 |
| 9 | 8 | 18.5 | 1840 | 14.6 | 548 | 170760 | 11076 | 1.562 |

Table 11: Driven cavity with adaptive refinement

Note that the iteration numbers in the inexact Uzawa method are quite high and that the iterative solver is by far the most time consuming component. We mention two possibilities for a significant improvement. In the first place, a nested iteration technique should be used. This means that for solving the discrete problem in iteration $i+1$ one can use the already computed discrete solution of iteration $i$ of the adaptive algorithm as a starting vector. Here, we use this approach only for the pressure components, as we cannot prolongate the $P_2$ velocity components yet. For these, we simply use the zero vector as starting vector for the Uzawa method. Secondly, the inexact Uzawa method can be accelerated by a CG method as explained in Remark 6.

Figures 12 and 14 show the adaptively generated grid, the pressure contours and the velocity field in the $x_2 = 0.5$ plane. Note, that Figure 12 only shows the upper right quadrant of the whole grid in the $x_2 = 0.5$ plane.

# 8 Outlook

The basic components which have been implemented form the skeleton of the DROPS parallel adaptive multigrid code. We summarize the main methods that are available in DROPS:

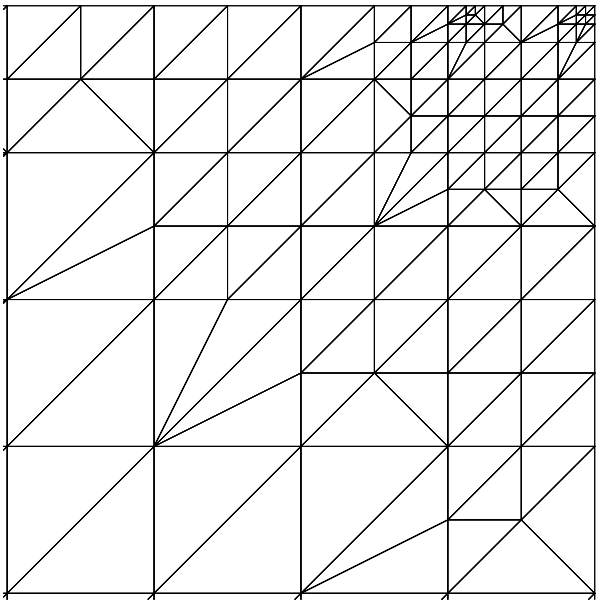

Figure 12: Driven Cavity: Triangulation in $x_2 = 0.5$ plane

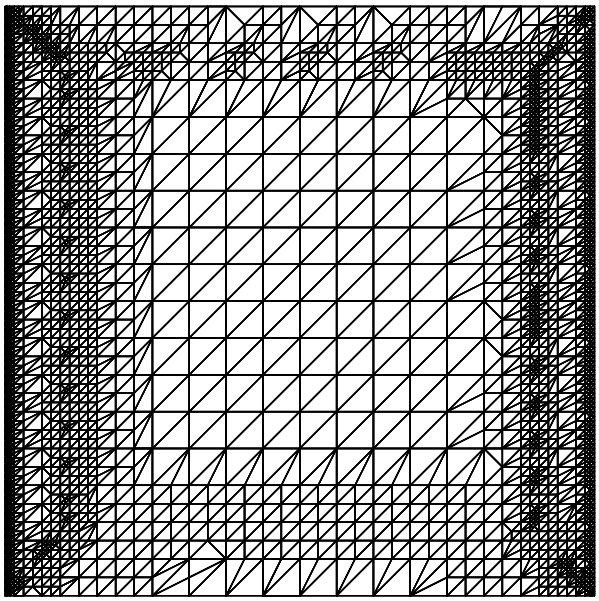

Figure 13: Driven Cavity: Triangulation in $x_3 = 1$ plane

**1.** A grid generation method which can construct a hierarchy of consistent and stable tetrahedral triangulations.

**2.** A conforming finite element discretization method for discretizing stationary elliptic problems (Poisson and (Navier-)Stokes).

**3.** A linearization method for the discrete Navier-Stokes equation.

**4.** The fractional-step method for time discretization of instationary (Navier-)Stokes equations.

**5.** The multigrid and SSOR-preconditioned CG iterative methods for solving discrete scalar problems (Poisson and convection-diffusion equations). The inexact Uzawa iterative method for solving discrete Stokes and Oseen problems.

**6.** A posteriori residual error estimators for the Poisson and stationary Stokes equation and marking strategies for grid refinement.

A parallel version of DROPS is currently developed. Parallel modules for the construction of the finite element discretization (assembling of the stiffness matrix) and for the sparse matrix-vector multiplication are already available. The implementation of a parallel version of the grid refinement method is almost finished.

Although the skeleton of the DROPS package is available now, the code is still far from a general purpose robust and efficient tool for CFD simulations. Below we briefly discuss some important issues concerning the further development of DROPS in the near future. We plan to address the following topics (not in chronological ordering):

- We will implement level set methods ([28, 37, 44]) for the treatment of two-phase incompressible flows with sharp interfaces. We intend to start with a variant of this level set technique (phase-field method) which has been successfully applied in the interdisciplinary research project ([25]).

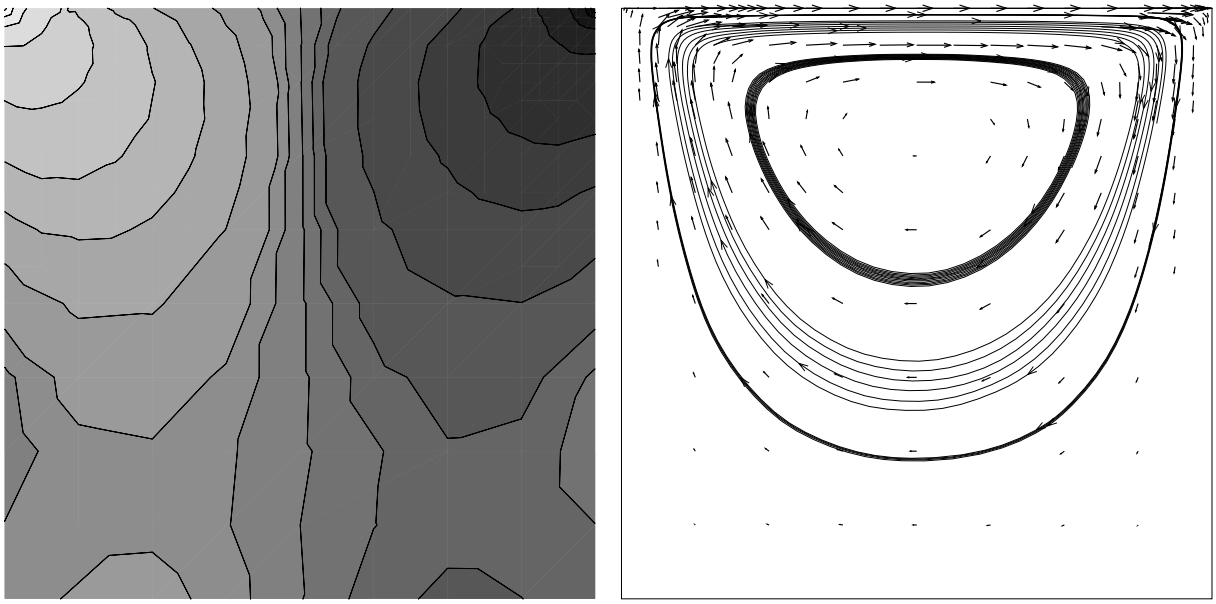

Figure 14: Driven Cavity: pressure contour lines and velocity field; $x_2 = 0.5$ plane

- As an application we consider a two-phase incompressible fluid flow consisting of a (small) drop in a surrounding fluid (as in [44]). One is interested in a detailed numerical simulation of the fluid dynamics in and around the drop (which may deform in time). In a second step also models for mass transport across the interface are implemented. Results of numerical simulations for a two-dimensional model problem can be found in [42].
- Grid coarsening strategies which are important in a time dependent setting should be implemented.
- We want to have a library of iterative solvers. Krylov subspace methods like GMRES and BiCGSTAB should be available. More multigrid components (other smoothers and other prolongation and restriction operators) have to be implemented. Some of these methods should be suitable for treating strongly nonsymmetric ("convection dominated") problems.
- More variants of the finite element discretization method are needed. For example, the streamline diffusion finite element method ([30, 34]) for the stable discretization of convection dominated flows.
- Other iterative methods of inexact Uzawa type will be considered. One needs, for example, methods which are robust with respect to the variation in the time step ([12, 24]). Methods as in [15, 16, 17] which can be used for convection dominated flows will be investigated.
- The work on parallelization will be continued.
- Stabilization techniques as in [27] will be implemented and tested.
- Error estimators for Navier-Stokes equations with large Reynolds numbers are needed. Techniques proposed in the literature (e. g., [18, 33]) will be studied.
- The development of the DROPS code will be problem driven. An important goal is to obtain satisfactory numerical simulations of incompressible flow models that are of interest in the interdisciplinary research project.
- A suitable user interface (with manual) has to be made available.

## References


[1] E. Bänsch, K. G. Siebert, *A Posteriori Error Estimation For Nonlinear Problems by Duality Techniques*, IAM Freiburg 12/1995

[2] P. Bastian, *Parallele adaptive Mehrgitterverfahren*, Teubner, Stuttgart, 1996.

[3] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert und C. Wieners, *UG - A flexible software toolbox for solving partial differential equations*, Computing and Visualization in Science 1:27–40, 1997.

[4] J. Bey, *Tetrahedral grid refinement*, Computing 55, pp. 355–378, 1995.

[5] J. Bey, *Finite-Volumen- und Mehrgitterverfahren für elliptische Randwertprobleme*, Advances in Numerical Methods, Teubner, Stuttgart, 1998.

[6] J. Bey, *Simplicial grid refinement: on Freudenthal's algorithm and the optimal number of congruence classes*, Numer. Math. 85, pp. 1–29, 2000.

[7] K. Birken, *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen*, Ph. D. thesis, University of Stuttgart, 1998.

[8] K. Birken, *Dynamic distributed data (DDD) — a software tool for distributed memory parallelization*, reference manual, `http://dom.ica3.uni-stuttgart.de/~ddd/`

[9] J. H. Bramble, *Multigrid Methods*, Pitman Research Notes in Mathematics Series 294. Longman Scientific & Technical, Harlow, 1993.

[10] J. H. Bramble, J. E. Pasciak, and A. T. Vassilev, *Analysis of the inexact Uzawa algorithm for saddle point problems*, SIAM J. Numer. Anal. 34/3, pp. 1072–1092, 1997.

[11] J. H. Bramble, J. E. Pasciak, and A. T. Vassilev, *Uzawa type algorithms for nonsymmetric saddle point problems*, Math. Comp. 69, pp. 667–689, 2000.

[12] J. H. Bramble, J. E. Pasciak, *Iterative techniques for time dependent Stokes problems*, Comput. Math. Appl. 33, No. 1-2, pp. 13–30, 1997.

[13] B. L. Chamberlain, *Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations*, technical report, University of Washington, 1998.
`http://www.cs.washington.edu/homes/brad/cv/pubs/degree/generals.html`

[14] W. Dörfler, *A convergent adaptive algorithm for Poisson's equation*, SIAM J. Numer. Anal. 33, pp. 1106–1124, 1996.

[15] H. C. Elman and D. Silvester, *Fast nonsymmetric iterations and preconditioning for Navier-Stokes equations*, SIAM J. Sci. Comp. 17, pp. 33–46, 1996.

[16] H. C. Elman, *Preconditioning of the steady-state Navier-Stokes equations with low viscosity*, SIAM J. Sci. Comp. 20, pp. 1299–1316, 1999.

[17] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Performance and analysis of saddle point preconditioners for the discrete steady-state Navier-Stokes equations*, Numer. Math. 90, pp. 665–688, 2002.

[18] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, *Introduction to adaptive methods for differential equations*, Acta Numerica 1995, pp. 105–158, Cambridge University Press, 1995.

[19] V. Girault and P.-A. Raviart, *Finite element methods for Navier-Stokes equations*, Springer, Berlin, 1986.

[20] G. Haase, *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen*, Teubner, Stuttgart, Leipzig, 1999.

[21] W. Hackbusch, *Multi-grid Methods and Applications*, Springer, Berlin, 1985.

[22] W. Hackbusch, *Iterative Solution of Large Sparse Systems of Equations*, Springer, Berlin, 1994.

[23] W. Hackbusch, *Theorie und Numerik elliptischer Differentialgleichungen*, Teubner, 1996.

[24] G. M. Kobelkov and M. A. Olshanskii, *Effective preconditioning of Uzawa type schemes for a generalized Stokes problem*, Numer. Math. 86, pp. 443–470, 2000.

[25] H. Müller-Krumbhaar, H. Emmerich, E. Brener and M. Hartmann, *Dewetting hydrodynamics in 1+1 dimensions*, Phys. Rev. E63, 2001.

[26] M. A. Olshanskii and A. Reusken, *Navier-Stokes equations in rotation form: A robust multigrid solver for the velocity problem*, SIAM J. Sci. Comput., accepted for publication.

[27] M. A. Olshanskii and A. Reusken, *Grad-Div stabilization for Stokes equations*, IGPM Report 208, 2001. Submitted.

[28] S. Osher and J. A. Sethian, *Fronts propagating with curvature dependent speed: algorithms based on Hamilton-Jacobi formulations*, J. Comp. Phys. 79, pp. 12–49, 1988.

[29] G. Karypis, K. Schloegel, and V. Kumar, *Parallel graph partitioning and sparse matrix ordering library. Version 2.0*, manual, University of Minnesota, 1998.
`http://www-users.cs.umn.edu/~karypis/metis/`

[30] A. Quarteroni and A. Valli, *Numerical Approximation of Partial Differential Equations*, Springer, Berlin, Heidelberg, 1994.

[31] 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.

[32] R. Rannacher, *Numerical analysis of nonstationary flow (a survey)*, in: Applications of Mathematics in Industry and Technology (V. C. Boffy and H. Neunzert, eds.), pp. 34–53, Teubner, Stuttgart, 1998.

[33] R. Rannacher, *Error control in finite element computations*, in: Proceedings NATO-Summer School "Error control and adaptivity in scientific computing" (H. Bulgak, C. Zenger, eds.), pp. 247–278, NATO science series, series C, Vol. 536. Kluwer, Dordrecht, 1999.

[34] H.-G. Roos, M. Stynes, and 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.

[35] A. Reusken, *Multigrid with matrix-dependent transfer operators for convection-diffusion problems*, in: Multigrid Methods 4, Proceedings of the fourth European Multigrid Conference (P. W. Hemker, P. Wesseling, eds.), International series of Numerical Mathematics Vol. 116, pp. 269–280, 1994.

[36] SFB 540 "Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems", http://www.lfpt.rwth-aachen.de/SFB540/

[37] M. Sussman, P. Smereka, and S. Osher, *A level set approach for computing solutions to incompressible two-phase flow*, J. Comp. Phys. 114, pp. 146–159, 1994.

[38] S. Turek, *Efficient solvers for incompressible flow problems: An algorithmic approach in view of computational aspects*, LNCSE Vol 6, Springer, Berlin, Heidelberg, 1999.

[39] R. Verfürth, *A Review of a Posteriori Error Estimation and Adaptive Mesh-refinement Techniques*, Wiley/Teubner, New-York, 1996.

[40] R. Verfürth, *A Posteriori Error Estimators for the Stokes Equations*, Numer. Math. 55, pp. 309–325, 1989.

[41] R. Verfürth, *A Posteriori error estimators for the Stokes equations II non-conforming discretizations*, Numer. Math. 60, pp. 235–249, 1991.

[42] M. A. Waheed, *Fluiddynamik und Stoffaustausch bei freier und erzwungener Konvektion umströmter Tropfen*, Ph. D. thesis RWTH Aachen. Shaker Verlag, Aachen, 2001.

[43] H. Yserentant, *Old and new convergence proofs for multigrid methods*, Acta Numerica 1993, pp. 285–326.

[44] H.-K. Zhao, B. Merriman, S. Osher, and L. Wang, *Capturing the behaviour of bubbles and drops using the variational level set approach*, J. Comp. Phys. 143, pp. 495–518, 1998.

[45] W. Zulehner, *Analysis of iterative methods for saddle point problems: a unified approach*, Math. Comp., to appear,2002.